Heuristics for Offline Rectangular Packing Problems

Frank Gerald Ortmann

Dissertation presented for the degree
PhD (Operations Research)
Department of Logistics, Stellenbosch University

Promoter: Prof JH van Vuuren

March 2010

Declaration

By submitting this dissertation electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the owner of the copyright thereof (unless to the extent explicitly otherwise stated) and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

March 1, 2010

Abstract

Packing problems are common in industry and there is a large body of literature on the subject. Two packing problems are considered in this dissertation: the *strip packing problem* and the *bin packing problem*. The aim in both problems is to pack a specified set of small *items*, the dimensions of which are all known prior to packing (hence giving rise to an *offline* problem), into larger objects, called *bins*. The strip packing problem requires packing these items into a single bin, one dimension of which is unbounded (the bin is therefore referred to as a *strip*). In two dimensions the width of the strip is typically specified and the aim is to pack all the items into the strip, without overlapping, so that the resulting packing height is a minimum. The bin packing problem, on the other hand, is the problem of packing the items into a specified set of bins (all of whose dimensions are bounded) so that the wasted space remaining in the bins (which contain items) is a minimum. The bins may all have the same dimensions (in which case the problem is known as the *single bin size bin packing problem*), or may have different dimensions, in which case the problem is called the *multiple bin size bin packing problem* (MBSBPP). In two dimensions the wasted space is the sum total of areas of the bins (containing items) not covered by items.

Many solution methodologies have been developed for above-mentioned problems, but the scope of the solution methodologies considered in this dissertation is restricted to *heuristics*. Packing heuristics follow a fixed set of rules to pack items in such a manner as to find good, feasible (but not necessarily optimal) solutions to the strip and bin packing problems within as short a time span as possible. Three types of heuristics are considered in this dissertation: (i) those that pack items into levels (the heights of which are determined by the heights of the tallest items in these levels) in such a manner that all items are packed along the bottom of the level, (ii) those that pack items into levels in such a manner that items may be packed anywhere between the horizontal boundaries that define the levels, and (iii) those heuristics that do not restrict the packing of items to levels. These three classes of heuristics are known as *level algorithms*, *pseudolevel algorithms* and *plane algorithms*, respectively.

A computational approach is adopted in this dissertation in order to evaluate the performances of 218 new heuristics for the strip packing problem in relation to 34 known heuristics from the literature with respect to a set of 1 170 benchmark problem instances. It is found that the new level-packing heuristics do not yield significantly better solutions than the known heuristics, but several of the newly proposed pseudolevel heuristics do yield significantly better results than the best of the known pseudolevel heuristics in terms of both packing densities achieved and computation times expended. During the evaluation of the plane algorithms two classes of heuristics were identified for packing problems, namely *sorting-dependent* and *sorting-independent* algorithms. Two new sorting techniques are proposed for the sorting-independent algorithms and one of them yields the best-performing heuristic overall.

A new heuristic approach for the MBSBPP is also proposed, which may be combined with level and pseudolevel algorithms for the strip packing problem in order to find solutions to the problem very rapidly. The best-performing plane-packing heuristic is modified to pack items into the largest bins first, followed by an attempted repacking of the items in those bins into smaller bins with the aim of further minimising wasted space. It is found that the resulting plane-packing algorithm yields the best results in terms of time and packing density, but that the solution differences between pseudolevel algorithms are not as marked for the MBSBPP as for the strip packing problem.

Uittreksel

Inpakkingsprobleme kom algemeen in die industrie voor en daar is 'n aansienlike volume literatuur oor hierdie onderwerp. Twee inpakkingsprobleme word in hierdie proefskrif oorweeg, naamlik die *strook-inpakkingsprobleem* en die *houer-inpakkingsprobleem*. In beide probleme is die doel om 'n gespesifiseerde versameling klein *voorwerpe*, waarvan die dimensies almal voordat inpakking plaasvind, bekend is (en die probleem dus 'n sogenaamde *aflyn*-probleem is), in een of meer groter *houers* te pak. In die strook-inpakkingsprobleem word hierdie voorwerpe in een houer, waarvan een dimensie onbegrens is, ingepak (hierdie houer word dus 'n *strook* genoem). In twee dimensies word die wydte van die strook gewoonlik gespesifiseer en is die doel om al die voorwerpe sonder oorvleueling op só 'n manier in die strook te pak dat die totale inpakkingshoogte geminineer word. In die houer-inpakkingsprobleem, daarenteen, is die doel om die voorwerpe op só 'n manier in 'n gespesifiseerde aantal houers (waarvan al die dimensies begrens is) te pak dat die vermorste of oorblywende ruimte in die houers (wat wel voorwerpe bevat) 'n minimum is. Die houers mag almal dieselfde dimensies hê (in welke geval die probleem as die *enkelgrootte houer-inpakkingsprobleem* bekend staan), of mag verskillende dimensies hê (in welke geval die probleem as die *veelvuldige-grootte houer-inpakkingsprobleem* bekend staan, afgekort as VGHIP). In twee dimensies word die vermorste ruimte geneem as die somtotaal van daardie deelareas van die houers (wat wel voorwerpe bevat) waar daar geen voorwerpe geplaas word nie.

Verskeie oplossingsmetodologieë is al vir die bogenoemde twee inpakkingsprobleme ontwikkel, maar die bestek van die metodologieë wat in hierdie proefskrif oorweeg word, word beperk tot *heuristieke*. 'n Inpakkingsheuristiek volg 'n vaste stel reëls waarvolgens voorwerpe in houers gepak word om so spoedig moontlik goeie, toelaatbare (maar nie noodwendig optimale) oplossings tot die strook-inpakkingsprobleem en die houer-inpakkingsprobleem te vind. Drie tipes inpakkingsheuristieke word in hierdie proefskrif oorweeg, naamlik (i) heuristieke wat voorwerpe langs die onderste randte van horisontale vlakke in die houers pak (die hoogtes van hierdie vlakke word bepaal deur die hoogtes van die hoogste item in elke vlak), (ii) heuristieke wat voorwerpe op enige plek binne horisontale stroke in die houers pak, en (iii) heuristieke waar inpakking nie volgens horisontale vlakke of stroke beperk word nie. Hierdie drie klasse heuristieke staan onderskeidelik as *vlakalgoritmes*, *pseudo-vlakalgoritmes* en *platvlakalgoritmes* bekend.

'n Berekeningsbenadering word in hierdie proefskrif gevolg deur die werkverrigting van die 218 nuwe heuristieke vir die strook-inpakkingsprobleem met die werkverrigting van 34 bekende heuristieke uit die literatuur te vergelyk, deur al die heuristieke op 1 170 toetsprobleme toe te pas. Daar word bevind dat die nuwe vlakalgoritmes nie 'n noemenswaardige verbetering in oplossingskwaliteit in vergeleke met soortgelyke bestaande algoritmes in die literatuur lewer nie, maar dat verskeie nuwe pseudo-vlakalgoritmes wel noemenswaardige verbeteringe in terme van beide inpakkingsdigthede en oplossingstye in vergeleke met die beste bestaande algoritmes in die literatuur lewer. Assessering van die platvlakalgoritmes het gelei tot die identifikasie van twee

deelklasse van algoritmes, naamlik sorteringsafhanklike- en sorteringsonafhanklike algoritmes. Twee nuwe sorteringstegnieke word ook vir die deelklas van sorteringsonafhanklike algoritmes voorgestel, en een van hulle lewer die algeheel beste inpakkingsheursitiek.

'n Nuwe heuristiese benadering word ook vir die VGHIP ontwikkel. Hierdie benadering kan met vlak- of pseudo-vlakalgoritmes vir die strook-inpakkingsprobleem gekombineer word om baie vinnig oplossings vir die VGHIP te vind. Die beste platvlakheuristiek vir die strook-inpakkingsprobleem word ook aangepas om voorwerpe eers in die grootste houers te pak, en daarna in kleiner houers te herpak met die doel om vermorste ruimte verder te minimeer. Daar word bevind dat die resulterende platvlakalgoritme die beste resultate in terme van beide inpakkingsdigtheid en oplossingstyd lewer, maar dat oplossingsverskille tussen die pseudo-vlakalgoritmes nie so opmerklik vir die VGHIP is as wat die geval met die strookinpakkings-probleem was nie.

Acknowledgements

Many people played a significant role in the work leading up to and during the writing of this dissertation. The author hereby wishes to express his deepest gratitude towards:

- Prof JH van Vuuren for his valuable insight, tireless work and support during the compilation of this dissertation, and financial assistance for trips to conferences and other events. It has been a great privilege to have been mentored by such an esteemed researcher for the past seven years;
- Dr N Ntene for the introduction to packing problems and many fun weeks of collaboration during the early stages of the research into the work presented in this dissertation;
- Prof JH Nel, Dr N Kourentzes and Prof SF Crone for their considerable assistance with the statistical methods included in this dissertation, and Dr MMC Lamont for supplying a second opinion on the validity of the tests;
- Dr I Nieuwoudt and Prof JH Nel for proofreading assistance;
- My colleagues on the second floor of the General Engineering Building and on the sixth floor of the CGW Schumann Building, for helping to create a great working environment;
- My friends and family, for their support, for the debates, for the laughter and for the distractions when I needed them;
- My parents, for their considerable financial and emotional support, and unwavering belief in me. This dissertation may not have been possible without it.

The Departments of Logistics and Mathematical Sciences at Stellenbosch University are hereby thanked for the use of their computing facilities and office space. The financial support of the South African National Research Foundation (NRF), in the form of a Grant Holder Bursary (under grant number GUN 2072815) and a Scarce Skills Bursary (with the Department of Labour) under grant number GUN 66976, and of Stellenbosch University in the form of three Merit Bursaries towards this research is hereby acknowledged. Any opinions or findings in this dissertation are those of the author and do not necessarily reflect the views of Stellenbosch University or the NRF.

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

Algorithms

2SMBSBP	Two-Stage Multiple Bin Size Bin Packing
B2F	Best Two Fit
B2FA	Best Two Fit, replace according to Area
B2FDH	Best Two Fit Decreasing Height
B2FW	Best Two Fit, replace according to Width
BFD	Best-Fit
BFD	Best-Fit Decreasing
BFDH	Best-Fit Decreasing Height
BFDH*	Modified Best-Fit Decreasing Height
BFLM	Best-Fit Left-Most
BFmLM	Modified Best-Fit Left-Most
BFmSN	Modified Best-Fit Smallest Neighbour
BFmTN	Modified Best-Fit Tallest Neighbour
BFS	Best-Fit with Stacking
BFSN	Best-Fit Smallest Neighbour
BFTN	Best-Fit Tallest Neighbour
BL	Bottom-Up Left-Justified
BLF	Bottom-Left Fill
DS	Double-Sided
E	Exact
FBL	Finite Bottom-Left
FBS	Finite Best-Strip
FC	Floor-Ceiling
FF	First-Fit
FFD	First-Fit Decreasing
FFDH	First-Fit Decreasing Height
FFDLR	First-Fit Decreasing using Largest bins, at end Repack to smallest possible bins
FFDLS	First-Fit Decreasing using Largest bins, but Shifting as necessary
FNF	Finite Next-Fit
GCS	Guillotine Cutting Stock
H	Heuristic
HBF	Hybrid Best-Fit
HFC	Hybrid Floor-Ceiling
HFF	Hybrid First-Fit
HNF	Hybrid Next-Fit
IBFD	Iterative Best-Fit Decreasing

IFFD	Iterative First-Fit Decreasing
KP	Knapsack Problem
KP_{TR}	Time Restricted Knapsack Problem
L	Level
LAW	Least Absolute Waste
LFLAW	Largest object First with Least Absolute Waste
LFLRW	Largest object First with Least Relative Waste
LRW	Least Relative Waste
M	Mixed
MH	Metaheuristic
NBL	Next Bottom-Left
NF	Next-Fit
NFD	Next-Fit Decreasing
NFDH	Next-Fit Decreasing Height
P	Plane
PL	Pseudolevel
S	Sleator
SAS	Size-Alternating Stack
SASm	Modified Size-Alternating Stack
SC	Stack Ceiling
SCR	Stack Ceiling with Re-sorting
SF	Split-Fit
SL	Stack Level
SP	Split
SPmF	Modified Split algorithm for Free packing
SPmG	Modified Split algorithm for Guillotine packing
UD	Up-Down
WF	Worst-Fit
WFD	Worst-Fit Decreasing
WFDH	Worst-Fit Decreasing Height

Packing Problems

1D	One-Dimensional
2D	Two-Dimensional
C&P	Cutting and Packing
MBSBPP	Multiple Bin Size Bin Packing Problem
OF	Oriented, Free
OG	Oriented, Guillotine
RF	Rotations allowed, Free
RG	Rotations allowed, Guillotine
SBSBPP	Single Bin Size Bin Packing Problem

Sorting Methods

DA	Decreasing Area
DADH	Decreasing Area, Decreasing Height
DADW	Decreasing Area, Decreasing Width
DH	Decreasing Height

DHDW	Decreasing Height, Decreasing Width
DHIW	Decreasing Height, Increasing Width
DW	Decreasing Width
DWDH	Decreasing Width, Decreasing Height
DWIH	Decreasing Width, Increasing Height
xRDWDH	Selected items (by number) sorted by DW, others by DH
xWDWDH	Selected items (by relative width) sorted by DW, others by DH

Miscellaneous

ANOVA	Analysis of Variance
CD	Critical Distance
CPU	Central Processing Unit
CSV	Comma-Separated Values
IQR	Interquartile Range
PC	Personal Computer
RAM	Random-Access Memory

List of Reserved Symbols

A number of symbols in this dissertation conform to the following convention:		
$\mathcal{A}$	Symbol denoting a **set**	(Calligraphic capitals)
$\mathbf{A}$	Symbol denoting a **matrix**	(Boldface capitals)
$\underline{A}$	Symbol denoting a **vector**	(Underlined capitals)

α^{SP}	Strip packing accuracy
$\mathtt{A}$	An arbitrary algorithm A
b	The number of bins available for packing
$\underline{\mathtt{B}}$	The number of bin sizes required for benchmark generation
$\mathcal{B}$	A set of bins
δ	The height/width percentage difference allowed for item joining
$\mathcal{E}$	A list of empty bins
F	Index value of the first item in a list
$\mathcal{F}$	A list of item-containing bins that have not been repacked
γ	Item area constraint for the generation of "nice" items
Γ^{SP}	Strip packing efficiency
Γ^{MS}	Multiple bin size bin packing efficiency
$h\left(\mathcal{R}_i\right)$	Height of arbitrary rectangle $\mathcal{R}_i$
$\mathtt{H}$	Strip height
$\mathcal{H}$	A list of horizontal cuts for the GCS algorithm
$\underline{\mathtt{I}}$	The number of items required for benchmark generation
$\mathcal{I}$	A set of items to be packed
L	Index value of the last item in a list
μ	MBSBPP utilisation
$\mathcal{M}$	A list of minimal cuts for the GCS algorithm
ν	Fitness for the MBSBPP
n	The number of items to be packed
N	Index value of the thinnest item in a list
$\mathcal{N}$	A list of narrow items
$\mathtt{OPT}$	The optimal strip packing height
$\mathcal{P}$	A list of packed items
ρ	Item aspect ratio constraint for the generation of "nice" items
R	A region in a strip or bin
$\mathcal{R}$	A list of previously empty bins into which items have been repacked
$\mathcal{S}$	A list of skyline segments
T	Index value of the tallest item in a list
$\mathcal{U}$	A list of super items for algorithm JOIN
$\mathcal{V}$	A list of vertical cuts for the GCS algorithm

$w(\mathcal{R}_i)$	Width of arbitrary rectangle $\mathcal{R}_i$
W	Index value of the widest item in a list
$\mathtt{W}$	Strip width
$\mathcal{W}$	A list of wide items
$\mathcal{Z}$	A list of items onto which further items may be packed for the SP algorithm

CHAPTER 1

Introduction

Contents

Cutting and packing (C&P) problems have probably existed for millennia, whether it be the packing of animals such as camels, mules or horses (see Figure 1.1(a)), seafaring vessels or early trains and vehicles (an example of which is shown in Figure 1.1(b)). These packing tasks would have been performed by means of intuition and experience on the part of the packer. However, C&P problems have evolved into a very active field of mathematical study since 1939, when Kantorovich [88] considered the minimisation of scrap — a one-dimensional (1D) cutting stock problem in which a number of short pieces of material are to be cut from a limited number of longer items (of which there are two sizes), and in which the aim is to minimise the waste that is trimmed from the cut pieces. Cutting stock problems were the most common type of C&P problems in the early literature (a detailed survey of early trim loss problems was performed by Hixman in 1980 [74]), with Eisemann [45] publishing work on the trim loss problem in 1957 and Gilmore and Gomory [57–59] considering the cutting stock problem in the 1960s.

(a) A pack mule circa 1876 [120].

(b) A delivery vehicle circa 1934 [29].

Figure 1.1: *Early applications of packing problems in the 19^{th} and 20^{th} centuries.*

This dissertation is a study of selected two-dimensional rectangular packing problems. The purpose of this chapter is to briefly introduce C&P problems in §1.1, while a more thorough introduction to the problems, and the scope of the dissertation, appear in the following chapter. A list of dissertation objectives is given in §1.3 and a general preview of the organisation of material in this dissertation is presented in §1.4.

(a) Cutting metal plates circa 2004 [136].

(b) A container vessel circa 2004 [124].

Figure 1.2: *Modern C&P problems in the 21st century.*

1.1 Background

There are many names for C&P problems in the literature. Dyckhoff [43, p. 145] lists the following as names that have appeared in the literature:

- cutting stock and trim loss problems (see Figure 1.2(a) for an example),
- bin packing, dual bin packing, strip packing, vector packing and knapsack (packing) problems,
- vehicle, pallet, container and car loading problems (see Figure 1.2(b) for an example),
- assortment, depletion, design, dividing, layout, nesting and partitioning problems, and
- capital budgeting, change making, line balancing, memory allocation and multiprocessor scheduling problems.

In cutting problems, large objects typically have to be cut into smaller items with the aim of minimising the waste that remains. This is why cutting problems are often called trim loss problems. Packing problems are typically characterised by large empty objects that should be filled by means of smaller items with the objective of either minimising the number of large objects utilised, maximising the value of the small items packed, or minimising the empty space remaining after the packing has taken place. Packing items into bins may be considered as "cutting" the empty space inside the bins, where the remaining empty space is "trim loss." Conversely, one may consider cutting problems as packing small items into the space occupied by large objects. Hence, there is a strong relationship between cutting and packing problems due to the duality of solid objects and the space that the objects occupy [43, pp. 148–149].

1.2 Informal Problem Description

Rectangular packing problems are common and widely studied. In these problems it is required to pack a specified set of *items* into one or more larger, rectangular objects (called *bins*) in such a manner that the items do not overlap each other and are completely contained within the bin. The aim is to pack these items in such a manner that the smallest amount of space remains unused within the bin(s), or that the packing height is a minimum.

Two of these packing problems are considered in this dissertation. The first is known as the *strip packing problem* and requires the packing of items into a bin of fixed width and unlimited height (referred to as a *strip*) in such a manner that the resulting height of the packed items is a minimum. The aim of the *bin packing problem* is to pack a specified list of items into bins, the dimensions of which are bounded, in such a manner the remaining wasted area in the bins that actually contain items is a minimum.

1.3 Dissertation Objectives

There are two aims of the work in this dissertation. The first is to develop new heuristics or improve known heuristics for the strip packing problem. These new and improved algorithms may then be used in conjunction with algorithms for bin packing problems in an attempt to find approximate solutions to packing problems as quickly as possible with an improvement in the utilisation of the bins. In order to realise these general aims, twelve specific objectives are pursued in this dissertation:

I To *perform* a literature survey of the different types of packing problems that have been published in order to define the family of packing problems for which the heuristics are developed.

II To *perform* a brief literature survey on methods traditionally used to solve packing problems.

III To *determine* suitable methods for comparing the performances of packing algorithms.

IV To *perform* a literature survey of known heuristics for the strip packing problem. This includes the following types of heuristics:

(a) level-packing heuristics,
(b) pseudolevel-packing heuristics, and
(c) plane-packing heuristics.

V To *improve* on the known heuristics documented in the literature, *i.e.* to find better

(a) level solutions,
(b) pseudolevel solutions, and
(c) plane solutions.

VI To *implement* these new and improved heuristics for the strip packing problem on a computer.

VII To *identify* suitable benchmarks for the strip packing problem in terms of which the qualities of solutions produced by the algorithms may be compared.

VIII To *perform* an appraisal of the strip packing algorithms in terms of the solution quality and execution times.

IX To *perform* a literature survey on heuristic methods for the bin packing problem; more specifically

(a) for the general case where the bins may not be of the same size, and

(b) for the specific case where the bins are all the same size.

X To *design* new heuristics for the bin packing problem that may improve on the known methods used to find solutions to these problems.

XI To *implement* these new and improved heuristics for the bin packing problems on a computer.

XII To *identify* suitable benchmarks for

(a) the general bin packing problem, and

(b) the special case where all bins are of the same dimensions,

in terms of which solution qualities of the solutions produced by the algorithms may be compared.

XIII To *perform* an appraisal of the algorithms in terms of their solution qualities and execution times for

(a) the general bin packing problem, and

(b) the special case where all bins are the same size.

XIV To combine the computer implementations of the algorithms designed and/or improved in this dissertation in order to establish a decision support system capable of solving strip and bin packing problems approximately.

1.4 Dissertation Organisation

In the second chapter of this dissertation, the scope of the problems under investigation is discussed in some detail. The chapter opens with an introduction to various classifications of C&P problems, which is concluded by the scope of C&P problems that will be covered in this dissertation. This is followed by a brief review of traditional solution methodologies for C&P problems, with a focus on heuristics, metaheuristics and exact methods and the introduction of the concept of pseudolevel-packing algorithms. This section is concluded by a description of the scope of the solution methodology that will be utilised in this dissertation. Finally, a selection of methods for the evaluation of algorithms for C&P problems, both theoretical and computational, are presented.

The third chapter is dedicated to level-packing algorithms for the strip packing problem. After a brief introduction, five known algorithms are described in some detail. A simple worked example, performance bounds (if they have been described in the literature), an estimation of the worst-case time complexity and some algorithmic variations or practical considerations to take into account when programming the algorithms are included in each case. This is followed by a description of two new level-packing algorithms, which are described in a similar manner to the known algorithms.

The fourth chapter opens with a number of practical considerations with respect to the implementation of pseudo-level algorithms. This is followed by a review of three known pseudolevel-packing algorithms. These algorithms are described verbally, by means of a pseudocode listing and are illustrated by means of a worked example. The worst-case time complexities of the algorithms are estimated, followed by suggestions for practical implementation considerations for these algorithms. Finally, five new algorithms are described in a manner similar to that of the three known algorithms.

The purpose of the fifth chapter is to review plane-packing algorithms from the literature. These algorithms are described in a manner similar to the descriptions of the algorithms in previous chapters. Some modifications to selected algorithms are proposed in an attempt to improve their packing efficiencies. The chapter is concluded by a new categorisation of plane-packing algorithms into those that depend strongly on the order in which items are sorted, and those algorithms that may yield good packing solutions regardless of the order in which the items are supplied to the algorithm. This distinction of algorithms allows for two novel methods of sorting items in an attempt to consistently find lower strip heights.

The sixth chapter contains an appraisal of all the strip packing algorithms of Chapters 3–5. The chapter opens with a description of 1 170 benchmark problem instances that are used to compare the algorithms. This is followed by a brief description of the statistical tests that are used to compare the algorithms and the results for the level-packing algorithms. These results are presented separately for related sets of algorithms, which are compared to one another, before the best algorithm from each set is selected for a final comparison. The pseudolevel-packing algorithms are separated into two sets, namely those that are guaranteed to yield guillotine layouts and those that no not adhere to the guillotine constraint. The plane-packing algorithms are separated into eight related sets and the best from each set are compared with the best from the two sets of pseudolevel-packing algorithms.

The *multiple bin size bin packing problem* (MBSBPP) and *single bin size bin packing problem* (SBSBPP) are introduced in the seventh chapter of the dissertation. A literature survey on these two problems is followed by a description of a new heuristic for the MBSBPP, including a pseudocode listing, a worked example and an estimation of the worst-case time complexity. Finally, modifications made to a plane-packing algorithm in order for it to find solutions to the MBSBPP are described.

The eighth chapter contains an appraisal of selected strip packing algorithms combined with the new algorithm for the MBSBPP. Brief descriptions of a number of benchmark problem instances for the MBSBPP are given, followed by a description of an algorithm that creates new benchmark instances for this problem. The level-packing algorithms that performed best in their respective sets are combined with the new algorithm for the MBSBPP in order to find results for a total of 1 357 benchmark instances. This is followed by an appraisal of selected guillotine and free-packing pseudolevel algorithms, and the results from the application of the plane-packing algorithm to the benchmark instances. Finally, selected algorithms from each set are all combined in an attempt to identify those algorithms that perform best for the MBSBP and SBSBP problems.

A summary of the contributions of the dissertation may be found in the penultimate chapter, as well as an appraisal of the impact of these contributions.

Various ideas for future work are presented in the final chapter.

CHAPTER 2

Dissertation Scope

Contents

In this chapter the scope of C&P problems covered in this dissertation is delimited and explained. In order to describe which C&P problems form part of this study, two typologies and two subtypologies for C&P problems are presented in §2.1. A systematic characterisation of C&P problems makes it possible to portray differences between problems accurately.

2.1 Classifications of Cutting and Packing Problems

In order to describe the C&P problems considered in this dissertation, three known typologies (the organisation of objects into categories according to certain criteria) of cutting and packing problems are reviewed. In 1990, Dyckhoff [43] attempted to sort the many forms of cutting and packing problems in the operations research literature into a typology that would be able to "unify the different use of notions in the literature and to concentrate further research on special types of problems" [43, p. 145]. This became necessary due to the vast variety of applications that had been found for C&P problems.

2.1.1 Dyckhoff's Typology for C&P Problems

Dyckhoff was the first person to attempt a categorisation of C&P problems into clearly-defined groups. In his paper he lists the various applications of C&P problems [43, p. 148]. Typical cutting applications may, for example, be found in the paper, metal, glass, wood, plastics, textiles and leather industries, while applications of packing or loading problems may be found in the industries dealing with vehicles, pallets of goods, containers, bins, *etc.* There also exist more abstract applications of C&P problems, such as packing in terms of weight dimensions (the knapsack problem), packing in terms of the time dimension (for scheduling problems), packing in terms of a financial dimension (budgeting), as well as packing in other dimensions, such as for memory allocation during data storage.

The four characteristics according to which Dyckhoff sorted C&P problems are dimensionality, the kind of assignment, the assortment of large objects and the assortment of small items. The *dimensionality* characteristic may be assigned one of four values, namely 1 for one-dimensional problems, 2 for two-dimensional problems, 3 for three-dimensional problems, or N $>$ 3 for N-dimensional problems. The *kind of assignment* characteristic may take one of two values: B (from the German word *Beladeproblem*) indicates that a selection of small items are to be used to determine packing patterns on all large items, while V (from the German word *Verladeproblem*) indicates that all small items are to be assigned to a selection of large items. There are three possible values for the *assortment of large objects* characteristic. Here a value of O indicates that only one object is to be packed, I represents the case where multiple identical large items are to be packed, and D indicates that multiple large items of various sizes are to be packed. Finally, for the *assortment of small items* characteristic, the value F indicates that there are few items (of different shapes), M indicates that there are many items of many different shapes, R denotes the case where there are many items with relatively few different shapes and C indicates that all small items are congruent (identical). This means that, according to Dyckhoff, $4 \times 2 \times 3 \times 4 = 96$ possible types of C&P problems exist.

Example 2.1 *Consider a company where various sizes of corrugated cardboard are kept in stock. Groups of items (cardboard boxes) of the same size are ordered by customers and these have to be cut from stock boards. The cutting problem in this situation may thus be classified as a* 2/V/D/R *problem, as the boards may be cut in only two dimensions, all items are cut from a selection of objects (boards), there are different sizes of objects and the items assortment is many items of relatively few shapes (each order is typically many items of the same shape).* ■

2.1.2 Wäscher's Improved Typology for C&P Problems

In 2006, Wäscher *et al.* [157] attempted to improve Dyckoff's proposed typology. Some weaknesses in his typology had become apparent during the fifteen years since Dyckhoff first proposed his typology for C&P problems. For example, Dyckhoff proposed that the strip-packing problem should be coded as 2/V/D/M, while other researchers preferred to code it as 2/V/O/M [157, p. 4]. Another problem, identified by Gradišar *et al.* [64, p. 1208], was that there was no possibility in the assortment of large objects for few groups of identical objects. They proposed a fourth possibility for this characteristic, labelled G (increasing the possible number of categories of C&P problems to 128). This eliminated the ambiguous notation for the case of items being packed into many variably-sized large objects versus the case where small items are packed into many large items that can be sorted into few groups of identically-sized items. Now, instead of both these problems being labelled as 1/V/D/R problems (which may be solved by means of

an item-oriented approach), the latter may be differentiated by being labelled as a 1/V/G/R problem (which may be solved by means of a pattern-oriented approach).

Example 2.2 *The addition of the labelling proposed by Gradišar et al. for an assortment of large objects consisting of few groups of many identical objects to Dyckhoff's typology allows the case in Example 2.1 to be relabelled. In that example the large objects were few groups of many identical boards. Thus, the cutting problem of Example 2.1 may now be labelled as a* 2/V/G/R *problem.* ■

Examples of pattern-oriented approaches to solving C&P problems are described by Eisemann [45], Gilmore and Gomory [57,58], Haessler and Talbot [66], Pandit [132] and Yanasse *et al.* [159]. These pattern-oriented approaches to C&P problems are typically solved by a column generation method. Lodi *et al.* [106] describe item-oriented algorithms for C&P problems as procedures in which each item is considered individually for packing into an object. These algorithms include, for example, the *First Fit*, *Best Fit*, *Next Fit* and *Worst Fit* algorithms, *etc.* and all their derivatives. Many of these algorithms are described in more detail later in this chapter. Other authors who have studied item-oriented packing include Coffman *et al.* [31], Lodi [101], Lodi *et al.* [106] and Ntene [125].

Wäscher *et al.* [157] agreed with Dyckhoff's dimensionality characterisation and left it unchanged. However, they felt that the German notations *Verladeproblem* and *Beladeproblem* should be avoided, leading to their proposal to change the problem categories to either input minimisation (a set of small items must be assigned to a set of large objects, such that all large objects are used), or output maximisation (a set of small items must be assigned to a set of large objects, such that all small items are used). Although Wäscher *et al.* did not develop codes for problem types in the same manner that Dyckhoff did, Ntene [125, p. 2] labelled the problem types IM (equivalent to Dyckhoff's V) and OM (equivalent to Dyckhoff's B), respectively. The assortment of small items characterisation was reduced to three options, namely identical small items (denoted by IS by Ntene, corresponding to Dyckhoff's C labelling), a weakly heterogeneous assortment of small items (many items are identical, labelled as W by Ntene, corresponding to Dyckhoff's R labelling) and S for a strongly heterogeneous assortment of small items (very few items are identical, labelled as S by Ntene, corresponding to Dyckhoff's M and F labels).

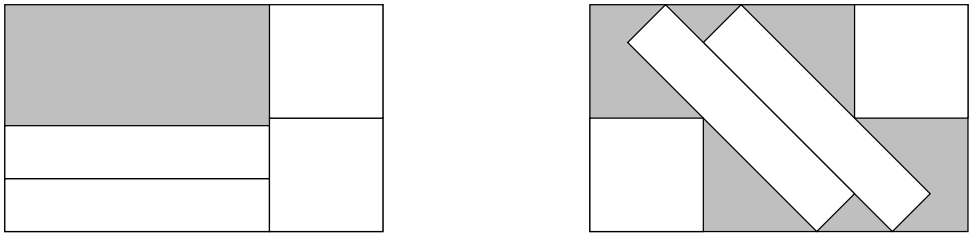
Although the improved typology is still similar to the original, it is in the assortment of large objects that the major change occurs. Here Wäscher *et al.* [157] group C&P problems into two categories, each with subcategories. The first set of problems is the class dealing with only one large object (labelled as O by Ntene), and this class may be partitioned into problems where all dimensions of the objects are fixed (subset labelled Oa by Ntene, identical to Dyckhoff's type O), and those where one dimension is variable (labelled Oo by Ntene, for strip packing problems) or where more dimensions of the object are variable (subset labelled Om by Ntene). The second set of problems are those dealing with several large objects (labelled Sf by Ntene). This set of problems may be divided into three subsets, namely those problems where the large objects are identical (labelled Si by Ntene, identical to Dyckhoff's type I), those problems where the objects are weakly heterogeneous (labelled Sw by Ntene) and those problems where the objects are strongly heterogeneous (labelled Ss by Ntene). The final two sets make up the grouping Dyckhoff called type D. Wäscher *et al.* claim that it does not seem important to differentiate between problems that deal with variable dimensions and those that do not within the Sf group, as only problems with fixed dimensions had been considered in literature [157, p. 8].

Example 2.3 *The case described in Example 2.1 may be characterised as a* 2/IM/Sw/W *problem using Ntene's labelling of Wäscher et al.'s typology. Wäscher et al. call it a* Multiple Stock Size Cutting Stock Problem. ■

There are some problems that Wäscher *et al.* do not include in their typology. These include

- problems where large objects are non-rectangular (such as disks).
- problems where items/objects are inhomogeneous, for example, stock material with defects.
- problems where items have irregular shapes (such as in the clothing industry).

These are considered problem variants. The packings in all problems of the typology by Wäscher *et al.* are also all assumed to be orthogonal, *i.e.* the edges of the small items must be parallel or perpendicular to the edges of the large objects into which packing occurs. Orthogonal and non-orthogonal packings of regular items are illustrated in Figure 2.1.



(a) Orthogonal Packing (b) Non-Orthogonal Packing

Figure 2.1: *A comparison of orthogonal and non-orthogonal packings (shaded areas denote empty spaces). A packing is orthogonal if the edges of a rectangular item are parallel or perpendicular to the sides of the large object.*

Wäscher *et al.* named all possible problems in their typology. These names may be found in Table 2.1 for output maximisation problems and in Table 2.2 for input minimisation problems.

2.1.3 Lodi's Subtypology for Packing Problems

Lodi *et al.* [101, 105] presented their own, limited typology for bin and strip packing problems. Their typology takes the form of three fields $dP|X|Y$, where d is the number of dimensions and P denotes the packing type (BP for bin packing or SP for strip packing). Later, the values such as CBP (for contiguous bin packing [110]), LSP and LBP (for level strip packing and level bin packing, respectively [107]) for P have been used. The value of $X \in \{\mathrm{O}, \mathrm{R}\}$ where O indicates that the items are oriented and R indicates that items may be rotated by 90°. More recently, Boschetti and Mingozzi [19, p. 136] suggested the addition of the letter M to represent the problem where a subset of the items to be packed are of type O (they may not be rotated) and the remaining items are of type R (they may be rotated). In many cases in industry, it must be possible to disentangle items from an object by means of edge-to-edge cuts that are either parallel or perpendicular to all edges of the object. Examples of guillotineable and non-guillotineable packings are shown in Figure 2.2. The value of $Y \in \{\mathrm{G}, \mathrm{F}\}$, where G indicates that the guillotine restriction applies and F indicates that the guillotine restriction does not apply. Lodi *et al.* [105] adopt the convention that an asterisk denotes multiple variants of a field.

Characteristics of Large Objects		Assortment of Small Items: Identical	Weakly Heterogeneous	Strongly Heterogeneous
All Dimensions Fixed	One Large Object	Identical Item Packing Problem (IIPP)	Single Large Object Placement Problem (SLOPP)	Single Knapsack Problem (SKP)
	Identical	N/A	Multiple Identical Large Object Placement Problem (MILOPP)	Multiple Identical Knapsack Problem (MIKP)
	Heterogeneous	N/A	Multiple Heterogeneous Large Object Placement Problem (MHLOPP)	Multiple Heterogeneous Knapsack Problem (MHKP)

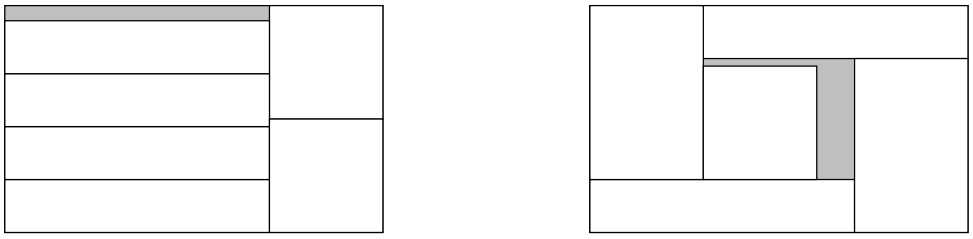
Table 2.1: *Improved typology of output maximisation problem types by Wäscher et al. (enough small items available to fill all large items) [157, p. 11]. The cases of multiple large items and identical small items is not a separate problem, as it may be reduced to identical packing problems for each of the objects [157, p. 10–11].*

Characteristics of Large Objects		Assortment of Small Items: Weakly Heterogeneous	Strongly Heterogeneous
All Dimensions Fixed	Identical	Single Stock Size Cutting Stock Problem (SSSCSP)	Single Bin Size Bin Packing Problem (SBSBPP)
	Weakly Heterogeneous	Multiple Stock Size Cutting Stock Problem (MSSCSP)	Multiple Bin Size Bin Packing Problem (MBSBPP)
	Strongly Heterogeneous	Residual Cutting Stock Problem (RCSP)	Residual Bin Packing Problem (RBPP)
One Large Object Variable Dimension(s)		Open Dimension Problem (ODP)	

Table 2.2: *Improved typology of input minimisation problem types by Wäscher et al. (enough large items for all small items to be packed) [157, p. 12].*

2.1.4 Ntene's Subtypology for Packing Problems

In 2007 Ntene [125, pp. 6–8] proposed a subtypology for packing problems. Her classification consists of six properties, denoted by $\boxed{\alpha \mid \beta \mid \chi \mid \gamma \mid \lambda \mid \tau}$. As with the other typologies, the first characteristic of a packing problem is the dimensionality. Thus, $\alpha \in \{1D, 2D, 3D, HoD\}$ denotes how many dimensions are considered in the problem. It is clear that $\alpha = 1D$ indicates

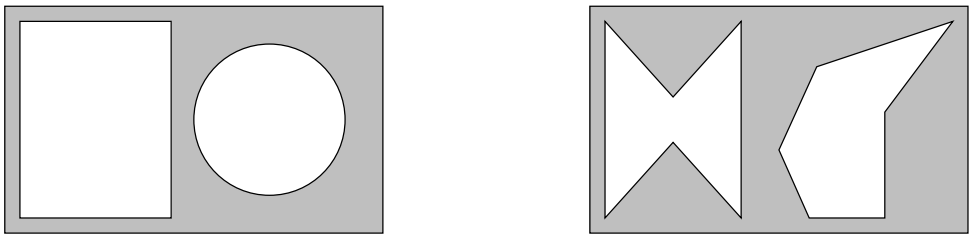


(a) Guillotineable Packing (b) Non-Guillotineable Packing

Figure 2.2: *A comparison of guillotineable and non-guillotineable (free) packings. If a packing arrangement is guillotineable, the items can be disentangled with edge-to-edge cuts that are parallel or perpendicular to all edges of the object.*

a one-dimensional problem (identical to the 1 of Dyckhoff and Wäscher *et al.*), *etc.*, while $\alpha = \text{HoD}$ indicates that the problem is in more than three dimensions (equivalent to the N of the other typologies).

Ntene's second characteristic is related to the shape of the small items. The items are either regular shapes, or they may be irregular, thus $\beta \in \{\text{I}, \text{R}\}$. Dyckhoff [43, p. 151] and Hopper and Turton [79, p. 259] define regular shapes to be those described by a few parameters (examples include rectangles and circles), while irregular shapes exhibit asymmetries and/or concavities. Thus, in Ntene's subtypology $\beta = \text{R}$ for the packing of exclusively regular shapes and $\beta = \text{I}$ if the problem includes the packing of irregular items. Examples of regular and irregular shapes may be seen in Figures 2.3(a) and 2.3(b), respectively.



(a) Regular items (b) Irregular items

Figure 2.3: *A comparison of regular and irregular items for cutting and packing problems (shaded areas denote empty spaces). Regular shapes may be described by few parameters, while irregular shapes typically exhibit asymmetries and/or concavities.*

In packing problems, the assortment of large objects into which the items must be packed consists of four possibilities. The first is that items have to be packed onto a strip. This is known as the strip packing problem (denoted by SP). A strip is most often a two-dimensional object (such as a roll of paper) where one dimension is unrestricted (the paper length, for example), while the other dimension is constant (typically the width). The second possibility is that the items have to be packed into many objects of equal fixed size (the bin packing or single-sized bin packing problem, labelled as MFB). However, sometimes these bins may not have the same dimensions; hence this problem is called the variable-sized bin packing problem (labelled as MVB). The final possibility is that a subset of items should be packed into a single bin so as to maximise the value of the items in the bin, or reduce the wasted space as much as possible. This is called the single bin packing problem, labelled SB by Ntene. Thus, the third characteristic $\chi \in \{\text{MFB}, \text{MVB}, \text{SB}, \text{SP}\}$.

The fourth characteristic indicates the nature of the information known about the items that have to be packed. Online problems (denoted by On) are packing problems where a list of items are packed one at a time without any prior knowledge about their size. Thus, only once an item has been packed, do the dimensions of the next item become known. In almost-online problems (labelled as Aon), some information may be known about the items before they are packed. An example of such information is the knowledge that new items are no larger than the previous item to be packed. Other information that may be known may be the number of items to be packed, or the maximum and/or minimum dimensions of the items. The offline case (denoted by Off) is the typical packing problem where the entire list of items is known before packing begins. Thus, $\gamma \in \{\text{Off}, \text{Aon}, \text{On}\}$.

There are many possible objectives for a packing. One might be to maximise the number of items to be packed (denoted by MaI), another may be to minimise the area of a packing (denoted by MiA). A minimisation of the number of bins is denoted by MiB, whilst minimising the cost of the packing is denoted by MiC. Furthermore, MiS indicates that the objective is to minimise the strip height for strip packing problems. Therefore, $\lambda \in \{\text{MaI}, \text{MiA}, \text{MiB}, \text{MiC}, \text{MiS}\}$.

The final characteristic may be used to accommodate further constraints for packing problems. Ntene split the final characteristic into four parts, so that $\tau = [\tau_o, \tau_p, \tau_m, \tau_g]$, where each unit of the vector is a binary variable.

- The first part, $\tau_o \in \{0, 1\}$ indicates whether the items may be rotated or not. A fixed orientation is represented by $\tau_o = 0$, while $\tau_o = 1$ indicates that rotation is allowed.
- The parameter $\tau_p \in \{0, 1\}$ is used to denote whether or not there are constraints on where items may be packed. Ntene uses the example of fragile items being part of an assortment of items. It is unwise, for example, to pack heavy items onto fragile items. If there is no restriction on the placement of items, then $\tau_p = 0$. Otherwise, $\tau_p = 1$ indicates that such a restriction on the placement of items exists.
- The third parameter $\tau_m \in \{0, 1\}$ indicates whether or not the shape of the small items may be modified while keeping another property constant (such as the area or volume). This is a phenomenon found in the scheduling of tasks on computers. The length and width may represent the time and computational resources required to complete a certain task, respectively. By lengthening the item, the task may take longer, but require less resources. On the other hand, widening the item means that more resources may be required, with the advantage of a possible reduction in the time required to complete the task. However, the changes in time and resource allocation computations do not change the number of computations (area) required to complete a task. For this parameter, $\tau_m = 0$ indicates that items may not be modified, while $\tau_m = 1$ means that modifications to item shapes are allowed.
- The last parameter $\tau_g \in \{0, 1\}$ indicates whether or not guillotineable cuts are required. If guillotine cuts are not required $\tau_g = 0$, while $\tau_g = 1$ indicates that any packing pattern must be guillotineable.

Finally, Ntene adopts the convention that an asterisk in any field indicates that the field is not restricted to any one of its possible values. This practice allows for a class of packing problems to be defined, rather than merely a specific packing problem. She goes on to state that the characteristics she defined are basic, but representative of packing problems. She purposefully constructed the classification to be flexible, so that by adding parameters to the final characteristic, for example, it is possible to define more problem types.

Example 2.4 *The case study briefly described in Example 2.1 is further characterised by four additional restrictions. The aim of the company is to minimise the waste remaining after the corrugated board is cut. Only regular items are considered, as orders for boxes are converted to orders for rectangular sheets of board and guillotineable cuts are required. Finally, orders for boxes are known before they are packed, so all items are known allowing for the use of offline algorithms. The fact that the board is corrugated allows for three further characterisations. Due to the direction of the flute of a corrugated board, the items may not be rotated. The large boards are homogeneous, so there are no restrictions on the packing of items and the shapes of the small items may not be modified. Thus, using Ntene's typology for packing problems, the situation is a*

2D	R	MVB	Off	MiA	$0, 0, 0, 1$

problem. ■

An attempt at a consolidation of the typologies by Dyckhoff and Wäscher *et al.*, and Ntene's subtypology is shown in Table 2.3. Tables 2.4 and 2.5 attempt to indicate how Dyckhoff's typology (labelled D and includes the improvement by Gradišar *et al.*), Ntene's interpretation of the typology of Wäscher *et al.* (labelled NW) and Ntene's subtypology (labelled N) fit together with respect to the problem types defined by Wäscher *et al.* (labelled WHS). These typologies may now be used to delimit the scope of C&P problems to be considered in this dissertation.

2.1.5 The Scope of C&P Problems in this Dissertation

In this dissertation, only two-dimensional C&P problems will be considered. Furthermore, the objective will always be to minimise the wasted space when items are packed into bins. Furthermore, a selection of small items will have to be packed into large objects (bins or strip) which will be assumed to be sufficient in number to accommodate all small items. This means that the problem is a *Verladeproblem* according to Dyckhoff [43, p. 154], or an input minimisation problem according to Wäscher *et al.* [157, pp. 6–7]. The C&P problems considered in this dissertation allow all large items to be identical, or they may vary in size. It will be assumed that the lengths and widths of all large objects and small items may *not* be modified.

According to Dyckhoff, the scope of C&P problems considered in this dissertation is the class of 2/V/*/* problems. Using Ntene's interpretation of the typology by Wäscher *et al.*, these problems are rather denoted by 2/IM/Oo,Sf/* problems. More specifically, problems called the *Open Dimension Problem* (ODP, commonly called the strip packing problem), the *Multiple Bin Size Bin Packing Problem* (MBSBPP) and (to a limited degree) the *single bin size bin packing problem* (SBSBPP) will be considered in this dissertation. According to the subtypology of Lodi *et al.* [101, 105], the problems under investigation include the 2SP|O|* and 2BP|O|* problems. Using Ntene's subtypology for packing problems one can place further restrictions on the scope of C&P problems to be considered. The problems in this dissertation may be represented by

2D	R	MFB/MVB/SP	Off	MiA/MiB/MiS	$0, 0, 0, ^*$

.

This classification indicates that only two-dimensional regular items are considered, that there are strips and multiple large objects (they may, or may not, all be the same size), that the entire list of small items is known before they are packed, that the area/cost of the packing should be minimised, that rotation of small items is not allowed, that there is no restriction on the placement of items, that the shapes of the items may not be altered and that guillotine cuts may or may not be required.

Dyckhoff [43]		Wäscher *et al.* [157]		Ntene [125]	
		Dimensionality			
1	One-dimensional	1	One-dimensional	1D	One-dimensional
2	Two-dimensional	2	Two-dimensional	2D	Two-dimensional
3	Three-dimensional	3	Three-dimensional	3D	Three-dimensional
N	N-dimensional	N	N-dimensional	HoD	Higher order dimensional
		Kind of Assignment			
B	Limited large objects, many items	OM	Output value maximisation	MaI	Maximise items packed
V	Limited items, many large objects	IM	Input value minimisation	MiA	Minimise packing area
				MiB	Minimise number of bins
				MiC	Minimise cost of packing
				MiS	Minimise strip height
		Assortment of Large Objects			
O	One large object	O	One object	SB	Single bin packing
I	Identical figures		Oa all dimensions fixed	SP	Strip packing
D	Different figures		Oo one variable dimension	MFB	Multiple bins of same size
G	Few groups of identical figures [64]		Om multiple variable dimensions	MVB	Multiple bins of different sizes
		Sf	Several figures		
			Si identical figures		
			Sw weakly heterogeneous		
			Ss strongly heterogeneous		
		Assortment of Small Items			
C	Congruent figures	IS	Identical small items	R	Regular items
R	Many items of few sizes	W	Weakly heterogeneous	I	Irregular items
M	Many items of many different sizes	S	Strongly heterogeneous		
F	Few items (of different figures)				

Table 2.3: *A consolidation of the typologies for C&P problems. Ntene's [125, pp. 6–8] subtypology for packing problems is incomplete here. The characteristic γ (which indicates whether the problem is offline, almost online or online) is not included, nor are the parameters τ_o, τ_p, τ_m, and τ_g. These characteristics are indicative of what is considered problem variants in the other typologies.*

Characteristics of Large Objects		Assortment of Small Items: Identical	Assortment of Small Items: Weakly Heterogeneous	Assortment of Small Items: Strongly Heterogeneous
All Dimensions Fixed	One Large Object	Identical Item Packing Problem D: */B/O/C NW: */OM/Oa/IS WHS: IIPP	Single Large Object Placement Problem D: */B/O/R NW: */OM/Oa/W WHS: SLOPP N: | * | * | SB | * | MaI | * * * * , , , |	Single Knapsack Problem D: */B/O/M NW: */OM/Oa/S WHS: SKP
	Identical	N/A	Multiple Identical Large Object Placement Problem D: */B/I/R NW: */OM/Si/W WHS: MILOPP N: | * | * | MFB | * | MaI | * * * * , , , |	Multiple Identical Knapsack Problem D: */B/I/M NW: */OM/Si/S WHS: MIKP
	Heterogeneous	N/A	Multiple Heterogeneous Large Object Placement Problem D: */B/(D/G)/R NW: */OM/S(w/s)/IS WHS: MHLOPP N: | * | * | MVB | * | MaI | * * * * , , , |	Multiple Heterogeneous Knapsack Problem D: */B/(D/G)/M NW: */OM/S(w/s)/S WHS: MHKP

Table 2.4: *Four typologies for output maximisation C&P problems (enough small items available to fill all large items). The description F (few items of different shapes) used by Dyckhoff [43, pp. 154] does not apply to output maximisation problems, as there have to be enough items to fill all large objects. Ntene's [125, pp. 6–8] subtypology is the same for each row as the sizes of small items are not taken into consideration. Dyckhoff's tyvology is denoted by D, Ntene's interpretation of the typology by Wäscher et al. [125, pp. 2] is denoted by NW, the typology by Wäscher et al. [157, pp. 11] is denoted by NHS and Ntene's subtypology is denoted by N.*

<table>
<tr><th colspan="2">Characteristics of Large Objects</th><th>Assortment of Small Items
Weakly Heterogeneous</th><th>Strongly Heterogeneous</th></tr>
<tr><td rowspan="3">All Dimensions Fixed</td><td>Identical</td><td colspan="2">Single Stock Size Cutting Stock Problem
D: */V/I/(R/C)
NW: */IM/Si/(IS/W)
WHS: SSSCSP
N: * | * | MFB | * | MiA/MiB/MiC | *,*,*,*</td><td>Single Bin Size Bin Packing Problem
D: */V/I/(F/M)
NW: */IM/Si/S
WHS: SBSBPP</td></tr>
<tr><td>Weakly Heterogeneous</td><td colspan="2">Multiple Stock Size Cutting Stock Problem
D: */V/G/(R/C)
NW: */IM/Sw/(IS/W)
WHS: MSSCSP
N: * | * | MVB | * | MiA/MiB/MiC | *,*,*,*</td><td>Multiple Bin Size Bin Packing Problem
D: */V/G/(F/M)
NW: */IM/Sw/S
WHS: MBSBPP</td></tr>
<tr><td>Strongly Heterogeneous</td><td colspan="2">Residual Cutting Stock Problem
D: */V/D/(R/C)
NW: */IM/Ss/(IS/W)
WHS: RCSP
N: * | * | MVB | * | MiA/MiB/MiC | *,*,*,*</td><td>Residual Bin Packing Problem
D: */V/D/(F/M)
NW: */IM/Ss/S
WHS: RBPP</td></tr>
<tr><td colspan="2">One Large Object Variable Dimension(s)</td><td colspan="2">Open Dimension Problem
D: */V/O/*
NW: */IM/O(o/m)/S
WHS: ODP
N: * | * | SP | * | MiC/MiS | *,*,*,*</td></tr>
</table>

Table 2.5: *Four typologies for input minimisation C&P problems (enough large items to accommodate all small items). Ntene's [125, pp. 6–8] subtypology is the same for each row as the sizes of small items are not taken into consideration. Dyckhoff's tyolology is denoted by D, Ntene's interpretation of the typology by Wäscher et al. [125, pp. 2] is denoted by NW, the typology by Wäscher et al. [157, pp. 11] is denoted by NHS and Ntene's subtypology is denoted by N.*

2.2 Packing Problem Solution Methodologies

The purpose of this section is to provide a brief introduction to the methodology typically employed to solve packing problems. Heuristics are approximate solution techniques that typically provide solutions in the least amount of time to the detriment of the solution quality. Exact methods find a best possible packing, but are slow and may be unable to provide solutions to large or realistically sized problem instances within reasonable time. The purpose of metaheuristics is to find a suitable compromise between heuristics and exact methods, in order to find, within a reasonable time period, solutions that are close to optimal. A metaheuristic is a high-level heuristic that delegates work to low-level heuristics in order to find good (but not necessarily optimal) solutions to optimisation problems.

2.2.1 Heuristics

The word heuristic is derived from the Greek word *heuristikein* or *heurisko*, which may be translated as "to find" or "to discover" [142]. Pearl [135, p. 3] states that "heuristics are

criteria, methods, or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal. They represent compromises between two requirements: the need to make such criteria simple and, at the same time, the desire to see them discriminate correctly between good and bad choices." Reeves [142, p. 6] remarks that what is now called a heuristic "would be better described as a *seeking* method, as it cannot guarantee to *find* anything." His formal description of a heuristic is "a technique which seeks good (*i.e.* near optimal) solutions at a reasonable computational cost without being able to guarantee either feasibility or optimality, or even in many cases to state how close to optimality a particular feasible solution is." However, this description of heuristics may include what are now known as metaheuristics — a tighter definition is required for the purposes of this dissertation. In this dissertation heuristics are those algorithms that pack items directly into bins, there is no high-level heuristic guiding other heuristics, nor is more than one repacking of items allowed. Furthermore, the solution is guaranteed to be feasible.

Consider a list $\mathcal{L}$ of n items. In the context of strip packing problems, heuristics consider these items one-by-one and attempt to pack them into a bin or strip so as to achieve a near-optimal solution without the guarantee that a solution is optimal. Two-dimensional bin/strip packing heuristics may belong to one or more of the following classes:

Plane Algorithms The class of algorithms that pack items anywhere in that region of the plane $\mathbb{R}^2$ defined by the boundaries of a bin or strip.

Pseudolevel Algorithms The sub-class of the plane algorithms where the permissible location of items is further restricted by levels. A level is defined by two parallel lines, called level boundaries, extending from one side of the bin or strip to the other, and perpendicular to the edges of the bin or strip. No item interior in a packing may be intersected by any level boundary.

Level Algorithms The sub-class of pseudolevel algorithms that produce packings in which at least one edge of each item coincides with the lower level boundary.

Figure 2.4 shows the heuristic classes are related. An example of a solution obtained by means of each class of heuristics applied to the same set of items (*i.e.* to the same packing instance) is shown in Figure 2.5.

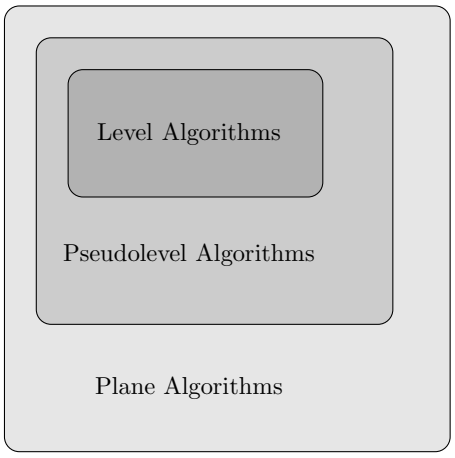
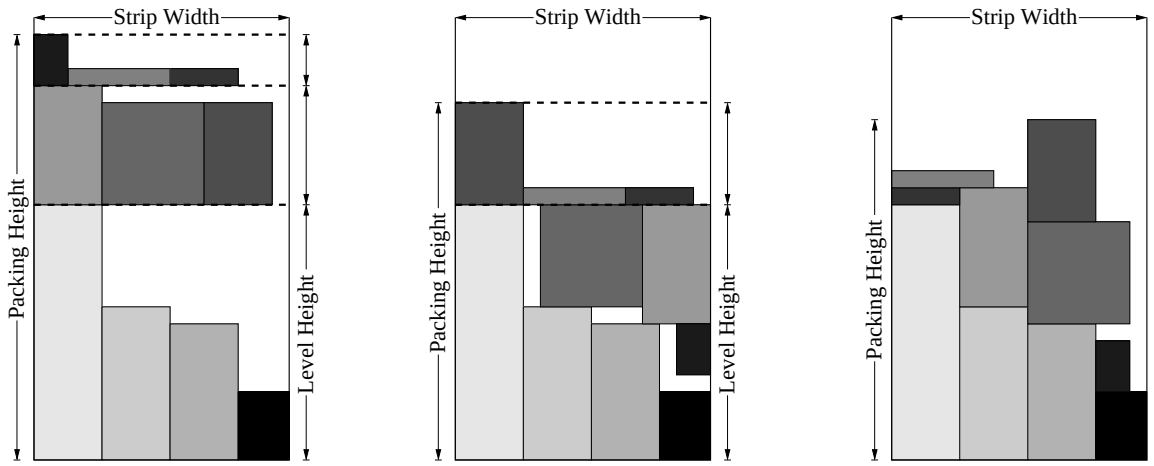


Figure 2.4: *A diagram showing the relationship between plane, pseudolevel and level algorithms.*



(a) Level solution (b) Pseudolevel solution (c) Plane solution

Figure 2.5: *Examples of solutions from the application of level, pseudolevel and plane algorithms to a strip packing problem. Level boundaries are shown as dotted lines.*

Two-dimensional offline packing heuristics typically sort the items in $\mathcal{L}$ in order of decreasing height. This has its origins in the heuristic solution to the offline one-dimensional bin packing problem (see Johnson *et al.* [84, 85]). Authors such as Coffman *et al.* [32], and Coffman and Shor [34] have adapted heuristics for the one-dimensional bin packing problem to the two-dimensional strip packing problem. Other strip packing algorithms, such as those by Lodi *et al.* [105, 106] and those by Ntene and Van Vuuren [125, 127] originated from these algorithms.

Other authors have considered the two-dimensional strip packing problem from a perspective independent of the one-dimensional bin packing problem. In this sense there are two classes of heuristics. The first class of heuristics sort $\mathcal{L}$ in order of decreasing width and allocate the items into sub-lists. These sub-lists are packed into certain regions of the strip. Coffman *et al.* [32], Sleator [148], Golan [62], Baker *et al.* [5], Coffman and Lagaris [33] and Coffman and Shor [34] designed such algorithms. The other class of heuristics simply pack items as far down and to the left as possible within a bin or strip. Baker *et al.* [6] published the first version of an algorithm in this class and authors such as Chazelle [25], Girkar *et al.* [60], Jakobs [83] and Liu and Teng [100] have subsequently made modifications to the original algorithm.

The heuristic approach to the single-size bin packing problem began with a combination of strip packing heuristics and heuristics for one-dimensional single-size bin packing. These algorithms (such as those by Chung *et al.* [28], Berkey and Wang [16] and Lodi *et al.* [106]) pack the items into a strip of width equal to the width of the bins, then pack the resulting levels of the strip into bins. In 1999, Lodi *et al.* [105] developed another form of two-phase packing strategy as an initialisation tool for a *tabu search* metaheuristic approach. Direct packing (or one-phase) heuristics, where items are packed directly into bins, have also been used (see Berkey and Wang [16] and Lodi *et al.* [105]), amongst other techniques.

Heuristics for the variable-sized bin packing problem are typically designed for the one-dimensional problem. Friesen and Langston [54] first suggested packing strategies that filled the largest bins first and then attempted to repack items into smaller bins. This was expanded on by Kang and Park [87] to include the FFD and BFD packing strategies. Chu and La [27] analysed four other packing strategies for this problem.

2.2.2 Metaheuristics

The prefix *Meta-* may be translated from Greek as "beyond", indicating that a metaheuristic is an algorithm that operates at a higher level than a heuristic. It may be defined as an algorithm "which basically tries to combine basic heuristic methods in higher level frameworks aimed at efficiently and effectively exploring a search space" [17, p. 270]. The term was first used by Glover [61, p. 541] and has since become widely adopted as the name for this class of algorithms. Blum and Roli [17, pp. 270–271] outline the fundamental properties characterising metaheuristics and summarise metaheuristics as "high level strategies for exploring search spaces by using different methods."

Where heuristics pack one item at a time from a list of items, metaheuristics often consider groupings of items and collectively assign to them a position in some manner. In packing, metaheuristics often make use of heuristics as decoding mechanisms. For example, a metaheuristic may find a suitable ordering of items and then some heuristic may use this ordering to pack the items into bins in order to test the quality of the solution [75, p. 19]. There are many types of metaheuristics and many of these algorithms are based on natural phenomena (see Reeves [142] and Blum and Roli [17] for a description of various metaheuristics). A selection of the more common metaheuristics specifically used in packing are briefly mentioned here.

One of the most widely used types of metaheuristics is the class of *evolutionary algorithms*. These algorithms represent solutions as chromosomes and make use of biological occurrences including reproduction, gene mutation, gene recombination and selection due to fitness in an attempt to find good solutions to combinatorial optimisation problems. *Genetic algorithms* are popular evolutionary algorithms that typically represent solutions as binary strings. These strings typically undergo cross-over and mutation operations to produce "offspring", which are subsequently tested for fitness. The fittest stings (the best approximate solutions) survive and are allowed to reproduce, while the weakest are discarded. Authors such as Falkenauer and Delchambre [48, 49], Hwang *et al.* [81], Jakobs [83], Runarsson *et al.* [144], Dagli and Poshyanonda [38], Liu and Teng [100], Hopper and Turton [75, 77–80], Valenzuela and Wang [153], Bortfeldt [18], Gonçalves and Resende [63] and Burke *et al.* [23] have studied genetic algorithms as a tool for solving packing problems.

Another common metaheuristic that has recently been applied to packing problems is *simulated annealing*, first published by Kirkpatrick *et al.* [93] (see also Eglese [44]). It originates in the annealing technique used in metallurgy, where metals are carefully heated and cooled in order to promote a better crystalline structure in the metal. When applied to packing problems the positions of atoms may be analogous to the positions of items in bins. The system begins with a certain temperature at which the items are free to move around. The movement of items becomes more constrained as the temperature decreases until they are no longer able to move. Authors that have studied simulated annealing for two-dimensional rectangular packing problems include Dowsland [41], Lai and Chan [95], Faina [47], Hopper and Turton [75, 77–80], Beisiegel *et al.* [13] and Burke *et al.* [23].

The third common metaheuristic in the packing problem literature is *tabu search*. It is a local search technique that maintains a list of recent moves or solution modifications that have been applied during the search and marks the reversal of these moves as "tabu", thereby preventing cycling during the search. Starting with an initial solution (that may be found by a heuristic), moves are made from one solution to another until a stopping criterion is reached. Examples of these moves may include an attempt to repack an item from one bin into another. Authors that have evaluated the tabu search method for two-dimensional rectangular packing problems

include Lodi *et al.* [101,104,105,108], Hopper and Turton [75,77–80], Alvarez-Valdes *et al.* [1–3], Burke *et al.* [23] and Pureza and Morabito [140].

2.2.3 Exact Methods

Exact packing methods are often called deterministic methods and are guaranteed to find an optimal solution to a problem (given sufficient time and a feasible region), while heuristics and metaheuristics attempt to find optimal solutions, but are not guaranteed to do so. Exact methods may often be too slow to solve large problem instances, but may typically be used to solve smaller ones. In pattern generation methods, an exhaustive list of columns is generated that represent all possible patterns formed by items within the bins. Linear or integer programming is then used to determine how many of each pattern to produce in order to fulfil the demand. In 1965 Gilmore and Gomory [59] expanded their earlier work on one-dimensional packing problems [57, 58] to two-dimensions with such a proposal. However, although this is a valid approach, it was deemed impractical with respect to finding exact solutions rapidly due to the difficulty of exhaustive pattern generation in two dimensions.

Gilmore and Gomory subsequently limited the number of permissible patterns by introducing a guillotine constraint. By only allowing two-stage cutting patterns they further reduced the number of permissible patterns by packing onto levels. If each item in the level was of the same height no trimming was required, otherwise a third stage cut was allowed, where items were not of the same height and trimming took place. Lodi *et al.* [107] designed an integer programming approach towards the level strip/bin packing problem employing combinatorial bounds that may be adapted to allow item rotation. Belov [11] did further work on these problems.

Martello *et al.* [110–112] developed a branch-and-bound algorithm that is able to solve the two-dimensional bin packing problem and strip packing problem, respectively, to optimality. Careful use of bounds and the use of heuristics to find good initial solutions made it possible for them to solve some small problem instances to optimality within a one hour time limit on a Pentium III 800 MHz computer [110, p. 318]. Scheithauer [146] made use of the equivalence and dominance of packing patterns to reduce the number of branches that the branch-and-bound algorithm evaluates. Pisinger and Sigurd developed a *branch-and-price* approach to the two-dimensional single-size bin packing problem [138] and also for the variable-size bin packing problem [137]. Hifi and Zissimopoulos [72] improved on the exact algorithm for the constrained two-dimensional cutting problem by Christofides and Whitlock [26], while Cui [35] developed an exact algorithm for the constrained two-dimensional cutting problem. Recently Cui *et al.* [37] also developed a recursive algorithm incorporating branch-and-bound techniques to solve the strip packing problem where guillotine cuts are required and rotations are allowed. A year later Bekrar and Kacem [10] developed a dichotomic algorithm to solve the guillotine strip packing problem to optimality, and Kenmochi *et al.* [90] developed exact algorithms for the strip-packing problem that allowed for oriented and rotational packing.

2.2.4 Scope of Methodology in this Dissertation

In the remainder of this dissertation only heuristic approaches to solving packing problems will be researched. Level, pseudolevel and plane packing algorithms will be considered that are capable of producing approximate solutions to problems within the scope described in §2.1.5.

2.3 Evaluation of Packing Algorithms

Two types of approaches exist that may be followed in order to evaluate the effectiveness of a packing algorithm. One is to perform a theoretical analysis of the algorithm by comparing an optimal solution to a hypothetical problem with the solution the algorithm provides (the theoretical analysis may also include a time analysis). The second approach is to evaluate implementations of the algorithms on a set of benchmark problem instances.

2.3.1 Theoretical Evaluation Methods

The *complexity* of an algorithm is the amount of computational resources required by a computer to employ an algorithm. There are typically two measures that define algorithmic complexity: the *space complexity* and the *time complexity*. The space complexity measures the amount of memory required to execute an algorithm and the time complexity measures the number of basic operations executed by an algorithm in order to quantify the expected time required to execute an algorithm. Only time complexity is considered in the remainder of this dissertation[1]. Let $T(n)$ denote the number of basic operations required to execute an algorithm for a problem of size n. If $T(n)$ increases slowly as n increases, then the algorithm may be useful for large values of n. However, if $T(n)$ grows very fast as n increases (such as when $T(n)$ is an exponential or factorial function of n), then the algorithm may not be able to solve moderately large instances of the problem within a realistic time span [143]. If an algorithm is known which can solve a problem within polynomial time (such as when $T(n)$ is a logarithmic, linear or quadratic function), then the problem is called *tractable*. Otherwise the problem is called *intractable* [24].

It is difficult to determine the exact number of basic operations that are required by an algorithm as it often depends on input size and conditional statements such as *if*-statements and *while*-loops. Instead, the *worst-case time complexity* is used to indicate the relationship between input size n and the time required to solve the problem in the worst case. Due to the difficulty of calculating exact upper bounds on $T(n)$, asymptotic upper bounds as $n \to \infty$ are preferred as a description of the worst-case growth behaviour [68]. If $g(n)$ is a function such that $T(n) \leq c_1\, g(n)$ for all n larger than some $n_1 \in \mathbb{N}$ (for some $c_1 \in \mathbb{R}^+$), then $g(n)$ is referred to as the *asymptotic upper bound* on $T(n)$ as $n \to \infty$. The most common form of expressing this type of bound is by means of the so-called "Big $\mathcal{O}$" notation, which takes the form $T(n) = \mathcal{O}(g\,(n))$ [24]. Table 2.6 indicates how algorithmic time complexity may be classified.

Some authors, including Johnson *et al.* [84, 85], Baker *et al.* [5, 6], Sleator [148], Coffman *et al.* [32], Brown [20], Golan [62], Chung *et al.* [28] and others, have theoretically evaluated the quality of solutions produced by packing heuristics. Consider a list $\mathcal{L}$ of items to be packed. Let $\texttt{OPT}(\mathcal{L})$ denote the value of some performance measure corresponding to an optimal solution to the problem. This measure is typically the strip height for strip packing problems, or the number of bins required to pack items for single-sized bin packing problems. The same performance measure evaluated for the solution provided by algorithm A is denoted by $\texttt{A}(\mathcal{L})$. An *absolute performance bound* for algorithm $\texttt{A}$ is the worst possible solution for any list $\mathcal{L}$ and has the form

$$\texttt{A}\,(\mathcal{L}) \leq \beta\, \texttt{OPT}\,(\mathcal{L})\,,$$

where $\beta \in \mathbb{R}$ and $\beta \geq 1$. An *asymptotic performance bound* for algorithm $\texttt{A}$ takes the form

$$\texttt{A}\,(\mathcal{L}) \leq \beta\, \texttt{OPT}\,(\mathcal{L}) + \gamma,$$

[1] In the current technological environment memory is cheap. Time may be an expensive resource and a common aim is to minimise the time required to complete calculations.

Rate of Growth	Name	If n is doubled
$\mathcal{O}(1)$	Constant	No change
$\mathcal{O}(\log\log n)$	Iterated logarithmic	Small time increments
$\mathcal{O}(\log n)$	Logarithmic	Constant time increments
$\mathcal{O}(n)$	Linear	Time is doubled
$\mathcal{O}(n\log n)$	Linearithmic	Slightly more than double time
$\mathcal{O}\left(n^2\right)$	Quadratic	Time increases 4-fold
$\mathcal{O}\left(n^3\right)$	Cubic	Time increases 8-fold
$\mathcal{O}(n^c)$	Polynomial	Time increases 2^c-fold
$\mathcal{O}(c^n)$	Exponential	Time required is squared
$\mathcal{O}(n!)$	Factorial	Time increases $(2n)!/n!$-fold
$\mathcal{O}(n^n)$	Super-exponential	Time increases extremely quickly

Table 2.6: *Classifications of algorithmic time complexity [143]. The constant $c \in \mathbb{R}^+$.*

where $\gamma \in \mathbb{R}$. Coffman *et al.* [32, p. 809] remark that an asymptotic bound is of greater interest than an absolute performance bound, because the latter often applies only to small, very specialised examples of items. Instead, an asymptotic performance bound characterises the performance of the algorithm as the ratio of the height of an optimal solution $\texttt{OPT}(\mathcal{L})$ to the height of the tallest item tends to infinity. Such a bound is often established for an item set where the maximum item height is 1. However, any height may be used (see Sleator [148] for an example), as it only affects the additive constant γ; the multiplicative bound β remains unchanged [32].

2.3.2 Computational Evaluation Methods

Dowsland and Dowsland [42, p. 8] comment that while average or worst-case performance bounds are useful guidelines, it is best to determine an algorithm's usefulness by testing it on data sets typical to the intended problem. Repositories are available on the internet where benchmark data sets for packing problems are stored for the purpose of evaluating algorithms. These include the benchmarks from the EURO Special Interest Group on Cutting and Packing (ESICUP) [46], the instances at PackLib^2 [50] or from the online repository by Van Vuuren and Ortmann [154].

In order to measure the solution quality of a strip packing algorithm, called Algorithm $\texttt{A}$, the packing height achieved by the algorithm (denoted by $\texttt{A}(\mathcal{I})$, where $\mathcal{I}$ denotes a set of items) may be divided by the packing height associated with an optimal solution (denoted by $\texttt{OPT}(\mathcal{I})$). This result is called the *strip packing accuracy* α^{SP}, and is defined as

$$\alpha^{\text{SP}}_{\texttt{A}}(\mathcal{I}) = \frac{\texttt{A}(\mathcal{I})}{\texttt{OPT}(\mathcal{I})}.$$

However, optimal solutions are not known for all benchmark instances. In order to compare algorithms by means of benchmark instances for which optimal solutions are not known, the packing height of the algorithms may be divided by valid lower bounds (such as those by Martello *et al.* [110]). In order to compare algorithms with respect to both solution quality and the speed at which the solution is found, the *strip packing efficiency* Γ^{SP} is defined as

$$\Gamma^{\text{SP}}_{\texttt{A}}(\mathcal{I}) = \frac{\texttt{OPT}(\mathcal{I})}{\texttt{A}(\mathcal{I})} \times \left(\frac{\tau_{\mathcal{I}}}{t^{\text{A}}_{\mathcal{I}}}\right)^{\frac{1}{\ell}},$$

where $\ell \in \mathbb{Z}^+$, $t_{\mathcal{I}}^{\mathtt{A}}$ denotes the time required by algorithm $\mathtt{A}$ to find a solution for the items in $\mathcal{I}$ and where $\tau_{\mathcal{I}}$ denotes the time required by the fastest algorithm in the comparison group to find a solution to the same problem. The influence of time on Γ^{SP} decreases as the value of ℓ increases. An algorithm may be labelled as more efficient than a second algorithm for a given value of ℓ if its efficiency is larger. If no optimal solution is known for a strip packing benchmark instance, the packing height corresponding to an optimal solution may be replaced with a valid lower bound (see Martello *et al.* [110]).

Two measures appear in the literature (see [75]) for the evaluation of packing solutions to the multiple bin size bin packing problem. The *utilisation* μ of a packing is the total area of the items in $\mathcal{I}$ (denoted by $A(\mathcal{I})$) divided by the area of the bins that contain items (denoted by $A(\mathcal{B}^{\mathcal{I}})$, where $\mathcal{B}$ denotes the set of bins), that is

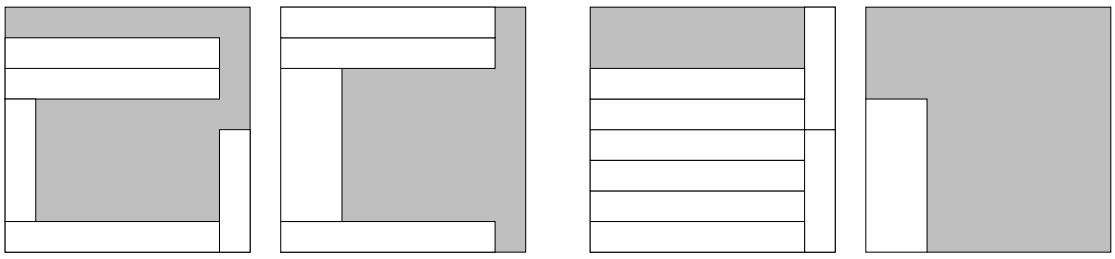
$$\mu_{\mathtt{A}}(\mathcal{P}) = \frac{A(\mathcal{I})}{A(\mathcal{B}^{\mathcal{I}})}.$$

The objective in the multiple bin size bin packing problem is to maximise μ.

The *fitness* ν of a solution to the multiple bin size bin packing problem is a measure that aims to reward algorithms for dense packing of bins. This allows for the separation of algorithms when their utilisations are equal for all solutions. A solution in which most bins are densely packed and one bin is not, would typically achieve a higher fitness score than an algorithm that packs bins less densely. Figure 2.6 illustrates how fitness may be used to differentiate between two solutions where utilisation cannot. The fitness of a solution is defined as

$$\nu_{\mathtt{A}}(\mathcal{P}) = \frac{\sum_{i=1}^{M} \left(\frac{A(\mathcal{I}^{\mathcal{B}_i})}{A(\mathcal{B}_i)}\right)^k}{M},$$

where $A\left(\mathcal{I}^{\mathcal{B}_i}\right)$ denotes the total area of the items packed into bin $\mathcal{B}_i$, M is the number of bins that contain items in the solution and typically $k = 2$ (as in Hopper [75]).



(a) Solution S_B (b) Solution S_G

Figure 2.6: *A comparison of two solutions to a packing problem. The bins have height and width 40. Both solutions have a utilisation of 0.479, while S_B has a fitness of 0.220 and S_G has a fitness of 0.317. A higher fitness is more desirable as it indicates a higher probability of the wasted space being of such a size that it may be used for further packing in the future. This may be important in, for example, trim loss or stock cutting problems, where the offcuts (wasted space) may be re-employed as raw material at a later stage.*

In order to compare the algorithms in terms of both solution quality and time, the *multiple bin size bin packing efficiency* Γ^{MS} is defined as

$$\Gamma_{\mathtt{A}}^{\mathrm{MS}}(\mathcal{P}) = \mu^{\mathtt{A}}(\mathcal{P}) \times \left(\frac{\tau_{\mathcal{P}}}{t_{\mathcal{P}}^{\mathtt{A}}}\right)^{\frac{1}{\ell}}$$

for algorithm $\mathtt{A}$.[23]

2.3.3 Scope of Algorithmic Evaluation in this Dissertation

In the remainder of this dissertation algorithms are almost entirely compared by means of computational measures. Comparison tools such as the strip/single bin size bin packing accuracy and efficiency, as well as the multiple bin size bin utilisation, fitness and efficiency, listed in §2.3.2, will be employed to compare heuristics that are reviewed or introduced in this dissertation. The worst-case time complexity of algorithms is the only theoretical evaluation method used to evaluate the heuristics. However, other theoretical evaluation methods performed on known heuristics will be presented if they have appeared in the literature.

2.4 Chapter Summary

The aim of this chapter has been to introduce the notion of cutting and packing problems, and to delimit the scope of this dissertation. After a short introduction, the typologies of Dyckhoff [43], Wäscher *et al.* [157], Lodi *et al.* [101, 105] and Ntene [125] for such problems were discussed in some detail, in fulfilment of Dissertation Objective I as stated in §1.3. This made it possible to clarify the scope of cutting and packing problems covered in the remainder of this dissertation.

After the discussion on typologies, a brief introduction to methods for solving packing problems highlighted the use of three classes of methods, namely heuristics, metaheuristics and exact methods in fulfilment of Dissertation Objective II. This was followed by a introduction to theoretical and computational methods for the analysis of algorithms designed to solve packing problems, in fulfilment of Dissertation Objective III as stated in §1.3. The theoretical analyses included worst-case algorithmic time complexity and two worst-case measures for the solution quality; the *absolute performance bound* and the *asymptotic performance bound*. Thereafter, the methods for the computational analysis of algorithms were presented in some detail.

[2] If the fitness of solutions is the preferred measure of solution quality, the *multiple bin size bin packing score* for algorithm $\mathtt{A}$ is given by

$$\Psi_{\mathtt{A}}^{\mathrm{MS}}(\mathcal{P}) = \nu^{\mathtt{A}}(\mathcal{P}) \times \left(\frac{\tau_{\mathcal{P}}}{t_{\mathcal{P}}^{\mathtt{A}}}\right)^{\frac{1}{\ell}}.$$

This score may be employed as a tool for the comparison of algorithms in terms of both fitness and time, but will not be used in this dissertation.

[3] In order to measure the solution quality of a single bin size bin packing algorithm, called Algorithm $\mathtt{A}$, the number of bins required to pack all items (denoted by $\mathtt{A}(\mathcal{P})$, where $\mathcal{P}$ is an abbreviation of the problem where the set of items $\mathcal{I}$ are to be packed into the set of bins $\mathcal{B}$) may be divided by the number of bins required to pack the items (called *packed bins*) in an optimal solution (denoted by $\mathtt{OPT}(\mathcal{P})$). This ratio is called the *single bin size bin packing accuracy* α^{SS}, and defined as

$$\alpha_{\mathtt{A}}^{\mathrm{SS}}(\mathcal{P}) = \frac{\mathtt{A}(\mathcal{P})}{\mathtt{OPT}(\mathcal{P})}.$$

However, an optimal solution may not be known for all benchmark instances. In order to compare a number of algorithms with respect to benchmark instances for which optimal solutions are not known, the number of bins required to pack the items in a solution found by the algorithms may be divided by a valid lower bound. In order to compare algorithms with respect to solution quality and the speed at which the solution is found, the *single bin size bin packing efficiency* Γ^{SS} is defined as

$$\Gamma_{\mathtt{A}}^{\mathrm{SS}}(\mathcal{P}) = \frac{\mathtt{OPT}(\mathcal{P})}{\mathtt{A}(\mathcal{P})} \times \left(\frac{\tau_{\mathcal{P}}}{t_{\mathcal{P}}^{\mathtt{A}}}\right)^{\frac{1}{\ell}}$$

for algorithm $\mathtt{A}$. These will not be used in this dissertation.

CHAPTER 3

Level Strip Packing Heuristics

Contents

Various level-packing approaches to the two-dimensional strip packing problem are considered in this chapter. These algorithms partition the strip by means of horizontal lines, thereby creating levels. A new level is initialised above the topmost level if items no longer fit on lower levels. The workings of a number of known heuristics following this approach are described and two new heuristics are introduced. All algorithms in this chapter yield a two-stage cutting pattern. This means that when the items are cut, the pattern will have to be turned by ninety degrees no more than twice in order for the items to be separated by means of a hypothetical set of linear blades. The first cut would be along the lines representing the levels and the second set of cuts would be to separate the items in each level from each other. This method of describing the solution pattern does not include the turn required for the trimming of waste from the items.

3.1 Introduction

An example set $\mathcal{I}$ of items is used to illustrate the working of the various algorithms presented in this chapter and the next. The dimensions of the items are listed in Table 3.1 and the items in $\mathcal{I}$ are shown in Figure 3.1.

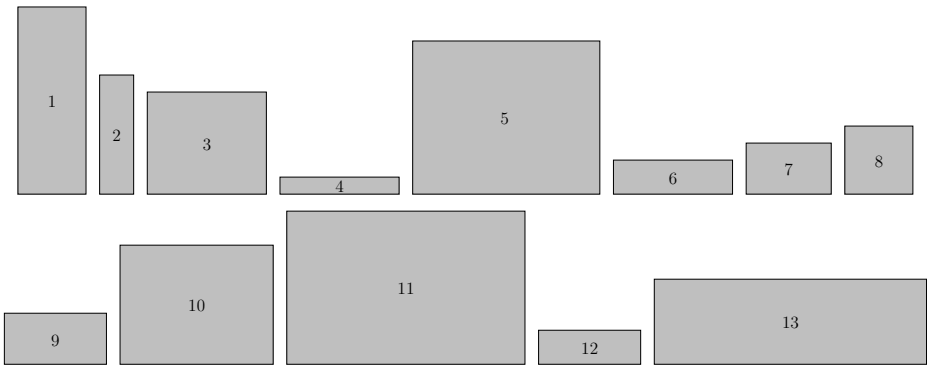


Figure 3.1: *The item set $\mathcal{I}$ used for illustrative purposes.*

Item, $\mathcal{I}_i$	1	2	3	4	5	6	7	8	9	10	11	12	13
Height, $h(\mathcal{I}_i)$	11	7	6	1	9	2	3	4	3	7	9	2	5
Width, $w(\mathcal{I}_i)$	4	2	7	7	11	7	5	4	6	9	14	6	16

Table 3.1: *Dimensions of the items in $\mathcal{I}$.*

It is often useful to sort the items before packing them. Johnson *et al.* [84,85] studied the *next-fit decreasing*[1] (NFD) algorithm [84], the *first-fit decreasing* (FFD) [85] and *best-fit decreasing* (BFD) algorithms [85], as well as the unsorted versions of these algorithms. They concluded that the worst-case result for each algorithm with sorted items is better than the worst-case scenario for the corresponding algorithm with unsorted items. In two dimensions the advantage gained from sorting the items by decreasing height before packing takes place is even more significant. To appreciate this observation, consider a list of items where each item in the list is taller than the item that precedes it in the list and the sum of the widths of the items is equal to or less than the width of the strip. Then the strip height resulting from an unsorted packing equals the sum of the items' heights, while sorting the items results in a strip height equal to the height of the tallest item in the list. Therefore, most offline algorithms sort the unpacked items before an attempt is made to pack them.

3.2 Known Level-Packing Heuristics

In this section known algorithms for strip packing are presented in some detail. A brief introduction to each algorithm is followed by a pseudocode listing of the procedure together with a worked example.

3.2.1 The Next-Fit Decreasing Height Algorithm

One of the earliest two-dimensional strip packing algorithms is the *next-fit decreasing height* (NFDH) algorithm published by Coffman *et al.* [32] in 1980. It is based on the *next-fit* (NF) algorithm for one-dimensional bin packing by Johnson [84]. The algorithm begins by packing an item into the bottom, left-hand corner of a strip. An attempt is made to pack subsequent items into the same level, as far to the left as possible with the lower edge of the item at the

[1] Although not strictly correct, sorting items by non-increasing height (width) is referred to as sorting them by decreasing height (width) from this point on, as is commonly done in the literature.

same height as the lower edge of all other items in that level. If the item does not fit into the current level, the level is closed and a new one is opened above it. Once a level is closed, no further items may be packed into it. The height of the new level equals the height of the top edge of the tallest (or leftmost) item on the level below. A pseudocode representation of the NFDH algorithm is shown in Algorithm 3.1.

Algorithm 3.1 Next-fit decreasing height algorithm (NFDH)

Input: A list $\mathcal{I}$ of items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items and the strip width `W`.

Output: A packing of the items in $\mathcal{I}$ into a strip of width `W`.

1: sort the list of items $\mathcal{I}$ by decreasing height
2: `level` $\leftarrow 1$, $i \leftarrow 1$, pack item $\mathcal{I}_i$ into `level`
3: $w(\texttt{level}) \leftarrow w(\mathcal{I}_i)$, $h(\texttt{level}) \leftarrow h(\mathcal{I}_i)$
4: **for** $i \leftarrow 2$ **to** $|\mathcal{I}|$ **do**
5: `level` $\leftarrow 1$, `Found` $\leftarrow$ False
6: **if** $w(\mathcal{I}_i) + w(\texttt{level}) \leq \texttt{W}$ **then**
7: pack $\mathcal{I}_i$ on `level`, adjacent to $\mathcal{I}_{i-1}$
8: $w(\texttt{level}) \leftarrow w(\texttt{level}) + w(\mathcal{I}_i)$
9: **else**
10: `level` $\leftarrow$ `level` $+ 1$, pack $\mathcal{I}_i$ on `level`
11: $h(\texttt{level}) \leftarrow h(\mathcal{I}_i)$, $w(\texttt{level}) \leftarrow w(\mathcal{I}_i)$
12: **end if**
13: **end for**

Worked Example

When sorting the items in Table 3.1 by means of the merge-sort algorithm according to decreasing height, the list $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_5, \mathcal{I}_{11}, \mathcal{I}_2, \mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_{13}, \mathcal{I}_8, \mathcal{I}_7, \mathcal{I}_9, \mathcal{I}_6, \mathcal{I}_{12}, \mathcal{I}_4\}$ results. Item $\mathcal{I}_1$ is the tallest item and initialises the first level. Item $\mathcal{I}_5$ is next in the list and fits adjacent to $\mathcal{I}_1$; hence it is packed into the first level. The next item in the list, $\mathcal{I}_{11}$, is too wide to fit into the first level and initialises a second level. The fourth item in the list of items is $\mathcal{I}_2$ and it fits adjacent to $\mathcal{I}_{11}$ in the second level. Insufficient space remains in the second level for item $\mathcal{I}_{10}$ and it initialises a third level. Item $\mathcal{I}_3$ follows $\mathcal{I}_{10}$ in the list and fits adjacent to $\mathcal{I}_{10}$ in the same level. The space between $\mathcal{I}_3$ and the right-hand boundary of the strip is insufficient for $\mathcal{I}_{13}$ to be packed there. Therefore it is the first item to be packed into a fourth level. The item that follows it in the sorted list, item $\mathcal{I}_8$, fits into the remaining space in the level and is packed adjacent to $\mathcal{I}_{13}$. Item $\mathcal{I}_7$ initialises a fifth level and $\mathcal{I}_9$ and $\mathcal{I}_6$ are packed into the same level. The final level has sufficient space for items $\mathcal{I}_{12}$ and $\mathcal{I}_4$. The resulting strip height is 37 and a graphical representation of the packing may be found in Figure 3.3(a).

Known Performance Bounds

Coffman *et al.* [32] established the absolute performance bound

$$\texttt{NFDH}(\mathcal{L}) \leq 3\,\texttt{OPT}(\mathcal{L}),$$

while an asymptotic performance bound is

$$\texttt{NFDH}(\mathcal{L}) \leq 2\,\texttt{OPT}(\mathcal{L}) + 1.$$

Gu *et al.* [65] proved the expected packing height $E[\texttt{NFDH}(\mathcal{L})]$ resulting from an NFDH packing of items in a list $\mathcal{L}$ is

$$E[\texttt{NFDH}(\mathcal{L})] \approx \frac{n}{3},$$

where the heights and widths of the items are independent and identically distributed in the interval $(0, 1)$.

Worst-case Time Complexity

The *merge-sort* algorithm [86], which has a worst-case time complexity of $\mathcal{O}(n \log n)$, may be utilised to sort the n items. The content of the *for*-loop spanning lines 4–13 is executed n times. There are no further loops and each step within the *for*-loop has a constant time complexity. Therefore the part of the algorithm after the sorting step has a time complexity of $\mathcal{O}(n)$. The worst-case time complexity of the sorting procedure dominates the time complexity of the *for*-loop; hence the NFDH algorithm has a worst-case time complexity of $\mathcal{O}(n \log n)$.

Algorithmic Variations

The algorithm may be altered by initially sorting the items in another manner. It would not be useful to sort the items in increasing order as each item would then initialise a new level, unless it has the same height as the previous item in the list. However, by sorting the items by decreasing height and then resolving any ties by additionally sorting these items by decreasing or increasing width, it is possible that other solutions may be found for the same problem instance [125]. The *next-fit decreasing height decreasing width* (NFDHDW) algorithm is the next-fit algorithm applied to an item list that has been sorted in this manner. The *next-fit decreasing height increasing width* (NFDHIW) algorithm is similar, except that it initially sorts the items by decreasing height and increasing width. Both these variations are evaluated alongside the NFDH algorithm in a subsequent chapter in this dissertation.

3.2.2 The First-Fit Decreasing Height Algorithm

The *first-fit decreasing height* (FFDH) algorithm was also developed by Coffman *et al.* [32] in 1980. It is a two-dimensional adaptation of the *first-fit decreasing* (FFD) algorithm for one-dimensional bin packing by Johnson *et al.* [85]. In this algorithm all items are sorted by decreasing height prior to packing. A level is initialised by the tallest unpacked item in the list. The level height equals the height of the item. Items are iteratively packed into the lowest level into which they fit. If an item does not fit into any existing level, a new level is initialised. A pseudocode listing of the FFDH algorithm may be found in Algorithm 3.2.

Worked Example

Sorting the example instance in Table 3.1 according to decreasing height, the list $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_5, \mathcal{I}_{11}, \mathcal{I}_2, \mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_{13}, \mathcal{I}_8, \mathcal{I}_7, \mathcal{I}_9, \mathcal{I}_6, \mathcal{I}_{12}, \mathcal{I}_4\}$ results. Item $\mathcal{I}_1$ initialises the first level by being packed into the lower left-hand corner of the strip. Item $\mathcal{I}_5$ is packed adjacent to $\mathcal{I}_1$. Item $\mathcal{I}_{11}$ is too wide to be packed into the first level and initialises a second level. However, item $\mathcal{I}_2$ does fit into the first level and is packed adjacent to $\mathcal{I}_5$. The remaining space in the first two levels is less than the space $\mathcal{I}_{10}$ requires; hence $\mathcal{I}_{10}$ initialises a third level. Insufficient space remains

Algorithm 3.2 First-fit decreasing height algorithm (FFDH)

Input: A list $\mathcal{I}$ of items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items and the strip width `W`.

Output: A packing of the items in $\mathcal{I}$ into a strip of width `W`.

```
1: sort the list of items I by decreasing height
2: level ← 1, i ← 1, NumLevels ← 1, pack item I_i into level
3: w(level) ← w(I_i), h(level) ← h(I_i)
4: for i ← 2 to |I| do
5:   level ← 1, Found ← False
6:   while level ≤ NumLevels and not Found do
7:     if w(I_i) + w(level) ≤ W then
8:       pack I_i on level
9:       w(level) ← w(level) + w(I_i), Found ← True
10:    else
11:      level ← level + 1
12:      if level > NumLevels then
13:        pack I_i on level, NumLevels ← NumLevels + 1, Found ← True
14:        h(level) ← h(I_i), w(level) ← w(I_i)
15:      end if
16:    end if
17:  end while
18: end for
```

for $\mathcal{I}_3$ to be packed into the first two levels and it is packed into the third level, adjacent to $\mathcal{I}_{10}$. The spaces on all existing levels are too narrow for $\mathcal{I}_{13}$ and it initialises a fourth level. A fifth level is initialised by $\mathcal{I}_7$ and items $\mathcal{I}_9$ and $\mathcal{I}_6$ are packed adjacent to it due to insufficient space remaining for the items in the lower levels. Finally, items $\mathcal{I}_{12}$ and $\mathcal{I}_4$ are packed into a sixth level, thereby completing the strip packing. The resulting strip height is 37 and a graphical representation of the packing may be found in Figure 3.3(b).

Known Performance Bounds

Coffman *et al.* [32] established the asymptotic performance bound

$$\texttt{FFDH}(\mathcal{L}) \leq \frac{17}{10}\,\texttt{OPT}(\mathcal{L}) + 1,$$

for the FFDH algorithm by expanding on the proofs for the one-dimensional FFD algorithm by Garey *et al.* [56] and Johnson *et al.* [85]. Furthermore, Coffman *et al.* [32] showed that if no rectangle has width exceeding $1/m$, an asymptotic performance bound is

$$\texttt{FFDH}(\mathcal{L}) \leq \left(1 + \frac{1}{m}\right)\texttt{OPT}(\mathcal{L}) + 1,$$

for some $m \geq 2$. If the list $\mathcal{L}$ comprises only squares, an asymptotic performance bound is

$$\texttt{FFDH}(\mathcal{L}) \leq \frac{3}{2}\,\texttt{OPT}(\mathcal{L}) + 1.$$

Coffman *et al.* [32] also established the absolute bound $\texttt{FFDH}(\mathcal{L}) \leq 2.7\,\texttt{OPT}(\mathcal{L})$ for the FFDH algorithm, while for items that are no wider than $1/m$ (for $m \geq 2$), an absolute performance bound is $\texttt{FFDH}(\mathcal{L}) \leq \left(2 + \frac{1}{m}\right)\texttt{OPT}(\mathcal{L})$.

Worst-case Time Complexity

The merge-sort algorithm, which has worst-case time complexity of $\mathcal{O}(n \log n)$, may again be utilised to sort the items to be packed. One iteration of the *for*-loop spanning lines 4–18 has time complexity $\mathcal{O}(n)$. The identification of a suitable level is performed by the *while*-loop spanning lines 6–17. This loop attempts to fit an item into the lowest level; it therefore has complexity $\mathcal{O}(n)$ in the worst case[2]. This forms part of the *for*-loop. The entire *for*-loop therefore has a worst-case time complexity of $\mathcal{O}(n^2)$. However, the FFDH algorithm may be implemented to require $\mathcal{O}(n \log n)$ time by using appropriate data structures [84, 102, 106].

Algorithmic Variations

In the same manner that other methods of sorting items are evaluated for the NFDH algorithm, the *first-fit decreasing height decreasing width* (FFDHDW) algorithm (the best-fit algorithm where the items have been sorted by decreasing width and decreasing height) and the *first-fit decreasing height increasing width* (FFDHIW) algorithm (the first-fit algorithm with items sorted in a decreasing height, increasing width manner) are evaluated alongside the FFDH algorithm later in this dissertation.

3.2.3 The Best-Fit Decreasing Height Algorithm

The *best-fit decreasing height* (BFDH) algorithm was first named and studied in detail by Coffman and Shor [34] in 1990, but Berkey and Wang [16, p. 425] had briefly described the algorithm in a 1987 paper on the two-dimensional single bin size bin packing problem. It is a two-dimensional adaptation of the *best-fit decreasing* (BFD) algorithm for one-dimensional bin packing by Johnson *et al.* [85]. Here items are packed (if they fit) into the level with minimum residual horizontal space — the unpacked space remaining width-wise in a level if the item were to be packed there. This algorithm is similar to the FFDH algorithm in terms of allowing previous levels to be revisited (the NFDH algorithm did not). If the item cannot be packed into any existing levels, a new level is initialised. A pseudocode listing of the BFDH algorithm may be found in Algorithm 3.3.

Worked Example

By sorting the items in Table 3.1 according to decreasing height by means of the merge-sort algorithm, the list $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_5, \mathcal{I}_{11}, \mathcal{I}_2, \mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_{13}, \mathcal{I}_8, \mathcal{I}_7, \mathcal{I}_9, \mathcal{I}_6, \mathcal{I}_{12}, \mathcal{I}_4\}$ results. Item $\mathcal{I}_1$ is the tallest item and initialises the first level. Item $\mathcal{I}_5$ is next in the list and fits adjacent to $\mathcal{I}_1$. Therefore it is packed into the first level. Item $\mathcal{I}_{11}$ is too wide to fit into the first level and initialises a second level. Item $\mathcal{I}_2$ fits into the first two levels. However, by packing $\mathcal{I}_2$ into the first level, less space remains between it and the right-hand boundary of the strip than if it were packed into the second level. Item $\mathcal{I}_{10}$ does not fit into any of the existing levels and initialises a third level. Only the third level has sufficient space for $\mathcal{I}_3$ and it is packed adjacent to $\mathcal{I}_{10}$. Item $\mathcal{I}_{13}$ does not fit into any existing levels and initialises a fourth level. The second, third and fourth levels have sufficient space for $\mathcal{I}_8$ and the third and fourth levels leave no residual horizontal space after having packed $\mathcal{I}_8$. Item $\mathcal{I}_8$ is packed into the lower of the two

[2]Consider the example where all items in a list $\mathcal{I}$ of size n have width $\frac{1}{2}\mathtt{W} < w(\mathcal{I}_i) < \mathtt{W}$. In this case the algorithm will evaluate every level for a possible packing location. Only one item fits into a level; n levels will therefore be searched for space.

Algorithm 3.3 Best-fit decreasing height algorithm (BFDH)

Input: A list $\mathcal{I}$ of items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items and the strip width `W`.

Output: A packing of the items in $\mathcal{I}$ into a strip of width `W`.

```
1: sort the list of items I by decreasing height
2: level ← 1, i ← 1, NumLevels ← 1
3: w(level) ← w(I_i), h(level) ← h(I_i)
4: for i ← 2 to |I| do
5:   MinResSpace ← W, MinResLevel ← 0
6:   for level ← 1 to NumLevels do
7:     if MinResSpace > W − w(level) and w(I_i) + w(level) ≤ W then
8:       MinResSpace ← W − w(level), MinResLevel ← level
9:     end if
10:  end for
11:  if MinResLevel = 0 then
12:    NumLevels ← NumLevels + 1, pack I_i on NumLevels
13:    h(level) ← h(I_i), w(level) ← w(I_i)
14:  else
15:    pack I_i on MinResLevel, w(MinResLevel) ← w(MinResLevel) + w(I_i)
16:  end if
17: end for
```

levels, namely the third level. Only the second level has sufficient space for $\mathcal{I}_7$ and it is packed adjacent to $\mathcal{I}_{11}$. The remaining four items do not fit into any of the existing levels. Items $\mathcal{I}_9$, $\mathcal{I}_6$ and $\mathcal{I}_{12}$ are packed into a fifth level and $\mathcal{I}_4$ initialises a sixth level. The resulting strip height is 36 and a graphical representation of the packing may be found in Figure 3.3(c).

Worst-case Time Complexity

In the worst case the algorithm is required to evaluate every level for a possible packing location. This is performed in the *for*-loop spanning lines 6–10 in Algorithm 3.3. If only one item fits into each level, as many levels are evaluated as items that have been packed. This procedure is performed for each of the items by the *for*-loop represented by lines 4–17. The second loop (the contents of which have constant time complexity) is nested within the first. Therefore the worst-case time complexity for the BFDH algorithm is $\mathcal{O}(n^2)$.

Algorithmic Variations

In the same manner that other methods of sorting items are evaluated for the NFDH and FFDH algorithms, the *best-fit decreasing height decreasing width* (BFDHDW) algorithm packs items that have been sorted by decreasing height, resolving ties by sorting according to decreasing width in a best fit manner. The *best-fit decreasing height increasing width* (BFDHIW) algorithm packs items in a best-fit manner after they have been sorted by decreasing height, resolving ties by sorting according to increasing width. They are both evaluated alongside the BFDH algorithm later in this dissertation.

3.2.4 The Knapsack Problem Algorithm

Lodi *et al.* [105, p. 7] described the *knapsack problem* (KP) group of algorithms in 1999. This algorithm sorts the items by decreasing height. The tallest unpacked item initiates a level and a knapsack problem is then solved in order to determine a set of unpacked items with the greatest combined area for the remaining space width-wise in the level. The profit of the item is its area, the cost of the item is its width and the knapsack's capacity is the initialising item's width subtracted from the strip width. A new level is initialised by the tallest unpacked item. This process is repeated until all items have been packed. The 0-1 knapsack problem may be formulated as

$$\begin{aligned} \text{maximise} \quad & \sum_{i=1}^{u} h\left(\mathcal{U}_i\right) w\left(\mathcal{U}_i\right) x_i, \\ \text{subject to} \quad & \sum_{i=1}^{u} w\left(\mathcal{U}_i\right) x_i \leq \mathtt{W} - w\left(\mathcal{I}_f\right), \\ & x_i \in \{0,1\}, \quad i = 1, \ldots, u, \end{aligned}$$

where u is the number of unpacked items remaining, $\mathcal{U}$ is the list of unpacked items and $\mathcal{I}_f$ is the item that initialised the level. A pseudocode listing of the KP algorithm may be found in Algorithm 3.4.

Algorithm 3.4 Knapsack problem algorithm (KP)

Input: A list $\mathcal{I}$ of items to be packed, the dimensions $\langle w\left(\mathcal{I}_i\right), h\left(\mathcal{I}_i\right)\rangle$ of the items and the strip width $\mathtt{W}$.

Output: A packing of the items in $\mathcal{I}$ into a strip of width $\mathtt{W}$.

1: sort the list of items $\mathcal{I}$ by decreasing height
2: $\mathtt{level} \leftarrow 0$, $\mathcal{U} \leftarrow \mathcal{I}$, $\mathcal{P} \leftarrow \emptyset$
3: **while** $\mathcal{U} \neq \emptyset$ **do**
4: re-index the items such that $\mathcal{U}_1$ is the first unpacked item and $\mathcal{U}_u$ is the last
5: $\mathtt{level} \leftarrow \mathtt{level} + 1$, $h\left(\mathtt{level}\right) \leftarrow h\left(\mathcal{U}_1\right)$
6: **for** $i = 2$ **to** u **do**
7: ObjectiveFunction$(i) \leftarrow h\left(\mathcal{U}_i\right) w\left(\mathcal{U}_i\right)$
8: Constraint$(i) \leftarrow w\left(\mathcal{U}_i\right)$
9: set $x\left(i\right)$ binary
10: **end for**
11: ConstraintRHS $\leftarrow \mathtt{W} - w\left(\mathcal{U}_1\right)$
12: solve the knapsack problem for $\underline{x}$
13: $\mathcal{P} \leftarrow \mathcal{P} \bigcup \{\mathcal{U}_1\}$, $\mathcal{U} \leftarrow \mathcal{U} \setminus \{\mathcal{U}_1\}$
14: **for** $i = 2$ **to** u **do**
15: **if** $x\left(i\right) = 1$ **then**
16: pack $\mathcal{U}_i$ on $\mathtt{level}$, $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{U}_i\}$, $\mathcal{U} \leftarrow \mathcal{U} \setminus \{\mathcal{U}_i\}$
17: **end if**
18: **end for**
19: **end while**

Worked Example

By sorting the items in Table 3.1 according to decreasing height by means of the merge-sort algorithm, the list $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_5, \mathcal{I}_{11}, \mathcal{I}_2, \mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_{13}, \mathcal{I}_8, \mathcal{I}_7, \mathcal{I}_9, \mathcal{I}_6, \mathcal{I}_{12}, \mathcal{I}_4\}$ results. Item $\mathcal{I}_1$ is

the tallest item and initialises the first level. The following 0-1 knapsack problem is solved:

$$\begin{aligned} \text{maximise} \quad & 14x_2 + 42x_3 + 7x_4 + 99x_5 + 14x_6 + 15x_7 + 15x_8 + 18x_9 + 63x_{10} + \\ & 126x_{11} + 12x_{12} + 80x_{13}, \\ \text{subject to} \quad & 2x_2 + 7x_3 + 7x_4 + 11x_5 + 7x_6 + 5x_7 + 4x_8 + 6x_9 + 9x_{10} + \\ & 14x_{11} + 6x_{12} + 16x_{13} \leq 16, \\ & x_i \in \{0, 1\}, \quad i = 2, \ldots, 16, \end{aligned}$$

with the result that $x_2 = x_{11} = 1$ and all other x values are zero. Therefore, items $\mathcal{I}_2$ and $\mathcal{I}_{11}$ are packed into the first level. A second level is initialised by item $\mathcal{I}_5$, the tallest of the remaining items. An integer program is formulated for this level with the remaining items and $x_{10} = 1$ results, with all other x values equal to zero. Therefore $\mathcal{I}_{10}$ is packed adjacent to $\mathcal{I}_5$. The tallest unpacked item is $\mathcal{I}_3$ and it initialises a third level. The solution to the corresponding knapsack problem results in $x_8 = x_9 = 1$ and the other x values are zero. A fourth level is initialised by $\mathcal{I}_{13}$ and the knapsack problems return a value of zero for all x values, because no unpacked items fit into the space between $\mathcal{I}_{13}$ and the right-hand boundary of the strip. Item $\mathcal{I}_7$ is the tallest unpacked item and initialises the fifth level. The corresponding knapsack problem yields $x_6 = x_{12} = 1$ and $x_4 = 0$. Item $\mathcal{I}_4$ is the last unpacked item and initialises the final level. The resulting strip height is 35 and a graphical representation of the packing may be found in Figure 3.3(d).

Worst-case Time Complexity

The sorting step in line 1 of Algorithm 3.4 may be performed by means of the merge-sort algorithm, which has $\mathcal{O}(n \log n)$ time complexity. The *while*-loop that spans lines 3–19 may execute its contents n times in the worst case (such as when all items have a width greater than half the width of the strip). The step listed in line 4 is not required in practice. However, it is present to simplify the pseudocode in the remainder of the algorithm. The *for*-loop spanning lines 6–10 has time complexity $\mathcal{O}(n)$ as its contents may be executed up to $n - 1$ times. All lines listed within the *for*-loop have constant time complexity. Setting the right-hand side of the constraint has constant time complexity, but the solving procedure on line 12 has an exponential worst-case time complexity of $\mathcal{O}(2^n)$. This is due to the linear programming solver *lp_solve* [15] making use of the simplex algorithm to solve the integer program. This time complexity overrides the $\mathcal{O}(n)$ time complexity of lines 13–18. Combined with the *while*-loop, the part of the algorithm spanning lines 3–19 has a time complexity of $\mathcal{O}(n2^n)$, which is the overall worst-case time complexity of the KP algorithm.

Practical Considerations

This is the first algorithm listed in this dissertation for which the items are not packed in the same order in which they appear in the list. There are three methods in which items may be removed from the list. The first method makes a copy of the list of items. If an item is packed, the item is removed from the list by shortening the array that stores the list of items. This has a time complexity of $\mathcal{O}(n)$ because if an item i is removed all $n - i$ items after i are copied to the position of the item that appears before it in the original list.

The second approach is to assign a boolean value to each item that is *true* when an item is packed and *false* when an item is unpacked. Unfortunately this may lead to repeated evaluations of the same items' packed status during searches for unpacked items.

Linked lists may be used to overcome the weaknesses of the two other approaches mentioned above. The data structure that represents an item is a decuple (10-tuple) that contains, amongst other properties, the items that appear before and after an item in a list. For example, consider the ordered list $\{\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3\}$. This may be represented as shown in Figure 3.2.

$\mathcal{I}(1).\texttt{prv} = -1$
$\mathcal{I}(1).\texttt{nxt} = 2$
$\mathcal{I}(2).\texttt{prv} = 1$
$\mathcal{I}(2).\texttt{nxt} = 3$
$\mathcal{I}(3).\texttt{prv} = 2$
$\mathcal{I}(3).\texttt{nxt} = -1$

Figure 3.2: *The use of* linked lists *to represent an ordered list of items.*

The properties .prv and .nxt represent the indices of the previous and next item in an ordered list, respectively. The first item in the list has its previous item set to negative one, to avoid any false loops that may occur (for example if $\mathcal{I}(1).\texttt{prv} = 3$ in the example above). The same applies for the property of the last item in the list which represents the next item. If an item is packed that has both its .nxt and .prv properties equal to negative one, then there are no further items to be packed and the algorithm may terminate. One or two additional integer variables are required to save the index values of the first and last unpacked items in the list. This allows the algorithms that employ the linked lists to begin a packing procedure from the first unpacked item and not from the first item in the list. The removal of an item is a simple procedure with constant time complexity; the .prv property of the next item (if it exists) is set equal to the .prv property of the item being removed and the .nxt property of the previous item (if it exists) is set equal to the .nxt property of the removed item. The procedure that links the items has a time complexity of $\mathcal{O}(n)$ and may be performed directly after the sorting procedure. The advantage is that the need to identify an item's packed status disappears, thereby saving time during the packing procedure, especially as the number of unpacked items diminish.

In order to restrict the time required by the algorithm to find a solution for large problems, some further restrictions may take place. The solver employed to solve linear programs throughout this dissertation, *lp_solve* [15], allows the user to specify a timeout period of an integer number of seconds. Thus, it is possible to restrict the time allowed to solve the knapsack problem for a level to one second. If an alternative solution has been found using, for example, an FFDH strategy to fill the level with unpacked items, it may replace the solution found by the solver if the solution is suboptimal and worse than the heuristic solution for the level. If the solution is suboptimal after the timeout period, but better than the alternative solution, then the suboptimal solution may be used to pack items into the level. An additional algorithm, called the *time-restricted knapsack problem* (KP_{TR}) algorithm, which makes use of the timeout function will also be considered for comparison purposes.

3.2.5 The JOIN Algorithm

In order to solve initial solutions for their exact approaches to the strip packing problem, Martello *et al.* [110] designed an algorithm called *JOIN*. The items are sorted by decreasing height, then the list is scanned for pairs of consecutive items $\mathcal{L}_i$ and $\mathcal{L}_{i+1}$ whose height difference is no larger than a proportion δ (typically in the range $[0, 10]$) and the combined width of the two is no larger than the strip width. If such an item pair exists, the pair is replaced by a

"super-item" of height $h(\mathcal{S}_i) = h(\mathcal{L}_i)$ and width $w(\mathcal{U}_i) = w(\mathcal{L}_i) + w(\mathcal{L}_{i+1})$. The scan then continues from item $\mathcal{L}_{i+2}$. At the end of the scan the new list of items is packed with one of the NFDH, FFDH or BFDH algorithms. The algorithm may also sort the items according to decreasing width initially, look for a pair whose difference in widths is no larger than the proportion δ, then replace them with a "super-item" of height $h(\mathcal{U}_i) = h(\mathcal{L}_i) + h(\mathcal{L}_{i+1})$ and width $w(\mathcal{U}_i) = w(\mathcal{L}_i)$.

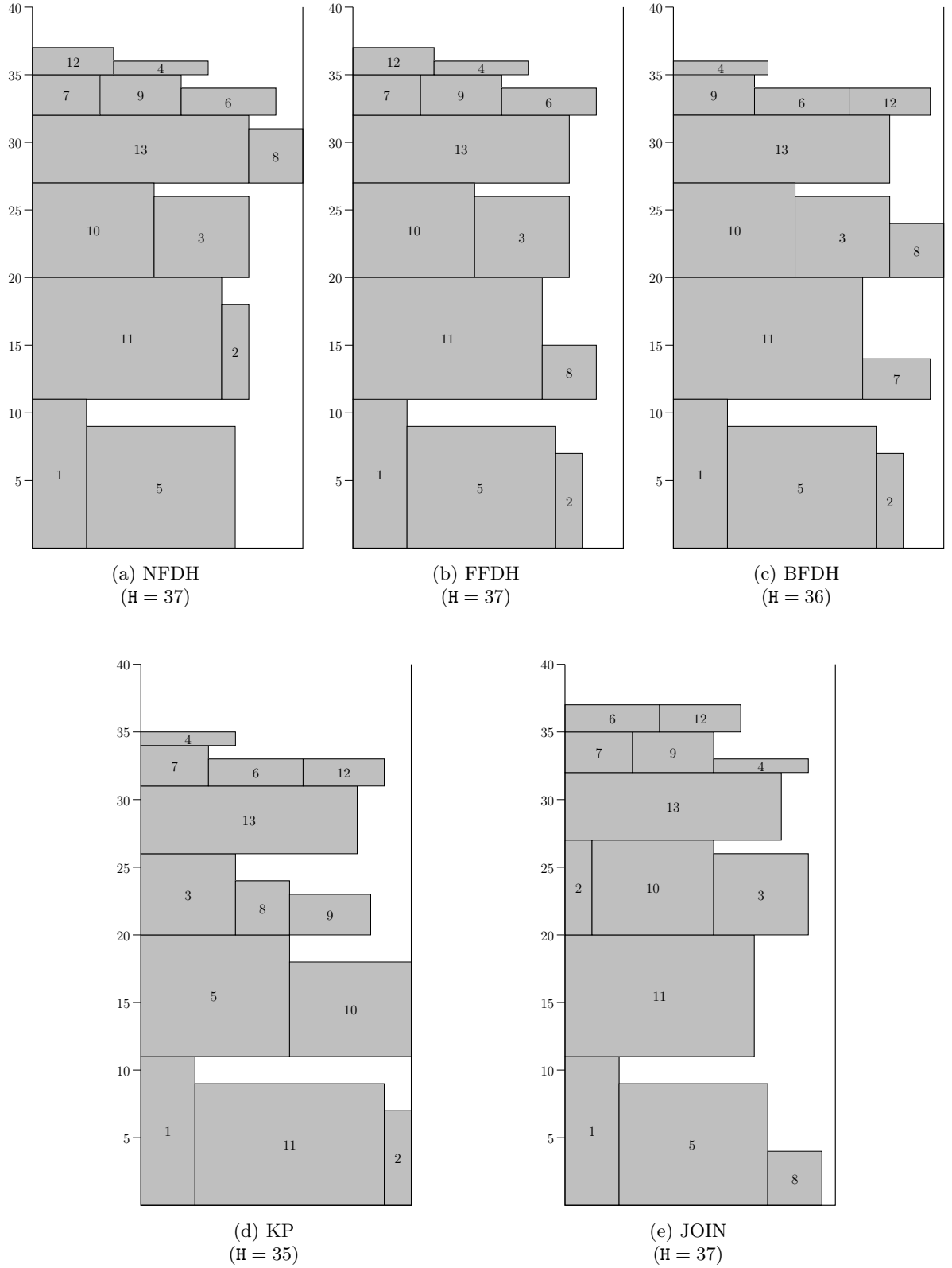
Algorithm 3.5 Algorithm JOIN

Input: A list $\mathcal{I}$ of items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items, the strip width `W`, the orientation of the joining (horizontal or vertical) and the proportion γ by which the items may differ in height or width for them to be joined.
Output: A packing of the items in $\mathcal{I}$ into a strip of width `W`.

sort the list of items $\mathcal{I}$ by decreasing height (if joining items horizontally)
or sort the items by decreasing width (if joining items vertically)
define $\mathcal{S} = \emptyset$ as the list of super-items, $i \leftarrow 1$, $j \leftarrow 1$
while $i \leq |\mathcal{I}|$ **do**
  if $i < |\mathcal{I}|$ **then**
    if horizontal **and** $\dfrac{h(\mathcal{I}_i) - h(\mathcal{I}_{i+1})}{h(\mathcal{I}_i)} \times 100 \leq \delta$ **and** $w(\mathcal{I}_i) + w(\mathcal{I}_{i+1}) \leq$ `W` **then**
      $w(\mathcal{S}_j) \leftarrow w(\mathcal{I}_i) + w(\mathcal{I}_{i+1})$, $h(\mathcal{S}_j) \leftarrow h(\mathcal{I}_i)$
      $i \leftarrow i + 2$
    else if vertical **and** $\dfrac{w(\mathcal{I}_i) - w(\mathcal{I}_{i+1})}{w(\mathcal{I}_i)} \times 100 \leq \delta$ **then**
      $h(\mathcal{S}_j) \leftarrow h(\mathcal{I}_i) + h(\mathcal{I}_{i+1})$, $w(\mathcal{S}_j) \leftarrow w(\mathcal{I}_i)$
      $i \leftarrow i + 2$
    else
      $\mathcal{S}_j \leftarrow \mathcal{I}_i$
      $i \leftarrow i + 1$
    end if
  else if $i = |\mathcal{I}|$ **then**
    $\mathcal{S}_j \leftarrow \mathcal{I}_i$
    $i \leftarrow i + 1$
  end if
  $j \leftarrow j + 1$
end while
use NFDH, FFDH or BFDH to pack $\mathcal{S}$
decode super-items back to original items

Worked Example

Set $\delta = 0$. By sorting the items in Table 3.1 according to decreasing height using the merge-sort algorithm, the list $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_5, \mathcal{I}_{11}, \mathcal{I}_2, \mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_{13}, \mathcal{I}_8, \mathcal{I}_7, \mathcal{I}_9, \mathcal{I}_6, \mathcal{I}_{12}, \mathcal{I}_4\}$ results. No items have the same height as $\mathcal{I}_1$ ($\mathcal{U}_1$), but items $\mathcal{I}_5$ and $\mathcal{I}_{11}$ have the same height. However, the sum of their widths is greater than the strip width and they are not joined. They are relabelled items $\mathcal{U}_2$ and $\mathcal{U}_3$, respectively. Items $\mathcal{I}_2$ and $\mathcal{I}_{10}$ have the same height, their combined width is less than the width of the strip and they are joined to form item $\mathcal{U}_4$. No items have the same heights as $\mathcal{I}_3$, $\mathcal{I}_{13}$ and $\mathcal{I}_8$; hence these are renamed $\mathcal{U}_5$, $\mathcal{U}_6$ and $\mathcal{U}_7$, respectively. Items $\mathcal{I}_7$ and $\mathcal{I}_9$ have the same height and are joined to form $\mathcal{S}_8$. Items $\mathcal{I}_6$ and $\mathcal{I}_{12}$ satisfy the joining conditions



(a) NFDH ($\mathtt{H} = 37$)

(b) FFDH ($\mathtt{H} = 37$)

(c) BFDH ($\mathtt{H} = 36$)

(d) KP ($\mathtt{H} = 35$)

(e) JOIN ($\mathtt{H} = 37$)

Figure 3.3: *Results obtained when packing items in $\mathcal{I}$ using the known level packing algorithms described in §3.2 for the strip packing problem. The resulting packing heights* H *are also shown.*

and are merged into one item to form $\mathcal{U}_9$. Item $\mathcal{I}_4$ is relabelled $\mathcal{U}_{10}$. The number of items has been reduced from thirteen to ten and may be packed by means of the NFDH, FFDH or BFDH algorithms. The resulting strip height is 37 if the FFDH algorithm is used and a graphical representation of the packing may be found in Figure 3.3(e).

Worst-case Time Complexity

The merge-sort algorithm, which has a worst-case time complexity of $\mathcal{O}(n \log n)$, may be utilised to sort the items in $\mathcal{L}$ either by decreasing height (for the horizontal joining of items), or decreasing width (for the vertical joining of items). Line 3 has constant time complexity. The *while*-loop spanning lines 4–21 will execute its contents n times in the worst case to create the super-items. The contents of this loop have constant time complexity, resulting in a worst-case time complexity $\mathcal{O}(n)$ for the loop. After the creation of the super-items, they may be packed by means of either the NFDH, FFDH or BFDH algorithms. The list $\mathcal{S}$ need not be sorted in the case of joining items horizontally due to the sorting in line 1. Therefore, the sorting steps in the packing algorithms may be avoided resulting in a worst-case complexity of $\mathcal{O}(n)$ for the NFDH algorithm. However, the sorting step must remain if the items are joined vertically, resulting in a worst-case time complexity of $\mathcal{O}(n \log n)$ for the NFDH algorithm. The time complexity of the FFDH and BFDH algorithms remain $\mathcal{O}\left(n^2\right)$. The decoding of super-items back to the original items on line 23 has time complexity $\mathcal{O}(n)$. If the NFDH algorithm is used to pack the items, algorithm JOIN has a worst-case time complexity of $\mathcal{O}(n \log n)$ due to the addition of three (or two) $\mathcal{O}(n)$ steps with one (or two) $\mathcal{O}(n \log n)$ step when joining items horizontally (or vertically). The time complexity of the FFDH or BFDH algorithms is greater than the worst-case time complexity of any of the other steps in algorithm JOIN. Thus, the worst-case time complexity for algorithm JOIN when using algorithms FFDH or BFDH is $\mathcal{O}\left(n^2\right)$.

Algorithmic Variations

In the same manner that other methods of sorting items are evaluated for the NFDH, FFDH and BFDH algorithms, the JOIN(DHDW) algorithm packs items that have been sorted by decreasing height and decreasing width, the JOIN(DHIW) algorithm packs items after they have been sorted by decreasing height and increasing width. The JOIN algorithm that joins items vertically may also have the items sorted in three ways; decreasing width — JOIN(DW), decreasing width, resolving ties by additionally sorting those items by decreasing height — JOIN(DWDH) or decreasing width, resolving ties by additionally sorting those items by increasing height — JOIN(DWIH). They are all evaluated in following chapters.

3.3 New Level-Packing Heuristics

In this section two new algorithms for strip packing are introduced in some detail. A brief introduction to each algorithm is followed by a pseudocode listing of the procedure together with a worked example.

3.3.1 The Worst-Fit Decreasing Height Algorithm

The origins of the *worst-fit decreasing height* (WFDH) algorithm may be found in the *worst-fit decreasing* (WFD) algorithm for one-dimensional bin packing by Johnson [84]. In the same

manner that Coffman *et al.* [32], and Berkey and Wang [16] and Coffman and Shor [34] adapted the FFD and BFD algorithms to two-dimensions, respectively, the WFD algorithm may be adapted to two dimensions. It is very similar to the BFDH algorithm, the only difference being that the level with the *maximum* residual horizontal space is used for packing (if the item fits), whereas the BFDH algorithm uses the *minimum* residual horizontal space to determine the level best suited for packing. A pseudocode listing of the WFDH algorithm may be found in Algorithm 3.6.

Algorithm 3.6 Worst-fit decreasing height algorithm (WFDH)

Input: A list $\mathcal{I}$ of items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items and the strip width `W`.

Output: A packing of the items in $\mathcal{I}$ into a strip of width `W`.

1: sort the list of items $\mathcal{I}$ by decreasing height
2: `level` $\leftarrow 1$, $i \leftarrow 1$, `NumLevels` $\leftarrow 1$
3: $w($`level`$) \leftarrow w(\mathcal{I}_i)$, $h($`level`$) \leftarrow h(\mathcal{I}_i)$
4: **for** $i \leftarrow 2$ **to** $|\mathcal{I}|$ **do**
5: `MaxResSpace` $\leftarrow 0$, `MaxResLevel` $\leftarrow 0$
6: **for** `level` $\leftarrow 1$ **to** `NumLevels` **do**
7: **if** `MaxResSpace` $< $ `W` $- w($`level`$)$ **and** $w(\mathcal{I}_i) + w($`level`$) \leq$ `W` **then**
8: `MaxResSpace` $\leftarrow$ `W` $- w($`level`$)$, `MaxResLevel` $\leftarrow$ `level`
9: **end if**
10: **end for**
11: **if** `MaxResLevel` $= 0$ **then**
12: `NumLevels` $\leftarrow$ `NumLevels` $+ 1$, pack $\mathcal{I}_i$ on `NumLevels`
13: $h($`level`$) \leftarrow h(\mathcal{I}_i)$, $w($`level`$) \leftarrow w(\mathcal{I}_i)$
14: **else**
15: pack $\mathcal{I}_i$ on `MaxResLevel`, $w($`MaxResLevel`$) \leftarrow w($`MaxResLevel`$) + w(\mathcal{I}_i)$
16: **end if**
17: **end for**

Worked Example

By sorting the items in Table 3.1 according to decreasing height by means of the merge-sort algorithm, the list $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_5, \mathcal{I}_{11}, \mathcal{I}_2, \mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_{13}, \mathcal{I}_8, \mathcal{I}_7, \mathcal{I}_9, \mathcal{I}_6, \mathcal{I}_{12}, \mathcal{I}_4\}$ results. Item $\mathcal{I}_1$ initialises the first level and $\mathcal{I}_5$ is packed adjacent to it in the same level. There is insufficient space in the first level for $\mathcal{I}_{11}$ and it initialises the second level. Item $\mathcal{I}_2$ fits into both existing levels, but the second level has the maximum residual horizontal space and $\mathcal{I}_2$ is packed into that level. The existing levels have insufficient space remaining for $\mathcal{I}_{10}$ and it initialises a third level. This is the only level with sufficient space for $\mathcal{I}_3$ and the item is packed adjacent to $\mathcal{I}_{10}$. Insufficient space remains for $\mathcal{I}_{13}$ in existing levels and it initialises a fourth level. All four existing levels have sufficient space for $\mathcal{I}_8$, but the second level has the maximum residual horizontal space and the item is packed adjacent to $\mathcal{I}_{11}$. Item $\mathcal{I}_7$ initialises the fifth level due to insufficient space remaining in the existing levels. Items $\mathcal{I}_9$ and $\mathcal{I}_6$ only fit into the fifth level and the last two remaining items, namely $\mathcal{I}_{12}$ and $\mathcal{I}_4$, are packed into a sixth level. The resulting strip height is 37 and a graphical representation of the packing may be found in Figure 3.6(a).

Worst-case Time Complexity

In order to sort the items, the merge-sort algorithm may be used and it has a time complexity of $\mathcal{O}(n \log n)$. The WFDH algorithm has the same structure as the BFDH algorithm, resulting in a worst-case time complexity of $\mathcal{O}(n^2)$ for the WFDH algorithm. The *for*-loop spanning lines 4–17 executes its contents n times. Within that *for*-loop a search is launched for a suitable level to pack an item (lines 6–10), which in the worst-case, may be executed $\mathcal{O}(n)$ times. Therefore, the algorithm has a worst-case time complexity of $\mathcal{O}(n^2)$, overriding the time complexity of the sorting part of the algorithm.

Algorithmic Variations

In the same manner that other methods of sorting items are evaluated for the NFDH, FFDH and BFDH algorithms, the *worst-fit decreasing height decreasing width* (WFDHDW) algorithm packs items that have been sorted by decreasing height, resolving ties (items of equal height) by additionally sorting those items by decreasing width, in a worst-fit manner. Similarly, the *worst-fit decreasing height increasing width* (WFDHIW) algorithm packs items in a worst-fit manner after they have been sorted by decreasing height, resolving ties by additionally sorting those items by increasing width.

3.3.2 The Best Two Fit Deceasing Height Algorithm

The *best two fit decreasing height* (B2FDH) algorithm is based on the *best two fit* (B2F) algorithm for one-dimensional bin packing by Friesen and Langston [55]. The B2F algorithm packs the first bin in an FFD manner and if the bin contains more than one item, an investigation takes place to determine whether two smaller unpacked items may replace the smallest item in the bin (the *occupant*). If such items exist, the two items whose sum is greatest (and larger than the size of the occupant) replace the occupant.

It is possible to extend this procedure to two dimensions by sorting the items by decreasing height, then packing the first level in a first-fit manner such that the entire list is searched for items to pack before an attempt is made to replace the last item in the level (the occupant) with two unpacked items. Once an attempt has been made to replace the occupant, unpacked items are packed into the next level in a similar manner. The replacement procedure is only performed if the two smaller items have a sum of widths (or areas) greater than the width (or area) of the occupant. If there are two or more pairs of items with the same sum of widths (areas) that may replace the occupant, the pair with the greatest sum of areas (widths) are chosen to replace the occupant. This process continues until all items have been packed. In order to differentiate between the algorithm that uses the items' widths in an attempt to achieve an improvement and the algorithm that uses the area of items in an attempt to improve the solution, the former is called the *best two fit (by width) decreasing height* (B2FWDH) algorithm and the latter is called the *best two fit (by area) decreasing height* (B2FADH) algorithm.

It may prove impractical to solve large problem instances using this approach if all unpacked items are searched for replacements for the occupant. Instead, the algorithm may be restricted to searching for suitable pairs within a fixed range. For example, the algorithm may be restricted to search only for pairs of items that are adjacent in an ordered list. Another option would be to restrict the search for pairs to within $k-1$ items ahead of an unpacked item under investigation. This algorithm is called the *best two fit k-decreasing height* ($\text{B2F}_k\text{DH}$) algorithm (or the

B2FW$_k$DH algorithm when the width is used as the improvement criterium, or the B2FA$_k$DH algorithm when the item area is used as the improvement criterium), where k indicates the range of items that may be considered for pairing in an ordered list. The convention is that the B2F$_n$DH algorithm is the algorithm allowing all unpacked pairs of items to be considered for occupant replacement, while the B2F$_2$DH algorithm only allows adjacent pairs of items to be evaluated for occupant replacement. A pseudocode listing of the B2FW$_k$DH algorithm may be found in Algorithm 3.7.

Algorithm 3.7 Best Two Fit Decreasing Height algorithm (B2FDH)

Input: A list $\mathcal{I}$ of items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items, the strip width `W` and the range of items permitted for substitution k.

Output: A packing of the items in $\mathcal{I}$ into a strip of width `W`.

1: sort the list of items $\mathcal{I}$ by decreasing height and decreasing width
2: $\mathcal{P} \leftarrow \emptyset$, `level` $\leftarrow 0$
3: **while** $\mathcal{I} \neq \emptyset$ **do**
4: `level` $\leftarrow$ `level` $+ 1$, $w($`level`$) \leftarrow$ `W`
5: **call** PACKFLOOR $(\mathcal{I}, \mathcal{P},$ `level`$)$
6: let $\mathcal{I}_\ell$ be the last item packed into the level
7: **if** two items could exist to replace the last item packed **then**
8: `AvSpace` $\leftarrow w($`level`$) + \mathcal{I}_\ell$, `B1` $\leftarrow 0$, `B2` $\leftarrow 0$
9: **call** FINDSUITABLE $(\mathcal{I}, k,$ `AvSpace`, `B1`, `B2`$)$
10: **if** `B1` > 0 **and** `B2` > 0 **then**
11: insert $\mathcal{I}_\ell$ back into the correct place within $\mathcal{I}$
12: $\mathcal{I} \leftarrow \mathcal{I} \cup \{\mathcal{I}_\ell\}$, $\mathcal{P} \leftarrow \mathcal{P} \setminus \{\mathcal{I}_\ell\}$
13: pack items $\mathcal{I}_{\texttt{B1}}$ and $\mathcal{I}_{\texttt{B2}}$ into `level`
14: $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{I}_{\texttt{B1}}\}$, $\mathcal{I} \leftarrow \mathcal{I} \setminus \{\mathcal{I}_{\texttt{B1}}\}$
15: $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{I}_{\texttt{B2}}\}$, $\mathcal{I} \leftarrow \mathcal{I} \setminus \{\mathcal{I}_{\texttt{B2}}\}$
16: **end if**
17: **end if**
18: **end while**

Procedure 3.7.1 PACKFLOOR $(\mathcal{I}, \mathcal{P},$ `level`$)$

1: $i \leftarrow$ index of first unpacked item
2: **while** $i \leq$ index of last unpacked item **and** $\mathcal{I} \neq \emptyset$ **do**
3: **if** $w(\mathcal{I}_i) \leq w($`level`$)$ **then**
4: $w($`level`$) \leftarrow w($`level`$) - w(\mathcal{I}_i)$
5: **if** it is the first item on `level` **then**
6: $h($`level`$) \leftarrow h(\mathcal{I}_i)$
7: **end if**
8: $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{I}_i\}$, $\mathcal{I} \leftarrow \mathcal{I} \setminus \{\mathcal{I}_i\}$
9: **end if**
10: $i \leftarrow$ index of next unpacked item
11: **end while**

Worked Example

Consider the B2FW$_n$DH algorithm. By sorting the items in Table 3.1 according to decreasing height with the merge-sort algorithm, the list $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_5, \mathcal{I}_{11}, \mathcal{I}_2, \mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_{13}, \mathcal{I}_8, \mathcal{I}_7, \mathcal{I}_9, \mathcal{I}_6,$

Procedure 3.7.2 FINDSUITABLE $(\mathcal{I}, k, \texttt{AvSpace}, \texttt{B1}, \texttt{B2})$

```
1: i ← index of item after I_ℓ, BestSpace ← w(I_ℓ)
2: while i < index of the last unpacked item do
3:    j ← index of item after I_i, m ← 2
4:    while j ≤ index of the last unpacked item and m ≤ k do
5:       if w(I_i) + w(I_j) < AvSpace and BestSpace < w(I_i) + w(I_j) then
6:          B1 ← i, B2 ← j, BestSpace ← w(I_i) + w(I_j)
7:       end if
8:       j ← index of next unpacked item, m ← m + 1
9:    end while
10:   i ← index of next unpacked item
11: end while
```

$\mathcal{I}_{12}$, $\mathcal{I}_4\}$ results. Item $\mathcal{I}_1$ initialises the first level and $\mathcal{I}_5$ is packed adjacent to it in the same level. Item $\mathcal{I}_{11}$ does not fit adjacent to $\mathcal{I}_5$, but there is sufficient space to pack $\mathcal{I}_2$. No unpacked items are narrow enough to fit between $\mathcal{I}_2$ and the right-hand boundary of the level. A search for two items that yield a combined width greater than the width of $\mathcal{I}_2$ yields no pair that may replace $\mathcal{I}_2$. Therefore the first level is closed. The second level is initiated by packing $\mathcal{I}_{11}$, and $\mathcal{I}_8$ is the tallest item that fits between $\mathcal{I}_{11}$ and the right-hand boundary of the strip. No item fits between $\mathcal{I}_8$ and the right-hand boundary of the strip and hence the remaining unpacked items are searched for a pair of items that may replace $\mathcal{I}_8$. No such pair exists and the second level is closed. Item $\mathcal{I}_{10}$ is the tallest unpacked item and initialises the third level. Sufficient space remains for $\mathcal{I}_3$ and it is packed into the level. No unpacked items are narrow enough to be packed between $\mathcal{I}_3$ and the right-hand boundary of the strip. A search of the unpacked items is performed to find a pair of items with a combined width greater than the width of $\mathcal{I}_3$ that is less than or equal to the space between $\mathcal{I}_{10}$ and the right-hand boundary of the strip. Items $\mathcal{I}_7$ and $\mathcal{I}_9$ have a combined width of 11, *i.e.* 4 units greater than the width of $\mathcal{I}_3$, and are small enough to fit into the remaining space. Items $\mathcal{I}_7$ and $\mathcal{I}_{12}$ have the same combined width as items $\mathcal{I}_7$ and $\mathcal{I}_9$, but the greater combined area of the former pair results in their placement into the third level. Item $\mathcal{I}_3$ is the tallest unpacked item and initialises the fourth level. The next unpacked item in the list is $\mathcal{I}_{13}$, but it does not fit into the remaining space. However, there is sufficient space for items $\mathcal{I}_6$ and $\mathcal{I}_{12}$ and hence they are packed into the fourth level. The right-hand edge of $\mathcal{I}_{12}$ coincides with the right-hand boundary of the strip. Therefore no search is performed for a pair of items that provide a better fit. Items $\mathcal{I}_{13}$ and $\mathcal{I}_4$ remain and are packed into their own levels, because their combined width is greater than the strip width. The resulting strip height is 39, as illustrated in Figure 3.6(b).

Worst-case Time Complexity

In order to sort the items, the merge-sort algorithm may be used and has a worst-case time complexity $\mathcal{O}(n \log n)$. The contents of the *while*-loop spanning lines 3–18 are executed $\mathcal{O}(n)$ times in the worst case. All lines contained in the loop, excluding those that call Procedures 3.7.1 and 3.7.2 (lines 6 and 9), have constant time complexity. Procedure 3.7.1 evaluates all unpacked items for packing into a level in the worst case. Therefore it has time complexity $\mathcal{O}(n)$. Procedure 3.7.2 attempts to find suitable items to replace the occupant. The *while*-loop spanning lines 4–9 has time complexity $\mathcal{O}(n)$ when unrestricted ($k = n$) as its contents have constant time complexity. This loop is contained within a *while*-loop spanning lines 2–11, which also contains line 3 with constant time complexity. Subsequently, Procedure 3.7.2 has a worst-

case time complexity of $\mathcal{O}(n^2)$ if $k = n$, or $\mathcal{O}(n)$ if k is restricted. Therefore, for unrestricted k the B2FDH algorithm has a worst-case time complexity $\mathcal{O}(n^3)$, and worst-case time complexity $\mathcal{O}(n^2)$ for restricted k.

Practical Considerations

Due to the selective removal of items from anywhere in the sorted list, and the addition of items to the list, an implementation of the algorithm should preferably make use of the linked lists discussed in the *practical considerations* subsection of §3.2.4. The addition of items is similar to the deletion of items. Consider a list of two items; $\mathcal{I}_1$ and $\mathcal{I}_3$. The linked list for these two items would have form illustrated in Figure 3.4.

$\mathcal{I}(1).\texttt{prv} = -1$
$\mathcal{I}(1).\texttt{nxt} = 3$
$\mathcal{I}(3).\texttt{prv} = 1$
$\mathcal{I}(3).\texttt{nxt} = -1$

Figure 3.4: *A list of items prior to the addition of another item.*

In order to add an item to the list, the changes illustrated in Figure 3.5 would be required.

$\mathcal{I}(1).\texttt{nxt} = 2$
$\mathcal{I}(2).\texttt{prv} = 1$
$\mathcal{I}(2).\texttt{nxt} = 3$
$\mathcal{I}(3).\texttt{prv} = 2$

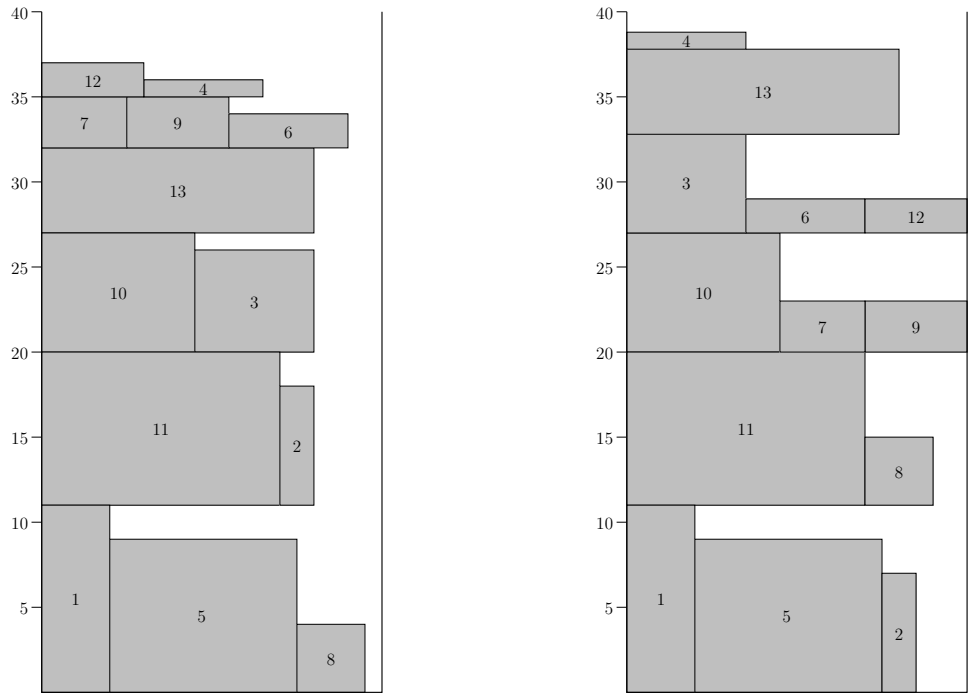
Figure 3.5: *The changes required to add an item to the list of items in Figure 3.4.*

Using these tools allows the worst-case time complexity to be reduced by a factor of n, because if sorted arrays were used instead of linked lists, a procedure of $\mathcal{O}(n)$ would be required to insert an item into the list. With the linked list, the procedure of adding items to a list of items has constant time complexity. Moreover, the additional cost of the copy procedures that occur in computer memory during the changing of an array's dimensions [139] are avoided.

In the same manner that other methods of sorting items are evaluated for the previously discussed algorithms, the *best two fit decreasing height decreasing width* (B2FDHDW) algorithm packs items that have been sorted by decreasing height, resolving ties (items with equal height) by additionally sorting items by decreasing width, in a best two fit manner. Similarly, the *best two fit decreasing height increasing width* (B2FDHIW) algorithm packs items in a best two fit manner after they have been sorted by decreasing height and equalities have been resolved by sorting them by increasing width.

3.4 Chapter Summary

Published level heuristics for the strip packing problem were reviewed in this chapter, in fulfilment of Dissertation Objective IV(a), as stated in §1.3. The NFDH algorithm by Coffman *et al.* [32] was considered first, followed by the FFDH algorithm by the same authors. Thereafter



(a) WFDH
(H = 37)

(b) B2FW$_n$DH
(H = 39)

Figure 3.6: *Results obtained when packing items in $\mathcal{I}$ into a strip of width 20 using the new level strip packing algorithms described in §3.3. The resulting packing heights* H *are also shown.*

the BFDH algorithm by Coffman and Shor [34] was reviewed, followed by the KP algorithm of Lodi *et al.* [105] and algorithm JOIN by Martello *et al.* [110].

The second part of the chapter contains two heuristics that have not appeared in the literature. The WFDH algorithm was adapted from the WFD algorithm for one-dimensional bin packing by Johnson [84]. Finally the B2FW$_k$DH and B2FA$_k$DH algorithms, two-dimensional adaptations of the B2F algorithm by Friesen and Langston [55] for one-dimensional bin packing, were introduced in fulfilment of Dissertation Objective V(a), as stated in §1.3.

CHAPTER 4

Pseudolevel Strip Packing Heuristics

Contents

Various pseudolevel-packing approaches to the two-dimensional strip packing problem are considered in this chapter. Pseudolevel algorithms are similar to level algorithms in that items are packed into levels. However, pseudolevel algorithms allow items to be packed anywhere in the plane defined by the boundaries of levels. The workings of a number of known heuristics following this approach are described and a number of new heuristics are introduced. Each of the algorithms is illustrated with the aid of a figure containing the packing pattern, a region designated as free space (for further packing) and arrows (as shown in Figure 4.1) that indicate the allowed packing directions for items.

4.1 Practical Considerations for Pseudolevel Algorithms

The packing of items onto the ceiling of a level (as some of the algorithms in this chapter do) renders the programmatic implementation of the algorithm more difficult than is the case with level algorithms. It is critical that the algorithm ensures that there is no overlap of ceiling-packed items and floor-packed items. If the item dimensions were guaranteed to be integer values, one may construct an array that represents the height of the floor-packed items (also called the *skyline*) between two consecutive horizontal integer coordinates of the strip (see Figure 4.2(a)). This was proposed by Burke *et al.* [22] as a tool for their plane packing algorithm in order to reduce the time required to find possible packing locations for unpacked items. A comparison

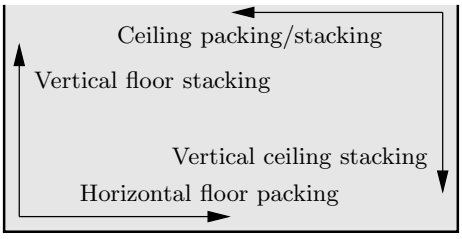
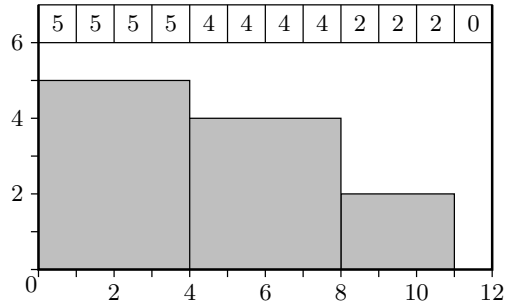
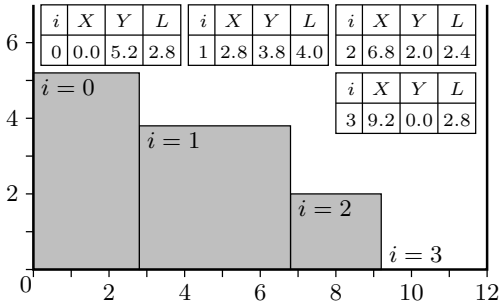


Figure 4.1: *A guide to the arrows depicting the directions in which algorithms in this chapter may pack items in the utilised space not used by the level algorithms of §3. Arrows pointing to the right indicate horizontal stacking from left to right, while arrows pointing left either indicate ceiling packing, or horizontal ceiling stacking, from right to left. Vertical arrows pointing upwards indicate that stacking may take place in an upwards fashion, while arrow pointing down indicate that stacking may take place from the ceiling down to the floor-packed items (without overlapping those items). These figures are employed to illustrate the differences in packing between pseudolevel algorithms and the level algorithms of §3. Hence the arrow representing horizontal floor packing (the manner in which items are packed by level algorithms) is omitted. Instead, a set of items is shown in the figures to represent how level algorithms would pack them.*

of the height (or vertical coordinate) of the bottom-left corner of an item under investigation for ceiling packing with the value in the array corresponding to the space to the right of the horizontal strip coordinate of the corner would yield either a possible overlap, or a valid position for the item. There are two weaknesses with this approach. The array will be very large ($\ell \times m$, where m is the integer value of the strip width) as it will have to cater for every possible position along the floor for every possible level (there may be ℓ levels). It is possible to reduce the size of the array if dynamic arrays are supported by the programming language of implementation. In that case, the array may be expanded as the number of levels increases. The second weakness is the lack of support for non-integer rectangle dimensions. This is a significant weakness as there are benchmark instances (such as by Wang and Valenzuela [156]) and real-world applications (such as by Wang [155]) of packing problems in which the item dimensions are non-integer.



(a) An integer approach to saving the height to which items have been packed.



(b) A floating-point approach to saving the height to which items have been packed.

Figure 4.2: *Two approaches to preserving the height to which floor-packed items have been packed (floor profile or skyline) without requiring a search through the entire list of items.*

In order to accurately monitor the skyline when items have floating-point lengths, Burke *et al.* [23] suggest the use of an array of triples (3-tuples). This data structure notes the horizontal coordinate of a change in height, the height (or vertical coordinate) of the skyline to the right of that point and the length of the region between subsequent changes in height (see Figure 4.2(b)). This triple may be reduced to a pair by saving only the horizontal coordinate and height, as the length may be determined by subtracting the horizontal coordinates of two adjacent points.

However, this comes at the cost of an additional element in the list representing the right-hand boundary of the strip. Some programming languages allow only one dimension of an array to be dynamic (Visual Basic .NET, the programming language used for the purposes of this dissertation, is an example of this [115]). This restriction does not allow the expansion of the array as the algorithm requires more space. Hence, it may be beneficial to allow the algorithm to alter the size of the array with each new level and use a set size for the dimension that represents the skyline in the array. This magnitude of the dimension m may be calculated using the formula

$$m = \min\left\{n, \left\lfloor \frac{\mathtt{W}}{w\left(\mathcal{I}_N\right)} \right\rfloor\right\} + 1,$$

where $\mathtt{W}$ denotes the strip width and $w\left(\mathcal{I}_N\right)$ denotes the width of item $\mathcal{I}_N$. Finding the thinnest item $\mathcal{I}_N$ has time complexity $\mathcal{O}(n)$ if n is the number of rectangles and the list is not sorted according to decreasing width. It is possible to reduce the size of the array at the expense of computational time. By sorting the items according to increasing width, one may find the value of m by adding the widths of the items in order until the sum is greater than the width of the strip. The value of m is the position in the sorted list of the item whose width resulted in the sum of widths greater than that of the strip width. This method of finding the length of one of the dimensions of the two-dimensional array of triples has a time complexity of $\mathcal{O}(n \log n)$.

4.2 Known Pseudolevel-Packing Heuristics

In this section known algorithms for strip packing are presented in some detail. A brief introduction to each algorithm is followed by a pseudocode listing of the procedure, together with a worked example, practical considerations when implementing the algorithm and an estimation of its worst-case time complexity.

4.2.1 The Floor-Ceiling Algorithms

Lodi *et al.* [103, 105] introduced the *floor-ceiling* (FC) algorithm in the late 1990s for the 2D SBSBPP. There are four variations of the algorithm, namely the oriented (O) case allowing non-guillotineable (free — F) packing ($\mathrm{FC_{OF}}$), the oriented case where a guillotineable (G) packing is required ($\mathrm{FC_{OG}}$), the case where 90° rotations (R) are allowed and a guillotineable packing is required ($\mathrm{FC_{RG}}$) and the case where rotations and free packing are allowed ($\mathrm{FC_{RF}}$). In these algorithm variations, the items are not only packed from left to right onto the floor (the horizontal line below which no item in the level may be packed), as in all the other algorithms, but also from right to left onto the ceiling of the level (a horizontal line above which no item in the level may be packed, typically at the same height as the top edge of the item that initialises the level). See Figure 4.3 for an illustration of the packing directions. The solution to the $\mathrm{FC_{OG}}$ algorithm is a four-stage cutting pattern in the worst case.

An item is *floor-feasible* for a level if it fits onto the floor of the level. The level is *ceiling-initialised* if some items are packed onto the ceiling. A ceiling is only initialised if an item is not floor-feasible. An item is *ceiling-feasible* when it can either initialise a ceiling, or be packed onto a ceiling that is already initialised. An item may only initialise the ceiling of a level if it does not fit onto the floor of any existing level. An attempt is first made to pack a new item onto an initialised ceiling (a ceiling that already has at least one item packed on it) and then onto

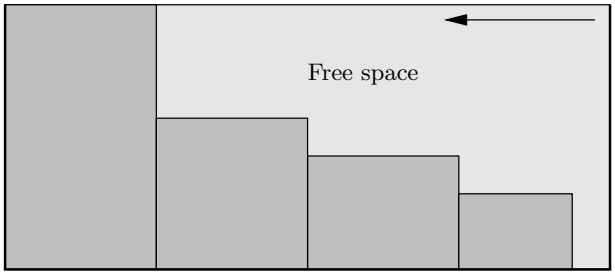


Figure 4.3: *An illustration of the free space above floor-packed items that the FC algorithms attempt to utilise. This free space is wasted by level algorithms if the items on the levels are of different heights.*

a floor if it does not fit onto any initialised ceiling, unless it fits onto an uninitialised ceiling. A new level is created if there is insufficient space for an item on the ceilings or floors of all existing levels. All floor or ceiling packing operations occur in a best-fit manner, where the item is assigned to the level in which the residual horizontal space (on the floor or ceiling, depending on the targeted area) is a minimum. The FC_{OG} algorithm ensures that the packing obeys the guillotine rule by often leaving gaps between items on the ceiling. Only oriented packing is considered in this dissertation — therefore the alterations required for the FC_{RF} and FC_{RG} algorithms are not presented in this dissertation. A pseudocode listing of the FC algorithm may be found in Algorithm 4.1.

Algorithm 4.1 Floor-ceiling algorithm (FC)

Input: A list $\mathcal{I}$ of items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items and the strip width `W`.

Output: A packing of the items in $\mathcal{I}$ into a strip of width `W`.

```
1: sort the list of items I by decreasing height
2: for i ← 1 to |I| do
3:    if I_i is ceiling-feasible for a level then
4:       pack I_i to the ceiling of the level with minimum residual horizontal space
5:       {Here FC_OG may have to leave space between items to ensure guillotine constraint
         holds.}
6:    else
7:       if I_i is floor-feasible for a level then
8:          pack I_i to the floor of the level with minimum residual horizontal space
9:       else
10:         initialise a new level and pack I_i in the bottom-left corner
11:      end if
12:   end if
13: end for
```

Sorting of the items may influence the results. Therefore three sorting schemes are considered: sorting the items by decreasing height only (FC_{OG}(DH))[1], by decreasing height and decreasing width (FC_{OG}(DHDW)), and by decreasing height and increasing width (FC_{OG}(DHIW)). It is clear that sorting by increasing height would result in solutions of inferior quality; hence this option is not considered.

[1] Lodi *et al.* [103, 105] did not consider the different methods of sorting. Instead they sorted according to decreasing height only. Therefore, in the remainder of this dissertation the names FC_{OG}(DH) and FC_{OF}(DH) algorithms will represent the original algorithms, and the names FC_{OG} and FC_{OF} algorithms will be used for the set of floor-ceiling algorithms that solve the oriented guillotine and free strip packing problems, respectively.

Worked Example

By sorting the items in Table 3.1 in order of decreasing height, the list $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_5, \mathcal{I}_{11}, \mathcal{I}_2, \mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_{13}, \mathcal{I}_8, \mathcal{I}_7, \mathcal{I}_9, \mathcal{I}_6, \mathcal{I}_{12}, \mathcal{I}_4\}$ results. Items $\mathcal{I}_1$ and $\mathcal{I}_5$ are packed onto the floor of the first level. An attempt is made to pack item $\mathcal{I}_{11}$ onto an initialised ceiling, but none exists. The item also does not fit onto the floor of an existing level, nor onto any uninitialised ceilings. Therefore, $\mathcal{I}_{11}$ is packed into a new level. There are no initialised ceilings. Hence an attempt is made to pack $\mathcal{I}_2$ onto the floor of an existing level. The item fits into the first level and is packed adjacent to $\mathcal{I}_5$. Item $\mathcal{I}_{10}$ does not fit onto the ceiling or floor of any existing levels and initialises a new level. The first two levels do not have sufficient space for $\mathcal{I}_3$ and the ceiling of the third level is not initialised; hence $\mathcal{I}_{10}$ is packed on the floor of the third level. Item $\mathcal{I}_{13}$ is too large to fit into any existing level. It initialises the fourth level. There are no initialised ceilings, but $\mathcal{I}_8$ fits onto the ceiling of the first level. Ceiling packing takes preference over floor-packings on higher levels; hence $\mathcal{I}_8$ is packed onto the ceiling of the first level and not adjacent to $\mathcal{I}_3$ in the third level. Item $\mathcal{I}_7$ does not fit onto any initialised ceiling and the only floor onto which it fits is the floor of the second level. There are no initialised ceilings with space for $\mathcal{I}_9$, nor is there space for the item on the floor of any existing level. However, $\mathcal{I}_9$ does fit onto the ceiling of the second level. During the search for a position on an initialised ceiling for $\mathcal{I}_6$, a space may be found adjacent to $\mathcal{I}_8$ on the ceiling of the first level. The guillotine version of the algorithm would pack $\mathcal{I}_6$ one unit of length further to the left in order for the solution to adhere to the guillotine constraint. Item $\mathcal{I}_{12}$ does not fit onto any ceiling, nor any existing floor; hence it is packed into a new level. There is sufficient space for $\mathcal{I}_4$ above $\mathcal{I}_3$ in the third level, resulting in a packing as illustrated in Figures 4.6(a) and 4.6(b) for the FC_{OF} and FC_{OG} algorithms, respectively.

Worst-case Time Complexity

Lodi *et al.* [106, p. 384] report that the FC algorithm requires $\mathcal{O}(n^3)$ time to complete. The loop spanning lines 2–13 is executed n times. In this *for*-loop, items are sequentially packed either onto a ceiling, or if not there, onto a floor. The nested search for a suitable position and the packing on lines 4 and 8 require $\mathcal{O}(n^2)$ time in the worst case. Therefore, the overall worst-case time complexity of the FC algorithm is $\mathcal{O}(n^3)$.

Practical Considerations

Implementations of the FC algorithms require the use of the triples as discussed in §4.1 in order to ensure that the items packed onto ceilings do not overlap the items already packed onto floors. In order to simplify the search for spaces on ceilings for unpacked items, two arrays may be used to save the feasible packing region on the ceilings of each level. The first array defines the left-hand boundary for feasible ceiling packing. This boundary is initialised with the horizontal coordinate of the right-hand side of the item that initialises the level (an item will never be packed onto the ceiling above this item as the top edge of the item defines the ceiling). For every item that is packed onto the floor, the left-hand boundary may be moved to the right if the space between the top edge of the item and the ceiling of the level is less than the next item in the list. The second array represents the right-hand boundary which is initialised at the same position as the right-hand boundary of the strip. The right-hand boundary on a level moves to the position of the left edge of an item when it is packed to a ceiling. When a place on a ceiling is sought for an unpacked item, the minimum residual horizontal space

may be found by subtracting the position of the left-hand boundary from the position of the right-hand boundary for each valid level. The updating of left-hand boundaries is unfortunately a computationally expensive procedure with a time complexity of $\mathcal{O}(n)$; each previously-packed item must be evaluated for its effect on the left boundary before the next item is to be packed.

4.2.2 Bortfeldt's Modified Best-Fit Decreasing Height Algorithm

In order to find an initial solution for his genetic algorithmic approach to the strip packing problem, Bortfeldt [18, pp. 825–826] modified the BFDH algorithm to make use of the space remaining between the top of many floor-packed items and the ceiling. He named this the BFDH* algorithm. Although the BFDH* algorithm was designed for the problem in which rotations are allowed, Bortfeldt does list changes that may be made in order to allow for items that may not be rotated. The solution is guaranteed to adhere to the guillotine constraint and is a four-stage cutting pattern in the worst case.

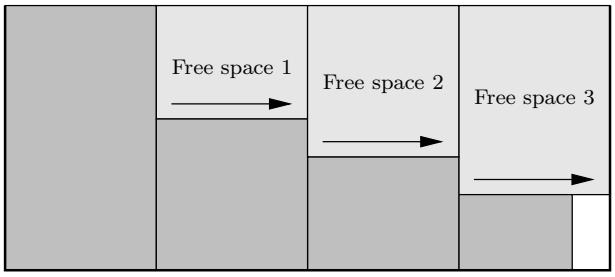


Figure 4.4: *An illustration of the free spaces utilised by the BFDH* algorithm for further packing of items.*

The algorithm begins as the BFDH algorithm presented in §3.2.3. If an item is packed into a level, the remaining space on the floor is evaluated. If that space is less than the width of the thinnest unpacked item, areas of free space are defined above the items (as illustrated in Figure 4.4). The free spaces are then filled with unpacked items, beginning with the left-most free space. The items are sorted by decreasing area and the first item that fits into a free space is packed into it. The items must be packed within the boundaries of the free space. Any unpacked items that fit into the space adjacent to the first item may be packed into the space. Once an attempt has been made to pack items into all the free spaces in the level, the algorithm packs the remaining unpacked items in a BFDH manner until the condition occurs again that leads to the definition of free spaces above floor-packed items.

Worked Example

By sorting the items in Table 3.1 according to decreasing height, the list $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_5, \mathcal{I}_{11}, \mathcal{I}_2, \mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_{13}, \mathcal{I}_8, \mathcal{I}_7, \mathcal{I}_9, \mathcal{I}_6, \mathcal{I}_{12}, \mathcal{I}_4\}$ results. Item $\mathcal{I}_1$ initialises the first level and $\mathcal{I}_5$, the next item in the list, is packed adjacent to it. There is no further space in the level for $\mathcal{I}_{11}$; hence it is initialises a new level. The two levels are evaluated for the minimum residual horizontal space resulting from a possible packing of $\mathcal{I}_2$. The first level is selected for a position for $\mathcal{I}_2$ and it is packed adjacent to $\mathcal{I}_5$. The width of the space between $\mathcal{I}_2$ and the right-hand boundary of the strip is less than the width of any unpacked item in $\mathcal{I}$. Therefore, two regions of free space are defined. The first space is between the top edge of $\mathcal{I}_5$ and the ceiling of the first level. The second region of free space is between the top edge of $\mathcal{I}_2$ and the ceiling of the level, with the width extended to be between the right edge of $\mathcal{I}_5$ and the right-hand boundary of the strip.

Algorithm 4.2 Bortfeldt's modified best-fit with decreasing height algorithm (BFDH*)

Input: A list $\mathcal{I}$ of items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items and the strip width `W`.

Output: A feasible packing of the items in $\mathcal{I}$ into a strip of width `W`.

1: sort the list of items $\mathcal{I}$ by decreasing height
2: make a copy $\mathcal{A}$ of $\mathcal{I}$, sort it by decreasing area and link its items to those in $\mathcal{I}$
3: `NumLevels` $\leftarrow 1$, $w(1) \leftarrow 0$
4: define $\mathcal{I}_F$ and $\mathcal{A}_F$ as the first unpacked items in each respective list
5: **while** there are unpacked items **do**
6: find `MinResLevel`, the level with minimum residual horizontal space
7: **if** the item does not fit onto any of the existing levels **then**
8: `NumLevels` $\leftarrow$ `NumLevels` $+ 1$, pack $\mathcal{I}_F$ on `NumLevels`
9: $h($`NumLevels`$) \leftarrow h(\mathcal{I}_F)$, $w($`NumLevels`$) \leftarrow$ `W` $- w(\mathcal{I}_F)$
10: remove the equivalent item from $\mathcal{A}$
11: **else**
12: pack item $\mathcal{I}_F$ into level `MinResLevel`
13: $w($`MinResLevel`$) \leftarrow w($`MinResLevel`$) - w(\mathcal{I}_F)$
14: determine the thinnest unpacked item $\mathcal{I}_N$
15: **if** $w($`MinResLevel`$) < w(\mathcal{I}_N)$ **then**
16: define the regions of free space above the items in `MinResLevel`
17: **while** empty regions of free space and unpacked items remain **do**
18: select the left-most empty region of free space
19: pack items in $\mathcal{A}$ onto the floor of the region until no more fit
20: remove the corresponding items from $\mathcal{I}$
21: **end while**
22: **end if**
23: **end if**
24: let F be the index of the first unpacked item in $\mathcal{I}$
25: **end while**

The unpacked item with largest area is $\mathcal{I}_{13}$, but it does not fit into the first free space. Items $\mathcal{I}_{10}$, $\mathcal{I}_3$, $\mathcal{I}_9$, $\mathcal{I}_8$, $\mathcal{I}_7$ and $\mathcal{I}_2$ also do not fit into the first free space. However, item $\mathcal{I}_6$ does fit into the first free space and is stacked onto $\mathcal{I}_5$. No further items fit into this space. Items $\mathcal{I}_{10}$, $\mathcal{I}_3$ and $\mathcal{I}_9$ do not fit into the second free space. This allows $\mathcal{I}_8$ to be stacked onto $\mathcal{I}_2$. No further items fit into this free space.

Item $\mathcal{I}_{10}$ does not fit onto any existing levels and initialises a third level. Only the third level has sufficient space for $\mathcal{I}_3$ and it is packed adjacent to $\mathcal{I}_{10}$. None of the existing levels have sufficient space for $\mathcal{I}_{13}$. It therefore initialises a fourth level. Of the existing levels, only the second level has sufficient space for $\mathcal{I}_7$. It is packed adjacent to $\mathcal{I}_{11}$ resulting in a space between the right-hand edge of $\mathcal{I}_7$ and the right-hand boundary of the strip that has a width smaller than the smallest width of the unpacked items. The area above $\mathcal{I}_7$, to the right of the right-hand edge of $\mathcal{I}_{11}$, to the left of the right-hand boundary of the strip and below the level's ceiling is designated a free space for further packing. Item $\mathcal{I}_9$ is the item with the largest area among the unpacked items and it fits into the region of free space. Therefore it is packed above $\mathcal{I}_7$. No further items fit into this free space. The last two remaining items, $\mathcal{I}_{12}$ and $\mathcal{I}_4$ do not fit into any of the existing levels and are hence packed into a new level, resulting in a solution shown in Figure 4.6(c).

Worst-case Time Complexity

The sorting in this algorithm may be performed by the merge-sort algorithm which has a time complexity of $\mathcal{O}(n \log n)$. Line 2 of the pseudocode consists of three parts. The time complexity of the first part, in which the item list is copied, is $\mathcal{O}(n)$. The second part is the sorting of the new list by decreasing area, which has time complexity $\mathcal{O}(n \log n)$ when performed with the merge sort algorithm. The third part constitutes the linking of the items in the two lists. This procedure has a time complexity of $\mathcal{O}(n^2)$, as a search of $\mathcal{O}(n)$ must be performed in the new list in order to find the item that corresponds with each of the n items in list $\mathcal{I}$. The two lines that follow have constant time complexity. Therefore, the procedures that are performed before the *while*-loop have a worst-case time complexity of $\mathcal{O}(n^2)$.

The *while*-loop spanning lines 5–25 adds a time complexity of $\mathcal{O}(n)$ to its contents as the loop may be executed for each of the items in $\mathcal{I}$. Line 6 has time complexity $\mathcal{O}(n)$, because there may be as many levels as packed items in the worst case and each of these levels would have to be evaluated for a possible packing location. All operations in the first part of the *if*-statement spanning lines 7–23 have constant time complexity. The first two lines in the second part have constant time complexity, while line 14 has a time complexity of $\mathcal{O}(n)$. The procedures contained within the *if*-statement spanning lines 15–22 has a time complexity of $\mathcal{O}(n^2)$ because for each item packed on the floor of the level, a search of $\mathcal{A}$ (of time complexity $\mathcal{O}(n)$) must be performed in order to find items that may be stacked onto it. Therefore the contents of the *while*-loop have a time complexity of $\mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n^2) = \mathcal{O}(n^2)$. Hence, the $\mathcal{O}(n^3)$ worst-case time complexity of the *while*-loop overrides the $\mathcal{O}(n^2)$ worst-case time complexity of the steps prior to it, resulting in an overall worst-case time complexity of $\mathcal{O}(n^3)$ for the BFDH* algorithm.

Practical Considerations

Bortfeldt [18] designed his algorithm to allow item rotation; hence the sorting of items according to decreasing area for the stacking procedure. It would be possible to re-sort the items when the stacking procedure takes place, but it is more time-efficient to copy the list of items when the algorithm begins. The list of items may be represented by an array of decuples (10-tuples). The information contained in the decuple includes the reference number of the item, its height, its width, its horizontal and vertical coordinates, the bin into which it is packed (used for bin packing algorithms), the level into which it is packed, a boolean variable indicating whether or not it is packed, the next item in an ordered list and previous item in an ordered list of items. Before a copy of the items is sorted according to decreasing area, one of the unused properties (for example `.int`, some arbitrary integer property) of the items may be set equal to the index value of each respective item. The second list is sorted by decreasing area and two arrays, say `A2H` and `H2A`, may be used to link the two lists of items. A *for*-loop, such as the one shown below, may be used to link the two lists.

---

sort $\mathcal{I}$ by decreasing height
make a copy of $\mathcal{I}$ called $\mathcal{A}$
for $i = 1$ **to** $|\mathcal{I}|$
    $\mathcal{A}(i)$`.int` $= i$
end for
sort $\mathcal{A}$ by decreasing area
for $i = 1$ **to** $|\mathcal{I}|$

$\texttt{A2H}(i) = \mathcal{A}(i).\texttt{int}$
$\texttt{H2A}(\texttt{A2H}(i)) = i$
end for
link the items in each list

If an item is removed from the list $\mathcal{I}$ when it is floor-packed, the array `H2A` may be used to remove the equivalent item in $\mathcal{A}$ so that it is not considered for stacking at a later stage. Similarly, if an item from $\mathcal{A}$ is stacked, the array `A2H` may be used to remove the equivalent item from $\mathcal{I}$. For example, if an item i in $\mathcal{I}$ is removed from $\mathcal{I}$, then the following steps would remove the equivalent item from $\mathcal{A}$ (note that the algorithm makes use of linked lists, described in detail in §3.2.4).

$\mathcal{I}(\mathcal{I}(i).\texttt{prv}) = \mathcal{I}(i).\texttt{nxt}$
$\mathcal{I}(\mathcal{I}(i).\texttt{nxt}) = \mathcal{I}(i).\texttt{prv}$
$\mathcal{A}(\mathcal{A}(\texttt{H2A}(i)).\texttt{prv}) = \mathcal{A}(\texttt{H2A}(i)).\texttt{nxt}$
$\mathcal{A}(\mathcal{A}(\texttt{H2A}(i)).\texttt{nxt}) = \mathcal{A}(\texttt{H2A}(i)).\texttt{prv}$

Similarly, if an item i in $\mathcal{A}$ is stacked and removed from $\mathcal{A}$, the equivalent item may be removed from $\mathcal{I}$ with the method described below.

$\mathcal{A}(\mathcal{A}(i).\texttt{prv}) = \mathcal{A}(i).\texttt{nxt}$
$\mathcal{A}(\mathcal{A}(i).\texttt{nxt}) = \mathcal{A}(i).\texttt{prv}$
$\mathcal{I}(\mathcal{I}(\texttt{A2H}(i)).\texttt{prv}) = \mathcal{I}(\texttt{A2H}(i)).\texttt{nxt}$
$\mathcal{I}(\mathcal{I}(\texttt{A2H}(i)).\texttt{nxt}) = \mathcal{I}(\texttt{A2H}(i)).\texttt{prv}$

4.2.3 The Size-Alternating Stack Algorithm

Ntene and Van Vuuren [125, 127] developed the *size-alternating stack* (SAS) algorithm in 2009. The list of items $\mathcal{I}$ is partitioned into two sublists, a sublist of narrow items $\mathcal{N}$ for which the height is greater than the width ($h(\mathcal{I}_i) > w(\mathcal{I}_i)$), and a sublist of wide items $\mathcal{W}$ for which $w(\mathcal{I}_i) \geq h(\mathcal{I}_i)$. The sublist $\mathcal{N}$ is sorted by decreasing height and the sublist $\mathcal{W}$ is sorted by decreasing width. The first elements in each list are compared and the one with largest height initialises a level. Then the first item from the other list is packed next to the item (if it fits). If the item is from $\mathcal{N}$, then other items from the list (of width less than or equal to the width of the bottom-most narrow item) may be stacked on top of it (see Figure 4.5 for an illustration of the algorithm's attempt at space utilisation). If the item is in $\mathcal{W}$, then other items from that list may be stacked onto it until insufficient vertical space remains to continue the packing in this manner. If the widths of the stacked items from $\mathcal{W}$ differ, the resulting space to the right of the upper item, and above the lower item, may be used to pack items from $\mathcal{N}$. As items are packed, they are removed from their respective sublists. If no further items fit into a level, a new level is created. A pseudocode listing of the SAS algorithm may be found in Algorithm 4.3. The solution to the SAS algorithm is a three-stage cutting pattern in the worst case.

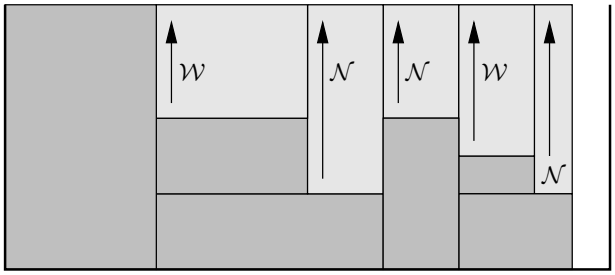


Figure 4.5: *An illustration of the free spaces utilised by the SAS algorithm in an attempt to pack levels more densely than achieved by the level-packing algorithms.*

Algorithm 4.3 Size-alternating stack algorithm (SAS)

Input: A list $\mathcal{I}$ of items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items and the strip width `W`.

Output: A packing of the items in $\mathcal{I}$ into a strip of width `W`.

1: partition the list of items $\mathcal{I} = \mathcal{N} \cup \mathcal{W}$ such that $\mathcal{N}$ is a list of items where $h(\mathcal{N}_i) > w(\mathcal{N}_i)$ for all $1 \leq i \leq |\mathcal{N}|$, while $\mathcal{W}$ is a list of items where $w(\mathcal{W}_j) \geq h(\mathcal{W}_j)$ for all $1 \leq j \leq |\mathcal{W}|$
2: sort $\mathcal{N}$ by decreasing height and sort $\mathcal{W}$ by decreasing width
3: define $\mathcal{P}$ as the set of packed items, $\mathcal{P} \leftarrow \emptyset$, `level` $\leftarrow 1$
4: **while** $\mathcal{N} \neq \emptyset$ **or** $\mathcal{W} \neq \emptyset$ **do**
5: define $\mathcal{N}_F$ and $\mathcal{W}_F$ to be the first unpacked item in either set
6: **if** $\mathcal{W} = \emptyset$ **or** $(\mathcal{N} \neq \emptyset$ **and** $h(\mathcal{N}_F) \geq h(\mathcal{W}_F))$ **then**
7: pack $\mathcal{N}_F$ onto `level`, $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{N}_F\}$, $\mathcal{N} \leftarrow \mathcal{N} \setminus \{\mathcal{N}_F\}$
8: $h($`level`$) \leftarrow h(\mathcal{N}_F)$, $w($`level`$) \leftarrow$ `W` $- w(\mathcal{N}_F)$
9: pack any other items in $\mathcal{N}$ with the same height
10: **else if** $\mathcal{N} = \emptyset$ **or** $(\mathcal{W} \neq \emptyset$ **and** $h(\mathcal{N}_F) < h(\mathcal{W}_F))$ **then**
11: pack $\mathcal{W}_F$ onto `level`, $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{W}_F\}$, $\mathcal{W} \leftarrow \mathcal{W} \setminus \{\mathcal{W}_F\}$
12: $h($`level`$) \leftarrow h(\mathcal{W}_F)$, $w($`level`$) \leftarrow$ `W` $- w(\mathcal{W}_F)$
13: pack any other items in $\mathcal{W}$ with the same height
14: **end if**
15: **while** an item from $\mathcal{N}$ or $\mathcal{W}$ fits on `level` **do**
16: **if** previous packed item $\in \mathcal{W}$ **or** no $\mathcal{W}$ fit on `level` **or** $\mathcal{W} = \emptyset$ **then**
17: **call** PACKNARROW $(\mathcal{N}, \mathcal{P}, h($`level`$), w($`level`$))$
18: **else if** previous packed item $\in \mathcal{N}$ **or** no $\mathcal{N}$ fit on `level` **or** $\mathcal{N} = \emptyset$ **then**
19: **call** PACKWIDE $(\mathcal{N}, \mathcal{W}, \mathcal{P},$ `level`$, h($`level`$), w($`level`$))$
20: **end if**
21: **end while**
22: `level` $\leftarrow$ `level` $+ 1$
23: **end while**

Worked Example

The list of items $\mathcal{I}$ in Table 3.1 is partitioned into a list $\mathcal{N}$ of narrow items and a list $\mathcal{W}$ of wide items and sorted according to decreasing width and decreasing height, respectively. The lists $\mathcal{N} = \{\mathcal{I}_1, \mathcal{I}_2\}$ and $\mathcal{W} = \{\mathcal{I}_{13}, \mathcal{I}_{11}, \mathcal{I}_5, \mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_4, \mathcal{I}_6, \mathcal{I}_9, \mathcal{I}_{12}, \mathcal{I}_7, \mathcal{I}_8\}$ result. Item $\mathcal{I}_1$ is the taller of the first items in each list; hence it initialises the first level. The first item packed was an item from $\mathcal{N}$ — therefore an item from $\mathcal{W}$ is packed next. The first item in $\mathcal{W}$ fits into the level; hence $\mathcal{I}_{13}$ is packed adjacent to $\mathcal{I}_1$. At this point the stacking procedure begins. There is

Procedure 4.3.1 PACKNARROW $(\mathcal{N}, \mathcal{P}, \texttt{Height}, \texttt{Width})$

pack first item $\mathcal{N}_i$ where $h(\mathcal{N}_i) \leq \texttt{Height}$ and $w(\mathcal{N}_i) \leq \texttt{Width}$
$\texttt{Height} \leftarrow \texttt{Height} - h(\mathcal{N}_i)$
$\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{N}_i\}$, $\mathcal{N} \leftarrow \mathcal{N} \setminus \{\mathcal{N}_i\}$
define L and N as the indices of the last and thinnest items in $\mathcal{N}$, respectively
let $\mathcal{N}_j$ be the item that follows $\mathcal{N}_i$ in the list
while $\texttt{Height} \geq h(\mathcal{N}_L)$ **and** $\texttt{Width} \geq w(\mathcal{N}_N)$ **and** $\mathcal{N} \neq \emptyset$ **and** $j \leq L$ **do**
  if $h(\mathcal{N}_j) \leq \texttt{Height}$ **and** $w(\mathcal{N}_j) \leq \texttt{Width}$ **then**
    stack $\mathcal{N}_j$, $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{N}_j\}$, $\mathcal{N} \leftarrow \mathcal{N} \setminus \{\mathcal{N}_j\}$
  end if
  set j to be the index of the item that follows $\mathcal{N}_j$ in the list $\mathcal{N}$
end while

Procedure 4.3.2 PACKWIDE $(\mathcal{N}, \mathcal{W}, \mathcal{P}, \texttt{level}, \texttt{Height}, \texttt{Width})$

pack the first item $\mathcal{W}_i$ for which $h(\mathcal{W}_i) \leq \texttt{Height}$ and $w(\mathcal{W}_i) \leq \texttt{Width}$
$\texttt{Height} \leftarrow \texttt{Height} - h(\mathcal{W}_i)$, $\texttt{Width} \leftarrow w(\mathcal{W}_i)$, $w(\texttt{level}) \leftarrow w(\texttt{level}) - w(\mathcal{W}_i)$
$\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{W}_i\}$, $\mathcal{W} \leftarrow \mathcal{W} \setminus \{\mathcal{W}_i\}$
while there is sufficient vertical space **and** $\mathcal{W} \neq \emptyset$ **do**
  search $\mathcal{W}$ for an item j for which $\texttt{Width} \geq w(\mathcal{W}_j)$
  if such an item exists **and** $\texttt{Height} \geq h(\mathcal{W}_j)$ **then**
    if $\texttt{Width} \neq w(\mathcal{W}_j)$ **then**
      call PACKNARROW $(\mathcal{N}, \mathcal{P}, \texttt{Height}, \texttt{Width} - w(\mathcal{W}_j))$
    end if
    stack $\mathcal{W}_j$, $\texttt{Height} \leftarrow \texttt{Height} - h(\mathcal{W}_j)$, $\texttt{Width} \leftarrow w(\mathcal{W}_j)$
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{W}_j\}$, $\mathcal{W} \leftarrow \mathcal{W} \setminus \{\mathcal{W}_j\}$
  end if
end while

insufficient space between $\mathcal{I}_{13}$ and the ceiling of the level for items $\mathcal{I}_{11}$, $\mathcal{I}_5$ and $\mathcal{I}_{10}$. However, item $\mathcal{I}_3$ does fit and is stacked onto $\mathcal{I}_{13}$. There is no further space between $\mathcal{I}_3$ and the ceiling, nor is there any space remaining on the floor of the level adjacent to $\mathcal{I}_{13}$, nor is there sufficient space between $\mathcal{I}_{13}$ and the level's ceiling for the stacking of any items from $\mathcal{N}$. Therefore, a new level is initialised and the taller of the first items of the two lists, item $\mathcal{I}_{11}$, is packed into the level first. The last item that was packed on the floor was an item from $\mathcal{W}$; hence an item from $\mathcal{N}$ is packed next. Item $\mathcal{I}_2$ is the only remaining item in $\mathcal{N}$ and it is packed adjacent to $\mathcal{I}_{11}$. No unpacked narrow items remain that may be stacked onto it. The widest item that fits into the remaining space on the level is $\mathcal{I}_8$ and it is packed between $\mathcal{I}_2$ and the right-hand boundary of the strip. No items with a width less than the width of $\mathcal{I}_8$ remain in $\mathcal{W}$; hence no stacking takes place and a new level is initialised. Item $\mathcal{I}_5$ is the first item to be packed into the third level. There are no remaining narrow items; hence $\mathcal{I}_{10}$ is packed adjacent to $\mathcal{I}_5$. There is sufficient space above $\mathcal{I}_{10}$ for the stacking of some unpacked items. Thus item $\mathcal{I}_4$, the widest unpacked item, is stacked onto $\mathcal{I}_{10}$. Insufficient space remains for further unpacked items in $\mathcal{W}$ and a new level is initialised with the first unpacked item in $\mathcal{W}$, namely item $\mathcal{I}_6$. Item $\mathcal{I}_9$ follows $\mathcal{I}_6$ in the list of wide items, but it is taller than $\mathcal{I}_6$ and is skipped. Item $\mathcal{I}_{12}$ does fit adjacent to $\mathcal{I}_6$ and is packed there. Item $\mathcal{I}_9$ initialises a fifth level and $\mathcal{I}_7$ is packed adjacent to it. The result is represented graphically in Figure 4.6(d).

Worst-case Time Complexity

The sorting in this algorithm may again be performed by the merge-sort algorithm which has a time complexity of $\mathcal{O}(n \log n)$. Procedure 4.3.1 has worst-case time complexity $\mathcal{O}(n)$, as the entire list of unpacked narrow items is first searched for the thinnest item, and then for an item to pack. Procedure 4.3.2 has worst-case time complexity $\mathcal{O}(n^2)$ because the list of unpacked wide items is searched for stacking and Procedure 4.3.1 is called for every wide item that is stacked. These procedures are nested within a *while*-loop in Algorithm 4.3. Therefore the overall worst-case time complexity[2] of the SAS algorithm is $\mathcal{O}(n^3)$.

Practical Considerations

The advantage that the SAS algorithm holds over the other algorithms is the fact that levels are treated one at a time. This saves the $\mathcal{O}(n)$ time taken to search for a suitable level for packing that algorithms such as FFDH, BFDH, JOIN, WFDH, B2FDH, FC_{OF}, FC_{OG} and BFDH* require. Furthermore, the approach of stacking items onto one another means that the triples discussed in §4.1 are not required to prevent the overlap of floor-packed items with items packed onto the ceiling (as seen in §4.2.1). The use of linked lists (described in detail in §3.2.4) to represent the two lists of items contributes to the speed of the algorithm.

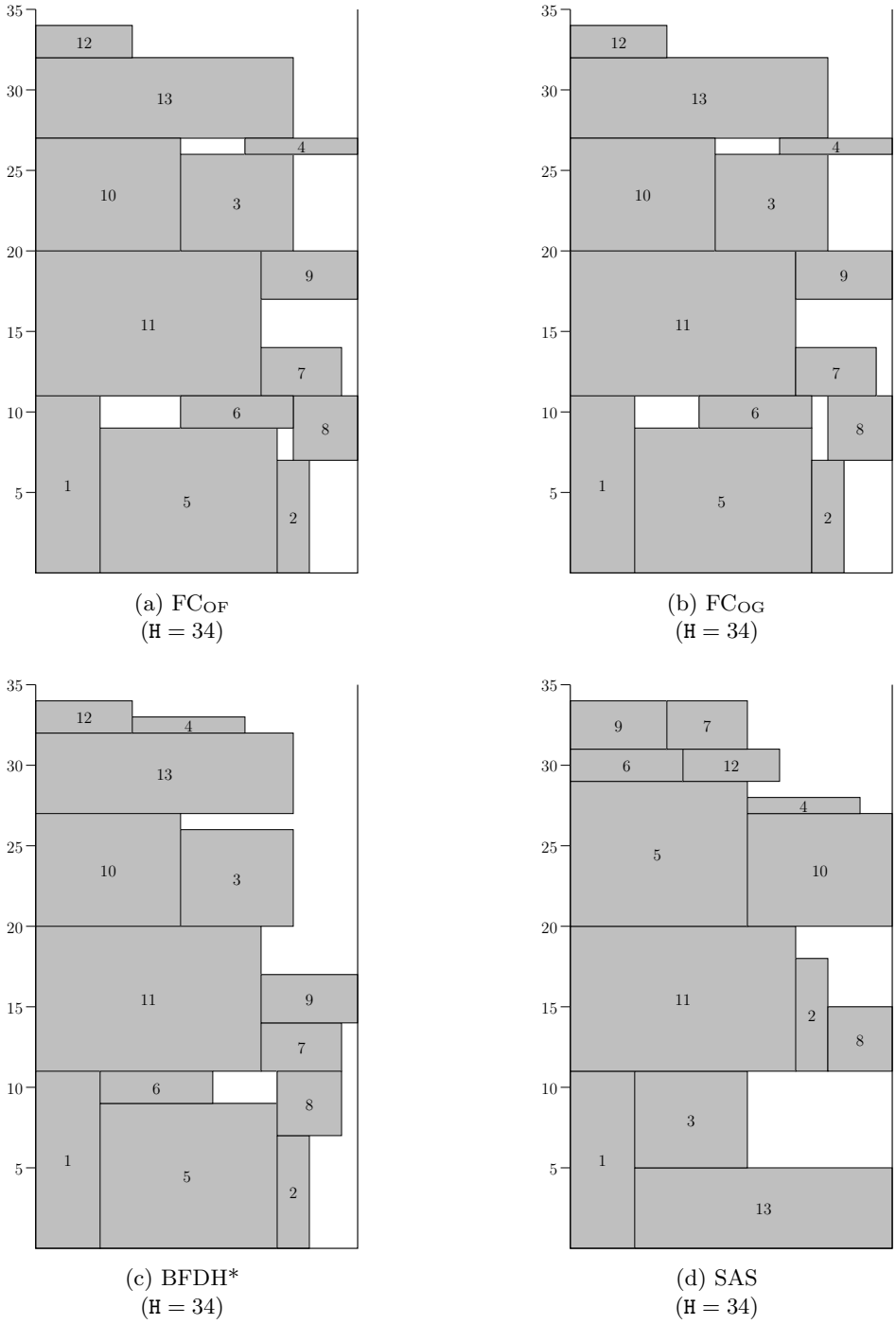
4.3 New Pseudolevel-Packing Heuristics

In this section a number of improvements to existing algorithms and a number of entirely new algorithms for the strip problem are presented in some detail. A brief introduction to each algorithm is followed by a pseudocode listing of the procedure, together with a worked example, practical considerations when implementing the algorithm and an estimation of the worst-case time complexity of the algorithm.

4.3.1 The Modified Size-Alternating Stack Algorithm

While studying the results of the SAS algorithm, it became clear that some improvements could be made to the algorithm. Sorting the items in $\mathcal{N}$ ($\mathcal{W}$) by decreasing height (width) and resolving any equalities by additionally sorting by decreasing width (height) leads to a small improvement, on average (if wider items were to be packed first, it is more likely that more items may be stacked onto it). Secondly, by searching through the entire list $\mathcal{W}$ for the tallest item $\mathcal{W}_T$ and comparing that to the first item in $\mathcal{N}$ (instead of the first item in $\mathcal{W}$) when initialising a level, an additional gain can be made as a new level will not necessarily have to be created if, say, the second item in $\mathcal{W}$ is taller than the first item of both lists. The SAS algorithm allows wide items to be stacked on top of one another and narrow items next to the wide items if there is sufficient space. By additionally allowing narrow items to be stacked on top of the last wide item on a stack, it is possible that more space may be utilised than before. The final improvement allows narrow items to be placed next to each other while stacking. The SAS algorithm only allows one narrow item to be stacked onto another. The modified space usage is illustrated in Figure 4.7 and Algorithm 4.4 contains a pseudocode listing for the SASm algorithm. The SASm algorithm produces a four-stage cutting pattern in the worst case.

[2] Ntene [125, p. 49] mistakenly notes that the worst-case time complexity of the SAS algorithm is $\mathcal{O}(n^2)$.



(a) $\mathrm{FC}_{\mathrm{OF}}$ ($\mathtt{H} = 34$)

(b) $\mathrm{FC}_{\mathrm{OG}}$ ($\mathtt{H} = 34$)

(c) BFDH* ($\mathtt{H} = 34$)

(d) SAS ($\mathtt{H} = 34$)

Figure 4.6: *Results obtained when packing items in $\mathcal{I}$ into a strip of width 20 using the known pseudolevel strip packing algorithms described in §4.2. The resulting packing heights* H *are also shown.*

Worked Example

The list of items $\mathcal{I}$ in Table 3.1 is partitioned into a list $\mathcal{N}$ of narrow items and a list $\mathcal{W}$ of wide items, sorted according to decreasing width and decreasing height, respectively. The lists $\mathcal{N} = \{\mathcal{I}_1, \mathcal{I}_2\}$ and $\mathcal{W} = \{\mathcal{I}_{13}, \mathcal{I}_{11}, \mathcal{I}_5, \mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_6, \mathcal{I}_4, \mathcal{I}_9, \mathcal{I}_{12}, \mathcal{I}_7, \mathcal{I}_8\}$ result. Item $\mathcal{I}_1$ is the tallest item and initialises the first level. The first item packed was an item from $\mathcal{N}$; hence an item from

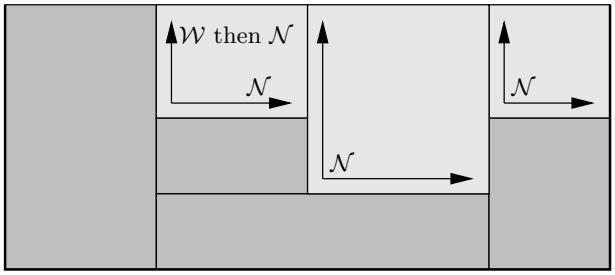


Figure 4.7: *An illustration of the free spaces utilised by the SASm algorithm in an attempt to pack levels more densely than the SAS algorithm.*

Algorithm 4.4 Modified size-alternating stack algorithm (SASm)

Input: A list $\mathcal{I}$ of items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items and the strip width `W`.

Output: A packing of the items in $\mathcal{I}$ into a strip of width `W`.

1: partition the list of items $\mathcal{I} = \mathcal{N} \cup \mathcal{W}$ such that $\mathcal{N}$ is a list of items where $h(\mathcal{N}_i) > w(\mathcal{N}_i)$ for all $1 \leq i \leq |\mathcal{N}|$, while $\mathcal{W}$ is a list of items where $w(\mathcal{W}_j) \geq h(\mathcal{W}_j)$ for all $1 \leq j \leq |\mathcal{W}|$
2: sort $\mathcal{N}$ by decreasing height and decreasing width
3: sort $\mathcal{W}$ by decreasing width and decreasing height
4: define $\mathcal{P}$ as the set of packed items, $\mathcal{P} \leftarrow \emptyset$, `level` $\leftarrow 1$
5: **while** $\mathcal{N} \neq \emptyset$ **or** $\mathcal{W} \neq \emptyset$ **do**
6: define $\mathcal{N}_F$ to be the first item in $\mathcal{N}$ and $\mathcal{W}_T$ to be the tallest item in $\mathcal{W}$
7: **if** $\mathcal{W} = \emptyset$ **or** $(\mathcal{N} \neq \emptyset$ **and** $h(\mathcal{N}_F) \geq h(\mathcal{W}_T))$ **then**
8: pack $\mathcal{N}_F$ onto `level`, $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{N}_F\}$, $\mathcal{N} \leftarrow \mathcal{N} \setminus \{\mathcal{N}_F\}$
9: $h($`level`$) \leftarrow h(\mathcal{N}_F)$, $w($`level`$) \leftarrow$ `W` $- w(\mathcal{N}_F)$
10: pack any other items in $\mathcal{N}$ with the same height
11: **else if** $\mathcal{N} = \emptyset$ **or** $(\mathcal{W} \neq \emptyset$ **and** $h(\mathcal{N}_F) < h(\mathcal{W}_T))$ **then**
12: pack $\mathcal{W}_T$ onto `level`, $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{W}_T\}$, $\mathcal{W} \leftarrow \mathcal{W} \setminus \{\mathcal{W}_T\}$
13: $h($`level`$) \leftarrow h(\mathcal{W}_T)$, $w($`level`$) \leftarrow$ `W` $- w(\mathcal{W}_T)$
14: pack any other items in $\mathcal{W}$ with the same height
15: **end if**
16: **while** an item from $\mathcal{N}$ or $\mathcal{W}$ fits on `level` **do**
17: **if** previous packed item $\in \mathcal{W}$ **or** no $\mathcal{W}$ fit on `level` **or** $\mathcal{W} = \emptyset$ **then**
18: **call** PACKNARROWMOD $(\mathcal{N}, \mathcal{P},$ `level`$, h($`level`$), w($`level`$))$
19: **else if** previous packed item $\in \mathcal{N}$ **or** no $\mathcal{N}$ fit on `level` **or** $\mathcal{N} = \emptyset$ **then**
20: **call** PACKWIDEMOD $(\mathcal{N}, \mathcal{W}, \mathcal{P},$ `level`$, h($`level`$), w($`level`$))$
21: **end if**
22: **end while**
23: `level` $\leftarrow$ `level` $+ 1$
24: **end while**

$\mathcal{W}$ is packed next. The first item in $\mathcal{W}$ fits into the existing level; hence $\mathcal{I}_{13}$ is packed adjacent to $\mathcal{I}_1$. The stacking procedure begins at this point. There is insufficient space between $\mathcal{I}_{13}$ and the ceiling of the level for items $\mathcal{I}_{11}$, $\mathcal{I}_5$ and $\mathcal{I}_{10}$. However, item $\mathcal{I}_3$ does fit and is stacked onto $\mathcal{I}_{13}$. There is no further space between $\mathcal{I}_3$ and the ceiling, nor does any space remain on the floor of the level adjacent to $\mathcal{I}_{13}$, nor is there sufficient space between $\mathcal{I}_{13}$ and the level's ceiling for the stacking of any items in $\mathcal{N}$. Therefore, a new level is initialised and the tallest unpacked item ($\mathcal{I}_{11}$) is packed into the level first. The last item that was packed on the floor was an item

Procedure 4.4.1 PACKNARROWMOD $(\mathcal{N}, \mathcal{P}, \texttt{level}, \texttt{Height}, \texttt{Width})$

pack first item $\mathcal{N}_i$ where $h(\mathcal{N}_i) \leq \texttt{Height}$ and $w(\mathcal{N}_i) \leq \texttt{Width}$
$\texttt{Height} \leftarrow \texttt{Height} - h(\mathcal{N}_i)$
if the item is being packed on the floor **then**
  $\texttt{Width} \leftarrow w(\mathcal{N}_i)$, $w(\texttt{level}) \leftarrow w(\texttt{level}) - w(\mathcal{N}_i)$
else {*the item is being stacked*}
  $\texttt{RemainingW} \leftarrow \texttt{Width} - w(\mathcal{N}_i)$
  while there is sufficient space next to $\mathcal{N}_i$ **and** $\mathcal{N} \supsetneq \mathcal{N}_i$ **do**
    search $\mathcal{N}$ for an item j for which $w(\mathcal{N}_j) \leq \texttt{RemainingW}$
    if such an item exists **and** $h(\mathcal{N}_j) \leq h(\mathcal{N}_i)$ **then**
      $\texttt{RemainingW} \leftarrow \texttt{RemainingW} - w(\mathcal{N}_j)$
      $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{N}_j\}$, $\mathcal{N} \leftarrow \mathcal{N} \setminus \{\mathcal{N}_j\}$
    end if
  end while
end if
$\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{N}_i\}$, $\mathcal{N} \leftarrow \mathcal{N} \setminus \{\mathcal{N}_i\}$
while there is sufficient vertical space **and** $\mathcal{N} \neq \emptyset$ **do**
  call PACKNARROWMOD $(\mathcal{N}, \mathcal{P}, \texttt{level}, \texttt{Height}, \texttt{Width})$
end while

Procedure 4.4.2 PACKWIDEMOD $(\mathcal{N}, \mathcal{W}, \mathcal{P}, \texttt{level}, \texttt{Height}, \texttt{Width})$

pack the first item $\mathcal{W}_i$ for which $h(\mathcal{W}_i) \leq \texttt{Height}$ and $w(\mathcal{W}_i) \leq \texttt{Width}$
$\texttt{Height} \leftarrow \texttt{Height} - h(\mathcal{W}_i)$, $\texttt{Width} \leftarrow w(\mathcal{W}_i)$, $w(\texttt{level}) \leftarrow w(\texttt{level}) - w(\mathcal{W}_i)$
$\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{W}_i\}$, $\mathcal{W} \leftarrow \mathcal{W} \setminus \{\mathcal{W}_i\}$
while there is sufficient vertical space **and** $\mathcal{W} \neq \emptyset$ **do**
  search $\mathcal{W}$ for an item j for which $w(\mathcal{W}_j) \leq \texttt{Width}$
  if such an item exists **and** $h(\mathcal{W}_j) \leq \texttt{Height}$ **then**
    if $w(\mathcal{W}_j) \neq \texttt{Width}$ **then**
      call PACKNARROWMOD $(\mathcal{N}, \mathcal{P}, \texttt{level}, \texttt{Height}, \texttt{Width} - w(\mathcal{W}_j))$
    end if
    stack $\mathcal{W}_j$, $\texttt{Height} \leftarrow \texttt{Height} - h(\mathcal{W}_j)$, $\texttt{Width} \leftarrow w(\mathcal{W}_j)$
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{W}_j\}$, $\mathcal{W} \leftarrow \mathcal{W} \setminus \{\mathcal{W}_j\}$
  end if
end while
if there is sufficient vertical and horizontal space **and** $\mathcal{N} \neq \emptyset$ **then**
  call PACKNARROWMOD $(\mathcal{N}, \mathcal{P}, \texttt{level}, \texttt{Height}, \texttt{Width})$
end if

from $\mathcal{W}$; hence an item from $\mathcal{N}$ is packed next. Item $\mathcal{I}_2$ is the only remaining item in $\mathcal{N}$; hence it is packed adjacent to $\mathcal{I}_{11}$ and no narrow items remain that may be stacked onto it. The widest item that fits into the remaining space on the level is $\mathcal{I}_8$ and it is packed between $\mathcal{I}_2$ and the right-hand boundary of the strip. There are no unpacked items in $\mathcal{W}$ or $\mathcal{N}$ with a width less than the width of $\mathcal{I}_8$; hence no stacking takes place and a new level is initialised. Item $\mathcal{I}_5$ is the tallest of the unpacked items and it initialises the third level. There are no remaining narrow items. Thus $\mathcal{I}_{10}$ is packed adjacent to $\mathcal{I}_5$. There is sufficient space above $\mathcal{I}_{10}$ for the stacking of items. Therefore item $\mathcal{I}_6$, the widest unpacked item, is stacked onto $\mathcal{I}_{10}$. Insufficient space remains for further unpacked items in $\mathcal{W}$ and a new level is initialised with the tallest unpacked item in $\mathcal{W}$, namely item $\mathcal{I}_9$. Item $\mathcal{I}_7$ is packed adjacent to $\mathcal{I}_9$ because it has the same height

as $\mathcal{I}_9$. Item $\mathcal{I}_4$ is the first item remaining in the list of unpacked items and is packed onto the floor of the fourth level, adjacent to $\mathcal{I}_7$. There is sufficient space above $\mathcal{I}_4$ for the stacking of $\mathcal{I}_{12}$, and hence $\mathcal{I}_{12}$ is stacked onto $\mathcal{I}_4$. The result is represented graphically in Figure 4.11(a).

Worst-case Time Complexity

Now that the procedure *pack narrow* allows stacked items to be packed adjacent to each other, Procedure 4.4.1 has an $\mathcal{O}(n^2)$ worst-case time complexity. Due to this fact, the modified *pack wide* procedure has time complexity $\mathcal{O}(n^3)$ which, in turn, means that the SASm algorithm has an overall worst-case time complexity of $\mathcal{O}(n^4)$.

It is possible to improve the SAS algorithm while preserving the worst-case time complexity of $\mathcal{O}(n^3)$. By calling Procedure 4.3.1 from both Algorithm 4.4 and Procedure 4.4.2, the additional complexity of packing adjacent narrow items during the stacking phase is avoided.

Practical Considerations

The practical implementation considerations for the SASm algorithm are the same as for the SAS algorithm. The use of linked lists (see §3.2.4) increases the speed of the algorithm as the items are not packed in the same order in which they appear in the two lists. The fact that items are stacked onto one another in an upward direction removes the need for data structures (and the related calculations) that monitor the skyline of the level (as discussed in §4.1).

4.3.2 The Best-Fit with Stacking Algorithm

The *best-fit with stacking* (BFS) algorithm is an attempt at improving the BFDH* algorithm by Bortfeldt [18]. It follows the same ideas as the BFDH algorithm, the major difference being that stacking (as seen in the packing of narrow items in the SASm algorithm) is allowed. Initially, instead of only sorting by decreasing height, the items are sorted by decreasing height and any equalities are resolved by sorting according to decreasing width. When the algorithm packs an item onto a floor, a procedure similar to Procedure 4.4.1 is used to stack any unpacked items that fit in order to adhere to the guillotine constraint. The change allows the stacking width to exceed the floor-packed item's width, if the space between the item and the right-hand side of the strip is less than the width of the thinnest unpacked item. The algorithm is represented in pseudocode form as Algorithm 4.5. The BFS algorithm produces a four-stage cutting pattern in the worst case.

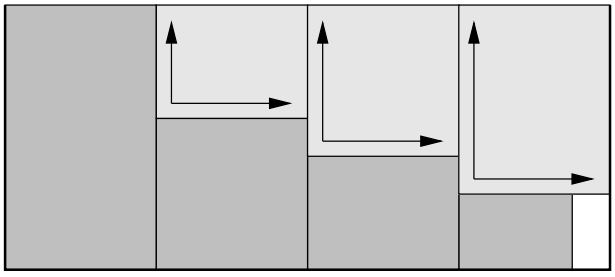


Figure 4.8: *An illustration of the free spaces utilised by the BFS algorithm in an attempt to pack levels more densely than the level packing algorithms in §3.*

Algorithm 4.5 Best-fit with stacking algorithm (BFS)

Input: A list $\mathcal{I}$ of items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items and the strip width `W`.

Output: A packing of the items in $\mathcal{I}$ into a strip of width `W`.

sort the list of items $\mathcal{I}$ by decreasing height and decreasing width
`level` $\leftarrow 1$, w(`level`) $\leftarrow 0$, `NumLevels` $\leftarrow 1$
while there are unpacked items **do**
  let $\mathcal{I}_F$ be the first unpacked item in the list $\mathcal{I}$
  find `MaxResLevel`, the level with the minimum residual horizontal space
  if the item does not fit onto any existing levels **then**
    `NumLevels` $\leftarrow$ `NumLevels` $+ 1$, pack $\mathcal{I}_F$ on `NumLevels`
    h(`NumLevels`) $\leftarrow h(\mathcal{I}_F)$, w(`NumLevels`) $\leftarrow$ `W` $- w(\mathcal{I}_F)$
  else
    call StackingBF ($\mathcal{I}$, `MinResLevel`, F, h(`MinResLevel`), w(`MinResLevel`))
  end if
end while

Procedure 4.5.1 StackingBF ($\mathcal{I}$, `level`, i, `Height`, `Width`)

define N as the index of the thinnest item in $\mathcal{I}$
if the item is being packed on the floor **then**
  pack $\mathcal{I}_i$ to the floor, w(`level`) $\leftarrow w$(`level`) $- w(\mathcal{I}_i)$
  if w(`level`) $< w(\mathcal{I}_N)$ **then**
    `Width` $\leftarrow w$(`level`) $+ w(\mathcal{I}_i)$
  else
    `Width` $\leftarrow w(\mathcal{I}_i)$
  end if
else {*the item is being stacked*}
  stack $\mathcal{I}_i$, `RemainingW` $\leftarrow$ `Width` $- w(\mathcal{I}_i)$
  while `RemainingW` $\geq w(\mathcal{I}_N)$ **do**
    search $\mathcal{I}$ for the tallest item j for which $w(\mathcal{I}_j) \leq$ `RemainingW`
    if such an item exists **and** $h(\mathcal{I}_j) \leq h(\mathcal{I}_i)$ **then**
      pack $\mathcal{I}_j$ next to $\mathcal{I}_i$, `RemainingW` $\leftarrow$ `RemainingW` $- w(\mathcal{I}_j)$
    end if
  end while
end if
if there exists an unpacked item $\mathcal{I}_j$ that may be stacked onto $\mathcal{I}_i$ **then**
  call StackingBF ($\mathcal{I}$, `level`, j, `Height` $- h(\mathcal{I}_i)$, `Width`)
end if

Worked Example

Once sorted according to decreasing height and decreasing width, the list $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_{11}, \mathcal{I}_5, \mathcal{I}_{10}, \mathcal{I}_2, \mathcal{I}_3, \mathcal{I}_{13}, \mathcal{I}_8, \mathcal{I}_9, \mathcal{I}_7, \mathcal{I}_6, \mathcal{I}_{12}, \mathcal{I}_4\}$ results for the example instance in Table 3.1. Item $\mathcal{I}_1$ initialises the first level and $\mathcal{I}_{11}$ is packed next to it. There is sufficient space between $\mathcal{I}_{11}$ and the ceiling of the level for some items to be stacked. The tallest item that fits is $\mathcal{I}_6$ and $\mathcal{I}_{12}$ is packed adjacent to it. No further items fit in the space between $\mathcal{I}_{11}$ and the ceiling of the level. Item $\mathcal{I}_5$ is too wide to be packed adjacent to $\mathcal{I}_{11}$ and initialises the second level. There is insufficient space in the first level for $\mathcal{I}_{10}$, but it does fit into the second level and is

packed there. There is sufficient space between the ceiling and $\mathcal{I}_{10}$ for item $\mathcal{I}_4$ to be stacked and $\mathcal{I}_4$ is stacked there. Insufficient space remains for further items to be stacked onto $\mathcal{I}_{10}$. The tallest unpacked item is $\mathcal{I}_2$ and it only fits into the first level and is packed between $\mathcal{I}_{11}$ and the right-hand boundary of the strip. There are no unpacked items that fit above $\mathcal{I}_2$. The next item in the list is $\mathcal{I}_3$, which does not fit into any of the existing levels and initialises the fourth level. The third and fourth levels have sufficient space to pack $\mathcal{I}_8$, but the fourth level is the level with minimum residual horizontal space. Therefore $\mathcal{I}_8$ is packed adjacent to $\mathcal{I}_{13}$ in the fourth level. Item $\mathcal{I}_9$ only fits into the third level and is packed adjacent to $\mathcal{I}_3$. There is sufficient space between $\mathcal{I}_9$ and the ceiling of the level for $\mathcal{I}_7$ to be stacked. Hence $\mathcal{I}_7$ is stacked onto $\mathcal{I}_9$. The resulting packing is shown in Figure 4.11(b).

Worst-case Time Complexity

The merge-sort algorithm may be used to sort the items and has complexity $\mathcal{O}(n \log n)$. Procedure 4.5.1 begins with the identification of the thinnest item in the list $\mathcal{I}$, an operation with a time complexity of $\mathcal{O}(n)$. The first part of the *if*-statement spanning lines 2–17 has constant time complexity, while the second part has a worst-case time complexity of $\mathcal{O}(n)$ due to the *while*-loop spanning lines 11–16 attempting to stack items adjacent to each other before the procedure is called again on line 19 to stack items on a higher sub-level. Therefore, Procedure 4.5.1 has a worst-case time complexity of $\mathcal{O}\left(n^2\right)$, because it may potentially call itself $\mathcal{O}(n)$ times. Furthermore, the *while*-loop spanning lines 3–12 of Algorithm 4.5 is executed $\mathcal{O}(n)$ times. The overall worst-case time complexity for the BFS algorithm is therefore $\mathcal{O}\left(n^3\right)$, overriding the time complexity of the sorting algorithm.

Practical Considerations

The items may not be packed in the same order in which they appear in the sorted list; hence the use of linked lists (see a detailed discussion in §3.2.4) may reduce the time required to search for unpacked items as the number of packed items increases. The upward direction of the stacking removes the need for the data structures that monitor the height of floor-packed items (discussed in §4.1) and the associated calculations (as seen in §4.2.1).

4.3.3 The Stack Level Algorithm

One advantage that the ceiling-stacking algorithms have over the floor-stacking algorithms is that items packed onto the ceiling are not restricted by the location of horizontal coordinates at which two floor-packed items meet. Consider the following example. There is a list of four items which are to be packed into a strip of width 3. The first item has dimensions $\langle w(\mathcal{L}_1), h(\mathcal{L}_1)\rangle = \langle 1, 3\rangle$ and the others have dimensions $\langle 1, 2\rangle$, $\langle 1, 2\rangle$ and $\langle 2, 1\rangle$. An algorithm such as BFS initialises the first level with $\mathcal{L}_1$ and packs $\mathcal{L}_2$ adjacent to it. Items $\mathcal{L}_3$ (too tall) and $\mathcal{L}_4$ (too wide) may not be stacked onto $\mathcal{L}_2$; hence $\mathcal{L}_3$ is packed adjacent to $\mathcal{L}_2$. Item $\mathcal{L}_4$ does not fit onto $\mathcal{L}_3$ and initialises a second level. Algorithms such as the FC group are able to pack $\mathcal{L}_4$ on the ceiling of the first level, thereby reducing the strip height, because the procedure that packs the items to the ceiling is not restricted by the horizontal boundaries of a single item (unless, of course, that floor-packed item overlaps with the ceiling-packed item). The *stack level* (SL) algorithm was developed in an attempt to relax the restriction on other floor-stacking algorithms, while returning a guillotineable solution. By borrowing the idea of linking items

from the JOIN algorithm by Martello *et al.* [110] (see §3.2.5), sublevels of greater width may be used for stacking. The stacking procedure is the same procedure used by the BFS algorithm.

The algorithm begins by sorting the list of items by decreasing height and decreasing width. The first unpacked item in the list is selected and initialises the level. The items that follow are packed in the same best-fit manner as in the BFDH and BFS algorithms. The difference in the SL algorithms is the comparison that occurs between the packed item and the item that follows in the list. If the difference in heights is within a proportion δ of the height of the packed item, and if sufficient space remains on the level, the item is packed next to the first item. If the item after the second item is also within the same height range, it too is packed adjacent to the second item. This process continues until there is either insufficient space remaining on the level, or the next item in the list is not close enough in height to the first item of this height group that was packed. Once this packing of similar items is complete, the stacking procedure begins. A region of free space above the items of similar height is defined and this is filled in a FFDH manner. The algorithm is represented in pseudocode form as Algorithm 4.6. The SL algorithm produces a four-stage cutting pattern in the worst case.

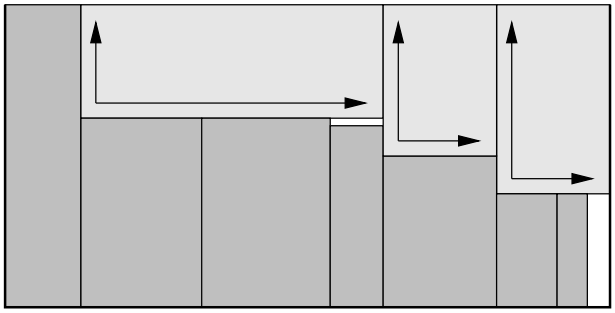


Figure 4.9: *An illustration of the free spaces utilised by the SL algorithm in an attempt to pack levels more densely than the level packing algorithms in §3.*

Algorithm 4.6 Stack level algorithm (SL)

Input: A list $\mathcal{I}$ of items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items, the strip width `W` and the percentage δ by which the heights of the items may differ.

Output: A feasible packing of the items in $\mathcal{I}$ into a strip of width `W`.

sort the list of items $\mathcal{I}$ by decreasing height and decreasing width
`NumLevels` $\leftarrow 1$, $w(1) \leftarrow$ `W`
define $\mathcal{I}_F$ as the first unpacked item in the list of items $\mathcal{I}$
while there are unpacked items **do**
  find `MinResLevel`, the level with minimum residual horizontal space
  if the item does not fit into any existing levels **then**
    `NumLevels` $\leftarrow$ `NumLevels` $+ 1$, pack $\mathcal{I}_F$ on level `NumLevels`
    $h($`NumLevels`$) \leftarrow h(\mathcal{I}_F)$, $w($`NumLevels`$) \leftarrow$ `W` $- w(\mathcal{I}_F)$
  else
    call StackingSL $(\mathcal{I}, F, \delta, h($`MinResLevel`$), w($`MinResLevel`$),$ `MinResLevel`$)$
  end if
  let F be the index of the first unpacked item in the list $\mathcal{I}$
end while

Procedure 4.6.1 STACKINGSL $(\mathcal{I}, i, \delta, \texttt{Height}, \texttt{Width}, \texttt{level})$

determine N, the index of the thinnest item in the list $\mathcal{I}$
if the item is being packed on the floor **then**
  pack $\mathcal{I}_i$ to the floor, $w(\texttt{level}) \leftarrow w(\texttt{level}) - w(\mathcal{I}_i)$, $\texttt{Width} \leftarrow w(\mathcal{I}_i)$
  define $i + j$ to be the j^{th} item after item i in the list of items, $j \leftarrow 1$
  while $\frac{h(\mathcal{I}_i) - h(\mathcal{I}_{i+j})}{h(\mathcal{I}_i)} \times 100 \leq \delta$ **and** $w(\texttt{level}) \geq w(\mathcal{I}_{i+j})$ **do**
    pack item adjacent to $\mathcal{I}_{i+j-1}$, $\texttt{Width} \leftarrow \texttt{Width} + w(\mathcal{I}_{i+j})$, $j \leftarrow j + 1$
  end while
  if $w(\texttt{level}) < w(\mathcal{I}_N)$ **then**
    $\texttt{Width} \leftarrow \texttt{Width} + w(\texttt{level})$
  end if
else {*the item is being stacked*}
  stack $\mathcal{I}_i$, $\texttt{RemainingW} \leftarrow \texttt{Width} - w(\mathcal{I}_i)$
  let j be the index after i in the list of unpacked items $\mathcal{I}$
  while $\texttt{RemainingW} \geq w(\mathcal{I}_N)$ **do**
    if $w(\mathcal{I}_j) \leq \texttt{RemainingW}$ **and** $h(\mathcal{I}_j) \leq h(\mathcal{I}_i)$ **then**
      pack $\mathcal{I}_j$ into the remaining space, $\texttt{RemainingW} \leftarrow \texttt{RemainingW} - w(\mathcal{I}_j)$
    end if
    let j be the next unpacked item in the list $\mathcal{I}$
  end while
end if
if an unpacked item $\mathcal{I}_j$ exists that may be stacked into the remaining space **then**
  call STACKINGSL $(\mathcal{I}, j, \delta, \texttt{Height} - h(\mathcal{I}_i), \texttt{Width}, \texttt{level})$
end if

Worked Example

Let $\delta = 0$, meaning that a collection of items will only allow stacking across their combined top edges if the heights of the items in the collection are the same. Once sorted according to decreasing height and decreasing width, the list $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_{11}, \mathcal{I}_5, \mathcal{I}_{10}, \mathcal{I}_2, \mathcal{I}_3, \mathcal{I}_{13}, \mathcal{I}_8, \mathcal{I}_9, \mathcal{I}_7, \mathcal{I}_6, \mathcal{I}_{12}, \mathcal{I}_4\}$ results for the example instance in Table 3.1. Item $\mathcal{I}_1$ initialises the first level and $\mathcal{I}_{11}$ is packed next to it. Item $\mathcal{I}_5$ has the same height as $\mathcal{I}_{11}$ and would be packed next to it if sufficient space remained, but the space remaining in the level is smaller than the width of $\mathcal{I}_5$; hence $\mathcal{I}_5$ remains unpacked. There is sufficient space between $\mathcal{I}_{11}$ and the ceiling of the first level for the stacking of items. The first two items that fit into this space are $\mathcal{I}_6$ and $\mathcal{I}_{12}$. Item $\mathcal{I}_5$ does not fit into the first level and initialises the second level. The next item in the list is $\mathcal{I}_{10}$. It does not fit into the first level, but it does fit adjacent to $\mathcal{I}_5$ and is packed there. Item $\mathcal{I}_2$ has the same height as $\mathcal{I}_{10}$, but there is insufficient space in the level for $\mathcal{I}_2$. However, there is sufficient space between $\mathcal{I}_{10}$ and the ceiling of the second level for the stacking of $\mathcal{I}_4$. The existing levels are evaluated for space for $\mathcal{I}_2$, and sufficient space is found in the first level. There are no unpacked items that have the same height as $\mathcal{I}_2$, nor do any items fit between $\mathcal{I}_2$ and the ceiling of the first level. The tallest remaining unpacked item is $\mathcal{I}_3$ and it initialises a third level. Item $\mathcal{I}_{13}$ is the tallest unpacked item. It does not fit into any of the existing levels and initialises a fourth level. Item $\mathcal{I}_8$ fits into the third and fourth levels, but is packed into the fourth level, because the horizontal space remaining on the fourth level after the packing of $\mathcal{I}_8$ is less than the space that would have remained on the third level had the item been packed there. Item $\mathcal{I}_9$ may only be packed into the third level, because the other levels do not contain sufficient space for it. Item $\mathcal{I}_7$ has the same height as $\mathcal{I}_9$ and is packed adjacent to $\mathcal{I}_9$,

resulting in a space of width 11 for possible stacking. However, no items remain unpacked and the algorithm terminates. The resulting strip height is 31 and the solution is shown in Figure 4.11(c).

Worst-case Time Complexity

Procedure 4.6.1 begins with a search for the width of the thinnest item in the list $\mathcal{I}$, a step that has a time complexity of $\mathcal{O}(n)$. Thereafter the procedure executes a number of steps that have constant time complexity. The *while*-loop spanning lines 5–7 has a time complexity of $\mathcal{O}(n)$, as all items in the list may be tested for suitability. The second part of the *if*-statement that spans lines 2–20 also has a worst-case time complexity of $\mathcal{O}(n)$, because all unpacked items may be evaluated for possible packing adjacent to each other. Once the *if*-statement has been executed, the procedure calls itself again (see line 22) in an attempt to pack more items adjacent to each other above those that have just been packed. The combined work of Procedure 4.6.1 thus has an overall worst-case time complexity of $\mathcal{O}\left(n^2\right)$, as the procedure may call itself $\mathcal{O}(n)$ times.

The SL algorithm begins by sorting the items using the merge-sort algorithm, which has $\mathcal{O}(n \log n)$ running time. A *while*-loop is entered after some steps that have constant time complexity. The search for the level with minimum residual horizontal space is a step with a $\mathcal{O}(n)$ time complexity. If a level exists in which the item may be packed, Procedure 4.6.1 with $\mathcal{O}\left(n^2\right)$ time complexity is called. Therefore the *while*-loop, and hence the entire SL algorithm, has a time complexity of $\mathcal{O}\left(n^3\right)$, overriding the time complexity of the sorting algorithm.

Practical Considerations

The SL algorithm uses the same approach as the other floor-stacking algorithms (such as the BFDH*, SAS and BFS algorithms). The linked lists used to preserve the order of items and their packed status improves the performance of the algorithm. The upward stacking improves the use of the space of the levels without the need to use the triples discussed in §4.1 in order to ensure items do not overlap.

4.3.4 The Stack Ceiling Algorithms

The newly proposed *stack ceiling* (SC) and *stack ceiling with re-sorting* (SCR) algorithms make use of the concept of packing onto ceilings as in the FC algorithms, and the stacking of items as in the SAS and floor-stacking algorithms. However, in these algorithms items are not stacked on the floor-packed items, instead they are stacked downwards from ceiling-packed items. Items are first sorted in order of decreasing height and any equalities are resolved by decreasing width. The entire list of items is sequentially searched to determine whether any items fit onto the floor. Those that fit onto the floor are removed from the list of unpacked items and added to the list of packed items.

When no further items fit onto the floor of a level, as many items as possible are placed onto the ceiling (the SCR algorithm first re-sorts the items according to decreasing width and height). This happens in the following manner. First, the tallest (widest for SCR) item is packed onto the ceiling. Thereafter, the list of unpacked items is searched for a rectangle that may be stacked below the ceiling-packed item. If one is found, it is stacked below the first item and if there is sufficient space next to the stacked item, further items may be stacked next to it (and further stacking may take place on those items). The list of unpacked items is then searched again for

an item that may fit onto the second item. Once no further items may be stacked below the first ceiling-packed item, the list is searched for the first item that will fit next to the first item that was packed onto the ceiling. Then an attempt is made to stack downwards onto that item. This process continues until no further items may be packed onto the ceiling. At this point the SCR algorithm re-sorts the items in order of decreasing height, resolving equalities in order of decreasing width. If any items remain, a new level is created and the process is repeated. Pseudocode listings for these two algorithms may be found in Algorithm 4.7. Packings produced by the SC algorithms are not guaranteed to be guillotineable.

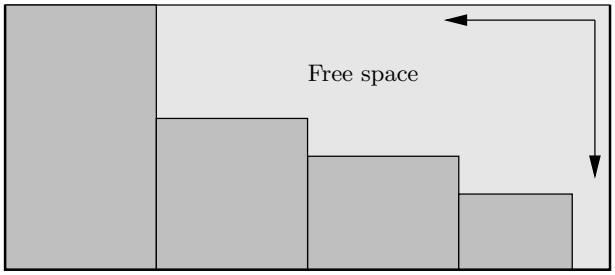


Figure 4.10: *An illustration of the free space utilised by the SC(R) algorithms in an attempt to pack levels more densely than the level packing algorithms in §3 and the FC algorithms described in §4.2.1.*

Algorithm 4.7 Stack ceiling {with re-sorting} algorithm (SC{R})

Input: A list $\mathcal{I}$ of items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items and the strip width `W`.

Output: A packing of the items in $\mathcal{I}$ into a strip of width `W`.

1: sort the list of items $\mathcal{I}$ by decreasing height and decreasing width
2: define $\mathcal{P}$ as the set of packed items, $\mathcal{P} \leftarrow \emptyset$, `level` $\leftarrow 0$
3: **while** $\mathcal{I} \neq \emptyset$ **do**
4: `level` $\leftarrow$ `level` $+ 1$, w (`level`) $\leftarrow$ `W`
5: {*Here SCR would re-sort the items by decreasing height and decreasing width*}
6: **call** PACKFLOOR ($\mathcal{I}, \mathcal{P},$ `level`)
7: **if** $\mathcal{I} \neq \emptyset$ **then**
8: {*Here SCR would re-sort the items by decreasing width and decreasing height*}
9: **call** PACKCEILING ($\mathcal{I}, \mathcal{P},$ `level`, `W`, h (`level`))
10: **end if**
11: **end while**

Worked Example

Consider, as an example, the list $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_{11}, \mathcal{I}_5, \mathcal{I}_{10}, \mathcal{I}_2, \mathcal{I}_3, \mathcal{I}_{13}, \mathcal{I}_8, \mathcal{I}_9, \mathcal{I}_7, \mathcal{I}_6, \mathcal{I}_{12}, \mathcal{I}_4\}$ from Table 3.1 sorted according to decreasing height and decreasing width. The SC algorithm attempts to fill the floor of the first level in a first-fit manner. Therefore, items $\mathcal{I}_1$, $\mathcal{I}_{11}$ and $\mathcal{I}_2$ are packed onto the floor. At this point the algorithm attempts to pack items onto the ceiling. Item $\mathcal{I}_6$ is the first item that fits and is packed into the top right-hand corner of the first level. No items exist in the list $\mathcal{I}$ that may be stacked below $\mathcal{I}_6$. Therefore, an attempt is made to pack items on the ceiling to the left of $\mathcal{I}_6$. Item $\mathcal{I}_{12}$ is the first item that fits and is packed into the level. No items in the list fit below or adjacent to $\mathcal{I}_{12}$, which results in the initialisation of a new level. Item $\mathcal{I}_5$ is the tallest unpacked item and is the first item to be packed into the second level. Item $\mathcal{I}_{10}$ is the next item in the list and fits adjacent to $\mathcal{I}_5$. No further items fit

Procedure 4.7.1 PACKFLOOR $(\mathcal{I}, \mathcal{P}, \texttt{level})$

1: define F as the index of the first unpacked item in $\mathcal{I}$, $i \leftarrow F$
2: define N as the index of the thinnest unpacked item in $\mathcal{I}$
3: **while** $w(\texttt{level}) \leq w(\mathcal{I}_N)$ **and** $\mathcal{I} \neq \emptyset$ **do**
4: **if** $w(\mathcal{I}_i) \leq w(\texttt{level})$ **then**
5: $w(\texttt{level}) \leftarrow w(\texttt{level}) - w(\mathcal{I}_i)$
6: **if** it is the first item on `level` **then**
7: $h(\texttt{level}) \leftarrow h(\mathcal{I}_i)$
8: **end if**
9: $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{I}_i\}$, $\mathcal{I} \leftarrow \mathcal{I} \setminus \{\mathcal{I}_i\}$
10: **end if**
11: let i be the index of the next unpacked item in the list $\mathcal{I}$
12: **end while**

Procedure 4.7.2 PACKCEILING $(\mathcal{I}, \mathcal{P}, \texttt{level}, \texttt{Width}, \texttt{CeilingH})$

1: `RemainingW` $\leftarrow$ `Width`
2: **while** items fit onto the ceiling with their top edges at height `CeilingH` **and** $\mathcal{I} \neq \emptyset$ **do**
3: find $\mathcal{I}_i$ such that $w(\mathcal{I}_i) \leq$ `RemainingW` and there is no overlap with floor-packed items
4: **if** such an item exists **then**
5: pack item $\mathcal{I}_i$, $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{I}_i\}$, $\mathcal{I} \leftarrow \mathcal{I} \setminus \{\mathcal{I}_i\}$
6: **call** PACKCEILING $(\mathcal{I}, \mathcal{P}, \texttt{level}, w(\mathcal{I}_i), \texttt{CeilingH} - h(\mathcal{I}_i))$
7: `RemainingW` $\leftarrow$ `RemainingW` $- w(\mathcal{I}_i)$
8: **end if**
9: **end while**

onto the floor of the level and the ceiling packing procedure begins. The only item that fits onto the ceiling is $\mathcal{I}_4$ and no unpacked items fit below or adjacent to it. The third level is initialised by $\mathcal{I}_3$, after which items $\mathcal{I}_8$ and $\mathcal{I}_9$ are packed onto the floor. No further items fit onto the ceiling and only $\mathcal{I}_7$ fits onto the ceiling. No unpacked items fit below or adjacent to $\mathcal{I}_7$ and the final level is initialised by $\mathcal{I}_{13}$. No items remain unpacked and the algorithm terminates. The resulting packing is shown in Figure 4.11(d).

The SCR algorithm makes copies of the items in $\mathcal{I}$ into a list $\mathcal{W}$ and sorts them according to decreasing width, resolving equalities according to decreasing height. The list $\mathcal{W} = \{\mathcal{I}_{13}, \mathcal{I}_{11}, \mathcal{I}_5, \mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_6, \mathcal{I}_4, \mathcal{I}_9, \mathcal{I}_{12}, \mathcal{I}_7, \mathcal{I}_1, \mathcal{I}_8, \mathcal{I}_2\}$ if formed from Table 3.1. Items $\mathcal{I}_1$, $\mathcal{I}_{11}$ and $\mathcal{I}_2$ are packed as in the SC algorithm. The ceiling packing is now performed with the order of items in the list $\mathcal{W}$. Item $\mathcal{I}_6$ is the first item that fits and is packed as in the SC algorithm. Item $\mathcal{I}_4$ is the next item in the list and is packed adjacent to $\mathcal{I}_6$. No further unpacked items fit onto the ceiling of the first level. The second level is filled in a first-fit manner with the item order in list $\mathcal{I}$. Items $\mathcal{I}_5$ and $\mathcal{I}_{10}$ are packed onto the floor and $\mathcal{I}_{12}$ is the first item that fits onto the ceiling. The remaining items are packed as in the SC algorithm. The resulting packing is shown in Figure 4.11(e).

Worst-case Time Complexity

Initially both algorithms use the merge-sort algorithm to sort the list of items. Procedure 4.7.1 has time complexity $\mathcal{O}(n)$. The first line has constant time complexity, while finding the thinnest item in a list is a step with a time complexity of $\mathcal{O}(n)$. Thereafter, the *while*-loop

evaluates every item for floor packing in the worst case; a step that has a time complexity of $\mathcal{O}(n)$. Procedure 4.7.2 has time complexity $\mathcal{O}(n^4)$, as for every item that is packed onto the ceiling, an attempt is made to stack the remaining items below it and next to it, and the floor-packed items must be considered for overlapping for every item packed. In the worst case, Procedure 4.7.2 has higher time complexity than Procedure 4.7.1 and the merge-sort algorithm. The total worst-case time complexity of the SC and SCR algorithms is therefore $\mathcal{O}(n^5)$ due to the additional time complexity of the *while*-loop in Algorithm 4.7.

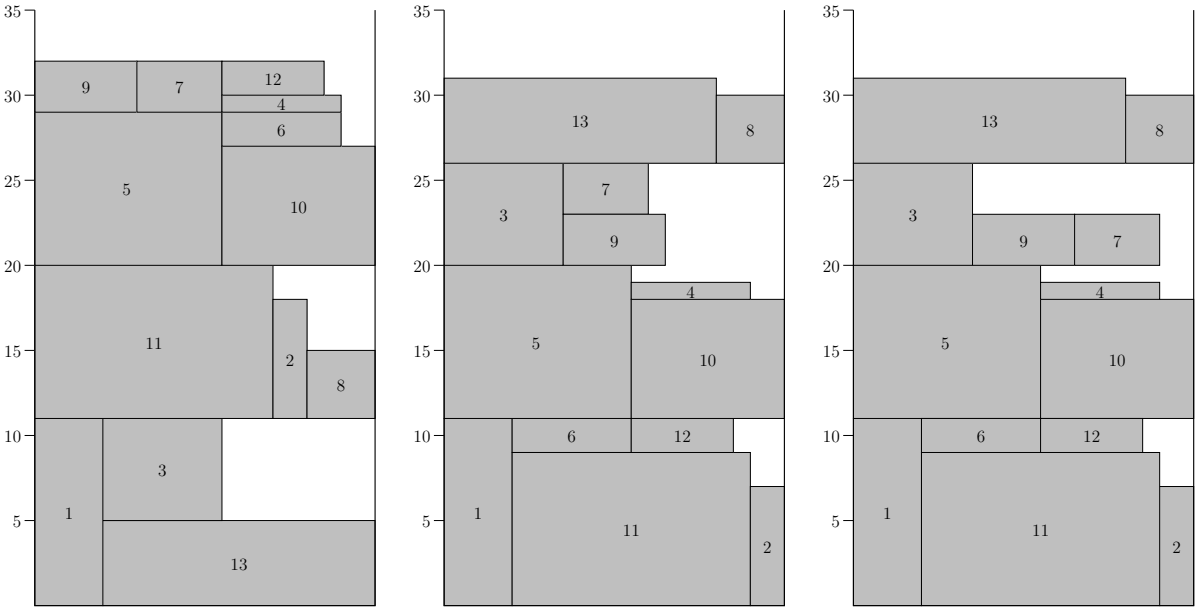
Practical Considerations

The SC and SCR algorithms stack items from the ceiling in a downward manner. Therefore, the use of the triples presented in §4.1 is crucial. The ceiling packing and stacking procedures may pack items different from their initial order in the list of items. The use of linked lists (see the practical considerations subsection of §3.2.4) to represent the sorted list of items may improve the running time. The use of an additional copy of the items to represent the list when sorted by width improves the execution time of the SCR algorithm dramatically. This is the same approach as discussed in §4.2.2.

4.4 Chapter Summary

Published pseudolevel heuristics for the strip packing problem were reviewed in this chapter, in fulfilment of Dissertation Objective IV(b), as stated in §1.3. The floor-ceiling algorithms were discussed first, followed by the oriented version of the BFDH* algorithm that Bortfeldt [18] developed for metaheuristics. Thereafter the SAS algorithm of Ntene and Van Vuuren [125,127] was discussed in some detail.

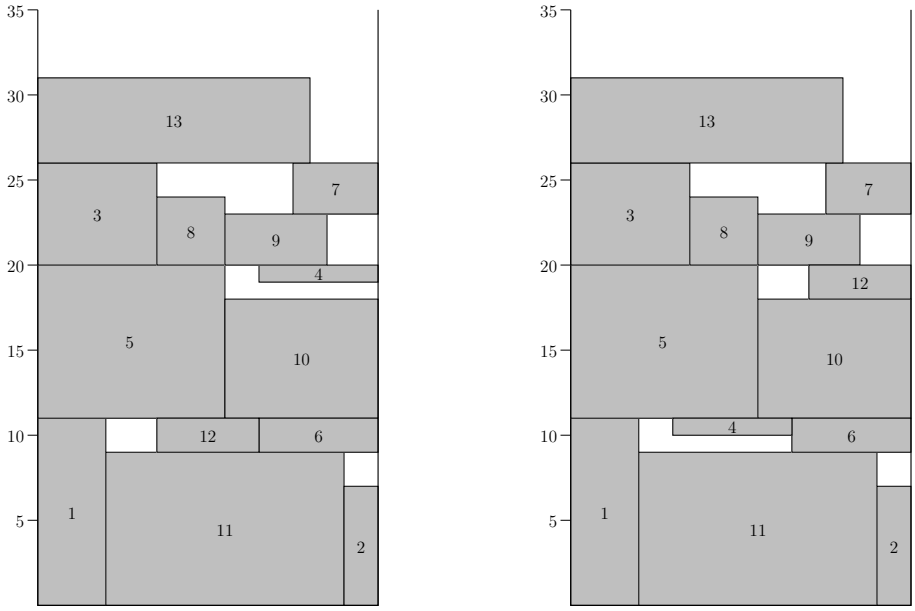
The second section of the chapter contains five improvements on the heuristics presented in the first section. The SAS algorithm was improved to arrive at the SASm algorithm. The BFS algorithm allows for more stacking in an attempt to improve on Bortfeldt's original algorithm. The SL algorithm makes use of the stacking principle found in the BFS algorithm and combines it with the joining principle of algorithm JOIN (see §3.2) in an attempt to pack items more densely. Finally, the SC and SCR algorithms were introduced with the aim of improving on the FC algorithms. These algorithms were introduced in fulfilment of Dissertation Objective V(b), as stated in §1.3.



(a) SASm (H = 32)

(b) BFS (H = 31)

(c) SL (H = 31)



(d) SC (H = 31)

(e) SCR (H = 31)

Figure 4.11: *Results obtained when packing the items in $\mathcal{I}$ (Table 3.1) into a strip of width 20 by means of the four new strip packing algorithms introduced in §4.3. The respective strip heights* H *are also shown.*

CHAPTER 5

Plane-Packing Strip Packing Heuristics

Contents

The strip packing algorithms considered in the two previous chapters packed items into levels. The heights of the levels generated by those algorithms were defined by the heights of the tallest items in the levels. Level algorithms (see §3) restrict the items to be packed on the floor, while pseudolevel algorithms (see §4) pack items anywhere within the boundaries of the levels. Plane algorithms pack items into a strip without the restriction of levels. Instead, plane algorithms may pack items anywhere within the boundaries of the strip subject only to the constraint that none of the items overlap.

5.1 Known Plane-Packing Algorithms

In this section nine known plane-packing algorithms for strip packing are reviewed in some detail. A brief introduction to each algorithm is followed by a pseudocode listing of the procedure together with a worked example, the algorithm's performance bounds (if they have been established), an analysis of the worst-case time complexity and a note on practical considerations associated with the programming of the algorithms.

5.1.1 Sleator's Algorithm

In 1980 Sleator [148] published his strip packing heuristic in the first of many papers on the topic to appear that year. In Sleator's algorithm (abbreviated as the S algorithm) all items of width larger than half the strip width are packed on top of one another. The remaining items are sorted according to decreasing height. The height of the top edge of the final item to be packed is denoted h_0. One level of items are packed at a height of h_0 until insufficient space remains for any unpacked items to be packed. A line is drawn that divides the strip into two half-strips. The top of the tallest items in either half-strip define the left and right baselines. At this stage, the right baseline is no taller than the left baseline.

The half with the lower baseline is selected. The unpacked items, in order of decreasing height, are packed onto this line until no further items may be packed into the half-level. Then the ceiling of the half-level becomes the baseline. The algorithm continues selecting the lower baseline and packs items onto it until no unpacked items remain. The result of the packing is not guaranteed to be guillotineable. If the line dividing the strip into two halves divides an item and the item is not of the same height as the tallest item in the level, the result is not guillotineable. A pseudocode listing of Sleator's algorithm may be found in Algorithm 5.1.

Algorithm 5.1 Sleator's algorithm

Input: A list $\mathcal{I}$ of items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items and the strip width `W`.

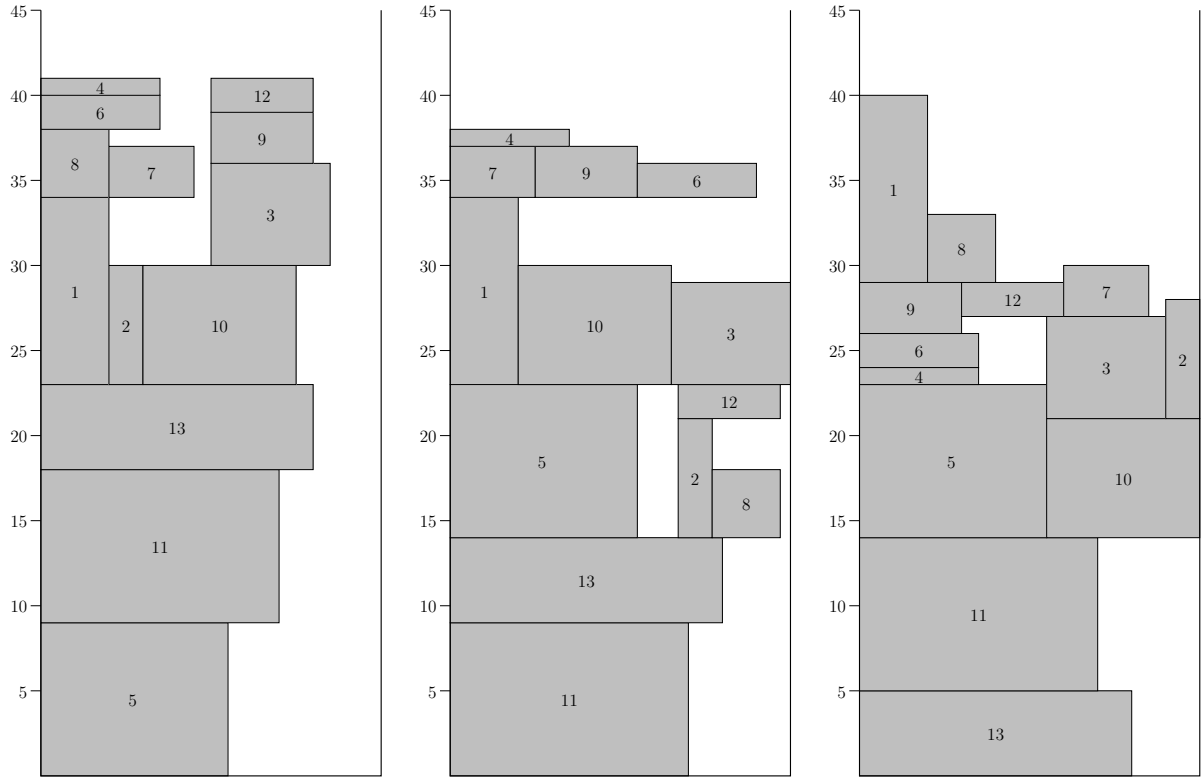
Output: A packing of the items in $\mathcal{I}$ into a strip of width `W`.

```
sort items according to decreasing height
stack all items of width > 0.5 × W
let i be the index of the first unpacked item, RemainingW ← W
while w(I_i) ≤ RemainingW do
    pack I_i at a height of h_0
    RemainingW ← RemainingW − w(I_i)
    let i be the index of the next unpacked item in I
end while
determine the height of the left and right columns
while there are unpacked items do
    RemainingW ← 0.5 × W
    if height of the left column ≤ height of the right column then
        while w(I_i) ≤ RemainingW do
            pack I_i into the left-hand column
            RemainingW ← RemainingW − w(I_i)
            let i be the index of the next unpacked item
        end while
    else
        while w(I_i) ≤ RemainingW do
            pack I_i into the right-hand column
            RemainingW ← RemainingW − w(I_i)
            let i be the index of the next unpacked item
        end while
    end if
end while
```

Worked Example

By sorting the items in Table 3.1 in order of decreasing height, the list $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_5, \mathcal{I}_{11}, \mathcal{I}_2, \mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_{13}, \mathcal{I}_8, \mathcal{I}_7, \mathcal{I}_9, \mathcal{I}_6, \mathcal{I}_{12}, \mathcal{I}_4\}$ results. The algorithm stacks those items that are wider than half the strip width onto each other; that is items $\mathcal{I}_5$, $\mathcal{I}_{11}$ and $\mathcal{I}_{13}$. The tallest remaining item is $\mathcal{I}_1$ and it is packed above $\mathcal{I}_{13}$. Items $\mathcal{I}_2$ and $\mathcal{I}_{10}$ follow in the list and are packed to the right of $\mathcal{I}_1$. No further packing takes place at this height, because the next item does not fit into the space between $\mathcal{I}_{10}$ and the right-hand boundary of the strip. The strip is split into two halves at a horizontal coordinate of 10; half of the strip width. The height of the left-hand half is 11 and the right-hand half has a height of 7. Item $\mathcal{I}_3$ follows and is packed into the right-hand half of the strip, the lower of the two halves.

The height of the right-hand half is now 13 and no further items fit between $\mathcal{I}_3$ and the strip boundary. The tallest unpacked item is $\mathcal{I}_8$ and it is packed into the left-hand side of the strip (the lower of the two halves), resulting in a left-hand half height increase to 15. Item $\mathcal{I}_7$ fits in the remaining space in that half and is packed adjacent to $\mathcal{I}_8$. The right-hand side is now the lower half; hence item $\mathcal{I}_9$ is packed onto $\mathcal{I}_3$, resulting in a height of 16 for the right-hand half. Therefore, $\mathcal{I}_6$ is packed into the lower left-hand half, which is then taller than the right-hand half. This means that $\mathcal{I}_{12}$ is packed into the right-hand side, while $\mathcal{I}_4$ is packed into the left-hand side to result in a final strip height of 41. The result is shown graphically in Figure 5.1(a).



(a) S (H = 41) (b) SF (H = 38) (c) BL (H = 40)

Figure 5.1: *Results obtained when packing the items in Table 3.1 into a strip of width 20 using the known plane strip packing algorithms described in §5.1.1–§5.1.3. The resulting packing heights* H *are also shown.*

Known Performance Bounds

Sleator [148] proved that a worst-case performance bound of his algorithm is

$$\mathtt{S}(\mathcal{L}) \leq 2\,\mathtt{OPT}(\mathcal{L}) + \frac{1}{2}h_{\text{tall}},$$

where $\mathtt{S}(\mathcal{L})$ denotes the packing height achieved by the algorithm for a list of items $\mathcal{L}$, where $\mathtt{OPT}(\mathcal{L})$ denotes the optimal packing height for that list of items and where h_{tall} is the height of the tallest item. Since $h_{\text{tall}} \leq \mathtt{OPT}(\mathcal{L})$, this may be rewritten as

$$\mathtt{S}(\mathcal{L}) \leq 2.5\,\mathtt{OPT}(\mathcal{L}).$$

Sleator showed that the time complexity of his algorithm is dominated by the sorting step because the time required for item placement has constant time complexity. Therefore, the time complexity of the algorithm is $\mathcal{O}(n \log n)$ if an algorithm such as merge-sort is used to perform the sorting procedure.

5.1.2 The Split-Fit Algorithm

Coffman *et al.* [32] introduced the *split-fit* (SF) algorithm in addition to the NFDH and FFDH algorithms described in §3.2. They define a parameter m as the largest integer for which all items have width $1/m$ or less (for a normalised item list). The list of items $\mathcal{L}$ is then split into two sublists. Sublist $\mathcal{W}$ (for wide) contains all items in $\mathcal{L}$ for which $w(\mathcal{L}_i) > \mathtt{W}/(m+1)$, while sublist $\mathcal{N}$ (for narrow) contains all items for which $w(\mathcal{L}_i) \leq \mathtt{W}/(m+1)$. All items in $\mathcal{W}$ are packed according to the FFDH algorithm. Then all levels are rearranged such that the levels for which $w(\texttt{level}) > \mathtt{W} \times (m+1)/(m+2)$ (the first set of levels) are below those for which $w(\texttt{level}) \leq \mathtt{W} \times (m+1)/(m+2)$ (the second set of levels). An area thus emerges above the first set of levels and to the left of the second set of levels. This region R has a height equal to the sum of the heights of the "narrow" levels resulting from the FFDH packing of the $\mathcal{W}$ items, and a width of $w(R) = \mathtt{W}/(m+2)$. This region may be filled with items from $\mathcal{N}$ such that no items exceed the boundaries of region R. All remaining items in $\mathcal{N}$ may be packed above the top-most level packed with items from $\mathcal{W}$ by means of the FFDH algorithm. The resulting packing is guillotineable. A pseudocode listing of the algorithm may be found in Algorithm 5.2.

Worked Example

The algorithm begins by finding $m = 1$ for the items in Table 3.1, because the widest item has a width of 16. Therefore sublist $\mathcal{W}$ is populated with the items $\mathcal{I}_5$, $\mathcal{I}_{11}$ and $\mathcal{I}_{13}$, while sublist $\mathcal{N}$ is populated with the remaining items. These items are packed by the FFDH algorithm; hence $\mathcal{I}_5$ is packed first and is followed by $\mathcal{I}_{11}$ and $\mathcal{I}_{13}$. Items $\mathcal{I}_{11}$ and $\mathcal{I}_{13}$ have a width greater than $2/3 \times \mathtt{W}$ and $\mathcal{I}_5$ has a width less than this; hence the levels are rearranged, resulting in $\mathcal{I}_5$ being packed above $\mathcal{I}_{11}$ and $\mathcal{I}_{13}$. A space R of width $1/3 \times \mathtt{W}$ and height equal to the height of $\mathcal{I}_5$ remains between $\mathcal{I}_5$ and the right-hand strip boundary. This space may be filled from items in $\mathcal{N}$.

An attempt is made to fill this space according to the FFDH algorithm. The first item, $\mathcal{I}_1$, is too tall to be packed into the space, but $\mathcal{I}_2$ is small enough to fit and is the first item to be packed. Items $\mathcal{I}_{10}$ and $\mathcal{I}_3$ are too wide to fit into R, allowing $\mathcal{I}_8$ to be packed adjacent to $\mathcal{I}_2$. The remaining space is too narrow for any of the remaining items to be packed and a new

Algorithm 5.2 Split-fit algorithm (SF)

Input: A list $\mathcal{I}$ of items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items and the strip width $\mathtt{W}$.

Output: A packing of the items in $\mathcal{I}$ into a strip of width $\mathtt{W}$.

find the widest item $\mathcal{I}_W$
$m \leftarrow \lfloor \mathtt{W} / w(\mathcal{I}_w) \rfloor$, $\mathtt{NumW} \leftarrow 0$, $\mathtt{NumN} \leftarrow 0$
for $i \leftarrow 1$ **to** $|\mathcal{I}|$ **do**
  if $w(\mathcal{I}_i) > \mathtt{W}/(m+1)$ **then**
    $\mathcal{W} \leftarrow \mathcal{W} \cup \{\mathcal{I}_i\}$
  else if $w(\mathcal{I}_i) \leq \mathtt{W}/(m+1)$ **then**
    $\mathcal{N} \leftarrow \mathcal{N} \cup \{\mathcal{I}_i\}$
  end if
end for
if $\mathcal{N} = \emptyset$ **or** $\mathcal{W} = \emptyset$ **then**
  pack the items using the FFDH algorithm
else
  pack the items in $\mathcal{W}$ with the FFDH algorithm
  for all levels **do**
    if $w(\mathtt{level}) > \mathtt{W} \times (m+1)/(m+2)$ **then**
      move $\mathtt{level}$ to the bottom
    else if $w(\mathtt{level}) \leq \mathtt{W} \times (m+1)/(m+2)$ **then**
      move $\mathtt{level}$ to the top
    end if
  end for
  create the region R above the wide levels and to the right of the narrow levels
  $w(R) \leftarrow \mathtt{W}/(m+2)$, $h(R) \leftarrow \sum$ narrow level heights
  pack items from $\mathcal{N}$ into region R with the FFDH algorithm
  if $\mathcal{N} \neq \emptyset$ **then**
    redefine R as the space above the packed items, $w(R) \leftarrow \mathtt{W}$, $h(R) \leftarrow \infty$
    pack items remaining in $\mathcal{N}$ into region R with the FFDH algorithm
  end if
end if

level is generated within the area. This level is large enough to accommodate $\mathcal{I}_{12}$, but no other items may be packed into the region. Therefore, items $\mathcal{I}_1$, $\mathcal{I}_{10}$, $\mathcal{I}_3$, $\mathcal{I}_7$, $\mathcal{I}_9$, $\mathcal{I}_6$ and $\mathcal{I}_4$ are packed into the strip above $\mathcal{I}_5$ by means of the FFDH algorithm, resulting in a strip height of 38. The packing is shown graphically in Figure 5.1(b).

Known Performance Bounds

Coffman *et al.* [32] established the asymptotic performance bound

$$\mathtt{SF}(\mathcal{L}) \leq \frac{3}{2}\mathtt{OPT}(\mathcal{L}) + 2$$

for the SF algorithm, where $\mathtt{SF}(\mathcal{L})$ denotes the packing height achieved by the SF algorithm and $\mathtt{OPT}(\mathcal{L})$ denotes the optimal packing height for a list of items $\mathcal{L}$. However, the best available worst-case performance bound is

$$\mathtt{SF}(\mathcal{L}) \leq 3\,\mathtt{OPT}(\mathcal{L}),$$

which is worse than the worst-case performance bound of the FFDH algorithm.

Worst-case Time Complexity

Finding the widest item in the list (see line 1) is an operation of order $\mathcal{O}(n)$ time complexity, because the widths of each of the items in the packing list are compared to the widest item that had been found prior to it. The operations on the line that follows have constant time complexity. The *for*-loop spanning lines 3–9 has a time complexity of $\mathcal{O}(n)$, because each item is sorted into either the wide or narrow sublists. If either of the two sublists of items is empty, then line 11 is executed with a worst-case time complexity of $\mathcal{O}(n^2)$. If there are items in both sublists, then line 13 is executed with a worst-case time complexity of $\mathcal{O}(n^2)$. This is followed by the sorting of levels and their shift either up or down in the *for*-loop spanning lines 14–20. This procedure has a worst-case time complexity of $\mathcal{O}(n)$ because each level may potentially contain only one item. The creation of the region R in lines 21 and 22 has constant time complexity, but the packing of narrow items into this region (see line 23) has a worst-case time complexity of $\mathcal{O}(n^2)$. If unpacked narrow items remain, then the remaining items are packed into a new region. This last packing procedure also has a worst-case time complexity of $\mathcal{O}(n^2)$. Therefore, the SF algorithm has a worst-case time complexity of $\mathcal{O}(n^2)$.

Practical Considerations

In order to move the wide levels to the bottom of the packing, a *for*-loop is entered that packs the levels from the floor of the strip upwards. This is achieved by maintaining a new array of quintuples that represents the levels. A quintuple contains the vertical coordinate of a level in a strip or bin, the height of the level, the width of the level (this may represent the sum of widths of the items in the level, or the horizontal space remaining in the level), the bin into which it is packed, and a boolean variable assuming the value true if the level has been packed into a bin, or the value false if it has not been packed into a bin. One of the output parameters of the algorithms programmed in the decision support system (described in detail later in this dissertation) is the array of quintuples representing the levels. The levels are moved according to the method shown in Figure 5.2. First the wide levels from the set of levels $\mathcal{L}$ are moved to a new set of levels, $\mathcal{F}$ (say).

```
NumLevels ← 0
Height ← 0
for i = 1 to UBound(L)
    if L(i).W > W × (m + 1) / (m + 2) then
        NumLevels ← NumLevels + 1
        F(NumLevels).H ← L(i).H
        F(NumLevels).Y ← Height
        Height ← Height + L(i).H
        L(i).bin ← NumLevels
        L(i).packed ← True
    end if
end for
```

Figure 5.2: *The assignment of wide levels to their correct place during execution of the SF algorithm.*

Once the *for*-loop is completed the boundaries of the region R may be defined. The region is represented by a quadruple, where one element represents the horizontal coordinate of the bottom, left-hand corner of the region, another represents its vertical coordinate and the remaining two represent the height and width of the region. After three of the properties of R have been defined, the narrow levels are packed according to a *for*-loop, as shown in Figure 5.3. The height of the region R is calculated during the loop and this property is defined once the loop is completed.

```
R.W ← W / (m + 2)
R.X ← W − R.W
R.Y ← Height
for i = 1 to UBound(L)
    if L(i).W ≤ W × (m + 1) / (m + 2) then
        NumLevels ← NumLevels + 1
        F(NumLevels).H ← L(i).H
        F(NumLevels).Y ← Height
        Height ← Height + L(i).H
        L(i).bin ← NumLevels
        L(i).packed ← True
    end if
end for
R.H ← Height − R.Y
```

Figure 5.3: *The assignment of narrow levels during an execution of the SF algorithm.*

The items may then be packed into their new positions according to the steps outlined in Figure 5.4. The items in $\mathcal{N}$ may then be packed into the region R. Any unpacked items may finally be packed into a region above the items in $\mathcal{W}$. During this part of the algorithm the linked lists implementation discussed in the *practical considerations* section of §3.2.4 is used to remove items from $\mathcal{N}$.

```
Height ← 0
for i = 1 to UBound(W)
    I(W(i).int).X ← W(i).X
    I(W(i).int).lvl ← L(i).lvl
    I(W(i).int).Y ← F(L(W(i).lvl).bin).Y
    I(W(i).int).packed ← True
end for
```

Figure 5.4: *The packing of items after their level assignment during an execution of the SF algorithm.*

5.1.3 The Bottom-Up Left-Justified Algorithm

The *bottom-up left-justified* (BL) class of algorithms was first published by Baker *et al.* [6] in 1980. An algorithm in this class packs an item as low as possible and then as far left as possible at that height. Some variations Baker *et al.* considered include those where items are sorted by increasing or decreasing width, and where items are sorted by increasing or decreasing height.

Unfortunately Baker *et al.* [6] do not provide a detailed explanation on a method that may be used to perform such packings computationally. Therefore, the pseudocode and subsequent time complexity analysis are according to the author's understanding of this class of algorithms. The concept of a skyline, as defined by Burke *et al.* [23] and discussed in some detail in §4.1, makes it possible to represent the shape of the packing at any time during execution of the heuristic.

The first step in a BL algorithm is to sort the items. However, this is not as important a step for the plane algorithms that do not split the items into groups as it is for the level and pseudolevel algorithms. The items are packed sequentially. First, the lowest point that can accommodate an item must be found. This is trivial when the lowest segment of the skyline is wide enough for the item. However, if the lowest segment is shorter than the item is wide, then the segments are raised to the height of their shortest neighbours until the lowest segment is wide enough to accommodate the item. The item is placed left-justified on the lowest segment (of sufficient width) of the strip. This procedure is illustrated in Figure 5.5.

After an item has been packed, the skyline is updated and the next item in the list is considered for packing. This process continues until all items have been packed. The resulting packing is not guaranteed to be guillotineable. Other versions of the BL class of algorithms exist. Chazelle [25], Jakobs [83], Liu and Teng [100] and Hopper and Turton [75,77,78] all investigated some of these variations. A pseudocode listing of the general algorithm may be found in Algorithm 5.3.

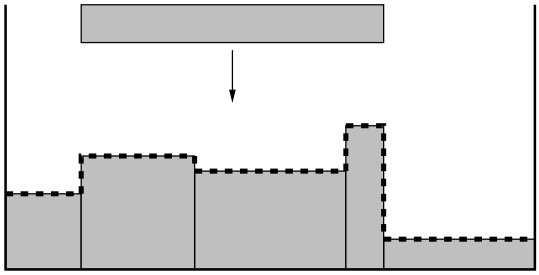
Algorithm 5.3 Bottom-Up Left-Justified (BL) algorithm

Input: A list $\mathcal{I}$ of n items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items and the strip width `W`.
Output: A packing of the items in $\mathcal{I}$ into a strip of width `W`.

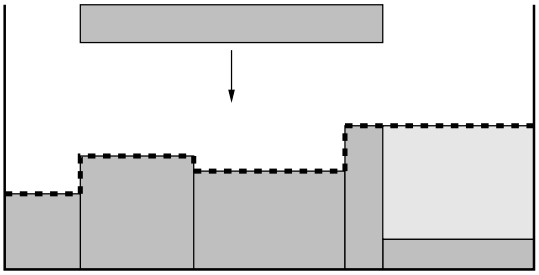
1: sort the items in any manner, if required
2: initialise a skyline $\mathcal{S}$ to monitor the space available for packing
3: **for** $i = 1$ **to** n **do**
4: **call** FINDLOWESTPOINT $(\mathcal{I}, i, \mathcal{S}, s, \texttt{Height})$
5: move $\mathcal{I}_i$ as far to the left as possible at height `Height`
6: update the skyline in order to reflect the additional item
7: **end for**

Worked Example

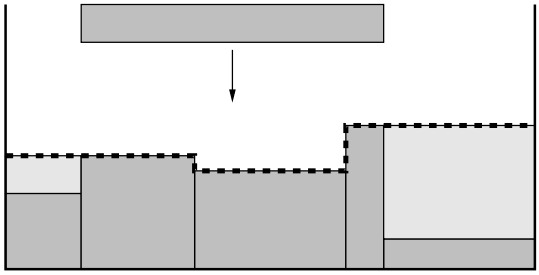
By sorting the items in Table 3.1 in order of decreasing width, the list $\mathcal{I} = \{\mathcal{I}_{13}, \mathcal{I}_{11}, \mathcal{I}_5, \mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_4, \mathcal{I}_6, \mathcal{I}_9, \mathcal{I}_{12}, \mathcal{I}_7, \mathcal{I}_1, \mathcal{I}_8, \mathcal{I}_2\}$ results. The first item, $\mathcal{I}_{13}$, is placed in the bottom-left corner of the strip and the skyline is updated to consist of two parts; the part above $\mathcal{I}_{13}$ and the part on the bottom boundary of the strip. The lowest part of the strip is to the right of $\mathcal{I}_{13}$, but it is too narrow for $\mathcal{I}_{11}$ which results in that part of the skyline being raised to the height of the left-hand section of the skyline. Now the lowest skyline segment has the width of the strip and wide enough to accommodate $\mathcal{I}_{11}$, resulting in its packing on top of $\mathcal{I}_{13}$ against the left-hand boundary of the strip. The skyline is updated to consist of three sections; the top edge of $\mathcal{I}_{11}$, the part of the top edge of $\mathcal{I}_{13}$ that is not below $\mathcal{I}_{11}$, and the bottom boundary of the strip. The right-hand section of the skyline is the lowest segment, but its width is less than that of $\mathcal{I}_5$; the segment is therefore raised to the height of the middle section. This combined skyline segment is now the lowest, but is still too narrow to accommodate $\mathcal{I}_5$, resulting in the segment being raised to the height of the top edge of $\mathcal{I}_{11}$. This new temporary segment spans the width



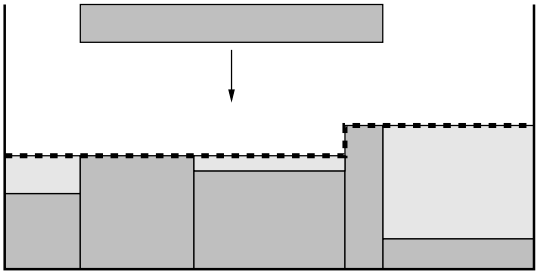
(a) The lowest possible location is sought in order to pack the item



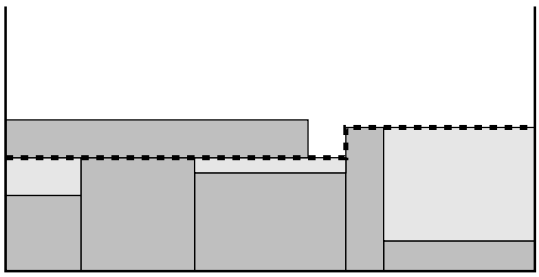
(b) The lowest temporary skyline segment is raised to the height of its lowest neighbour



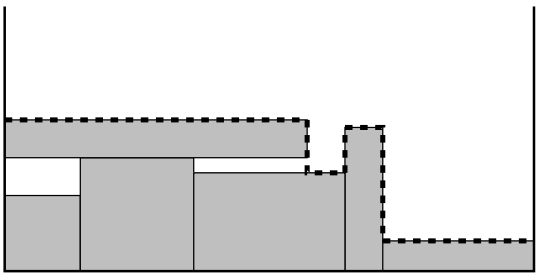
(c) The item does not fit onto the new lowest temporary skyline segment; hence the lowest segment is raised to the height of its lowest neighbour



(d) The lowest temporary skyline segment remains too narrow for the item and the lowest segment is raised to the height of its lowest neighbour



(e) The lowest temporary skyline segment is wide enough to accommodate the item



(f) The skyline is updated to reflect the position of the new item

Figure 5.5: *The process used to pack an item during execution of the BL algorithm. A copy is made of the skyline (represented by the dotted line) and this copy is modified until the lowest segment is wide enough for the item to be packed onto it.*

of the strip; hence $\mathcal{I}_5$ is packed on top of $\mathcal{I}_{11}$, against the left-hand boundary of the strip. There are now four skyline segments.

Item $\mathcal{I}_{10}$ is the next item in the list and is wider than the lowest skyline segment, but wider than the combined width of the two lowest segments. However, it has the same width as the combined width of the three lowest linked segments and is placed adjacent to $\mathcal{I}_5$. This results in a skyline reduced to two segments, the segment above $\mathcal{I}_5$ and the segment above $\mathcal{I}_{10}$. The next item in the list, $\mathcal{I}_3$, is placed in the lowest position, on top of $\mathcal{I}_{10}$. The space between $\mathcal{I}_3$ and the right-hand strip boundary is too narrow for $\mathcal{I}_4$, which is packed onto $\mathcal{I}_5$. The same applies for $\mathcal{I}_6$, which is also wider than the space between $\mathcal{I}_4$ and $\mathcal{I}_3$, resulting in its placement on top of $\mathcal{I}_4$. The same arguments apply for the placement of $\mathcal{I}_9$ on top of $\mathcal{I}_6$. Item $\mathcal{I}_{12}$ is too wide to

Procedure 5.3.1 FINDLOWESTPOINT $(\mathcal{I}, i, \mathcal{S}, s, \texttt{Height})$

```
1: make a copy K of the skyline S, Found ← False
2: while not Found do
3:    let s be the index of the skyline segment with the lowest height
4:    if w(K_s) ≥ w(I_i) then
5:       Height ← h(K_s), Found ← True
6:    else
7:       let p be the index of the left-hand skyline segment
8:       let n be the index of the right-hand skyline segment
9:       if h(K_p) ≤ h(K_n) then
10:         w(K_p) ← w(K_p) + w(K_s), remove index s from the skyline K
11:      else
12:         w(K_s) ← w(K_s) + w(K_n), remove index n from the skyline K
13:      end if
14:   end if
15: end while
16: discard the temporary skyline K
```

be packed on the right-hand side of $\mathcal{I}_3$, but the top edge of $\mathcal{I}_3$ is lower than the top edge of $\mathcal{I}_9$ and the space between $\mathcal{I}_4$, $\mathcal{I}_6$ and $\mathcal{I}_9$ is narrower than $\mathcal{I}_{12}$, resulting in its packing on top of $\mathcal{I}_3$, but against the right-hand edge of $\mathcal{I}_9$. The skyline now consists of a segment spanning the top edges of $\mathcal{I}_9$ and $\mathcal{I}_{12}$, a segment spanning the part of the top edge of $\mathcal{I}_3$ that is not below $\mathcal{I}_{12}$ and a segment spanning the part of the top edge of $\mathcal{I}_{10}$ that is not below $\mathcal{I}_3$.

The lowest part of the skyline on which $\mathcal{I}_7$ fits is that part above $\mathcal{I}_3$. Item $\mathcal{I}_1$ finds its best position on top of $\mathcal{I}_9$ against the left-hand strip boundary. Item $\mathcal{I}_8$ is at its lowest position adjacent to the left-hand edge of $\mathcal{I}_1$. The final item, $\mathcal{I}_2$, is narrow enough to be packed onto the lowest skyline segment, the segment above $\mathcal{I}_{10}$ and to the right of $\mathcal{I}_3$. The final strip height is 40 and a graphical representation of the packing may be found in Figure 5.1(c).

Known Performance Bounds

Baker *et al.* [6] established the worst-case performance bound

$$\texttt{BL}(\mathcal{L}) \leq 3\,\texttt{OPT}(\mathcal{L}),$$

where $\texttt{BL}(L)$ is the packing height of the BL algorithm for a list of items $\mathcal{L}$, and $\texttt{OPT}(L)$ is the optimal packing height for those items. This bound is the same as the worst-case performance bound for the SF algorithm, but worse than the bounds for Sleator's algorithm and the FFDH algorithm. If the items are all squares, then the worst-case performance bound may be improved to

$$\texttt{BL}(\mathcal{L}) \leq 2\,\texttt{OPT}(\mathcal{L}).$$

Worst-case Time Complexity

Copying the skyline in line 1 of Procedure 5.3.1 has a worst-case time complexity of $\mathcal{O}(n)$ as there are up to $n + 1$ segments in the skyline. Determining the lowest skyline segment in line 3 also has a worst-case time complexity of $\mathcal{O}(n)$, because each segment's height is

investigated. The contents of the *if*-statement spanning lines 4–14 has constant time complexity. The deletion of a skyline segment has a constant time complexity due the use of linked lists. Finding an appropriate position for the item may take $\mathcal{O}(n)$ computations in the *while*-loop spanning lines 2–15. Therefore, the overall time complexity of Procedure 5.3.1 is $\mathcal{O}(n)+\mathcal{O}(n)\times(\mathcal{O}(n)+\mathcal{O}(1))=\mathcal{O}\big(n^2\big)$.

The sorting in Algorithm 5.3 may be performed by the merge-sort algorithm, which has a time complexity of $\mathcal{O}(n\log n)$. The skyline initialisation in line 2 has a time complexity of $\mathcal{O}(n)$, as there may be $n+1$ skyline segments. The *for*-loop spanning lines 3–7 contains a call to Procedure 5.3.1 (which has a time complexity of $\mathcal{O}\big(n^2\big)$), an attempt to move the item further left and an update of the skyline. The attempt to move the item further left has a time complexity of $\mathcal{O}(n)$, as the location of the right-hand edge of every item packed before $\mathcal{I}_i$ is compared to the left-hand boundary of $\mathcal{I}_i$. If there is no item that has a right-hand edge coinciding with the left-hand edge of $\mathcal{I}_i$, then $\mathcal{I}_i$ may be moved left until its left-hand edge does coincide with the right-hand edge of another item or the left-hand boundary of the strip. Updating the skyline has a worst-case time complexity of $\mathcal{O}(n)$ as either a number of skyline segments are deleted from the skyline (if the item is longer than any skyline segment), or an index value must be found for the additional segment generated by the item that has been packed. Therefore, the BL algorithm has a time complexity of $\mathcal{O}(n\log n)+\mathcal{O}(n)+\mathcal{O}(n)\times\big(\mathcal{O}\big(n^2\big)+\mathcal{O}(n)+\mathcal{O}(n)\big)=\mathcal{O}\big(n^3\big)$. This corresponds to the claim by Chazelle [25, p. 697] that the BL algorithm has a naïve $\mathcal{O}\big(n^4\big)$, or at best a $\mathcal{O}\big(n^3\big)$ worst-case time complexity.

Practical Considerations

Due to the sequential approach of the algorithm (the items are packed in the order in which they are sorted), there is no need to use a linked list representation of the items. However, the use of linked lists is useful for the representation of the skyline since skyline segments may thus be added or deleted during the packing procedure. The skyline is represented by a sextuple with attributes representing the vertical and horizontal coordinates, the width of the skyline segment, the index value of the previous segment, the index value of the next skyline segment and the status of the segment. The status component is used to determine whether the index in the array is active in representing the skyline. In this manner it is easy to determine which indices in the array have to be inspected when searching for a packing location. The use of linked lists allows for the addition and deletion of segments to the skyline. This may happen when a skyline is updated after an item is packed. If an item `i` is packed onto a skyline segment `o` and the item is not as wide as the segment, then an addition must be made to the skyline. Once an index value `n` has been found in the skyline array that is not in use, the procedure in Figure 5.6 may be used to add the segment to the skyline and modify the skyline accordingly.

It is important to ensure that the skyline is reflected accurately in the array of sextuples. If an item is packed onto a skyline segment, it is not sufficient to simply raise the skyline segment. If the new height of the skyline segment has the same height as any of its neighbours, then the segments should be combined such that all adjacent segments of the same height are represented by one segment in the array. This prevents the algorithm from packing a narrow item into a narrow low segment, when the adjacent segments are the same height and could have accommodated a wider item. Therefore, after every skyline modification, the left-hand (see Figure 5.7) and perhaps right-hand neighbours (see Figure 5.8) of the changing skyline segment are inspected to determine whether their heights are the same. The right-hand side only needs to be inspected if the right-hand edge of the item coincides with the right-hand side of a skyline segment.

```
S(n).active ← True
S(n).Y ← S(o).Y
S(n).X ← S(o).X + I(i).W
S(n).W ← S(o).W − I(i).W
S(n).nxt ← S(o).nxt
S(n).prv ← o
if S(n).nxt > −1 then
    S(S(n).nxt).prv ← o
end if
S(o).W ← I(i).W
S(o).Y ← S(c).Y + I(i).H
S(o).nxt ← n
```

Figure 5.6: *Adding to the skyline during execution of the BL algorithm when an item has been packed in the left-hand corner of a skyline segment.*

```
if S(i).prv > −1 then
    p ← S(i).prv
    if S(p).Y = S(i).Y then
        S(p).W ← S(p).W + S(i).W
        S(p).nxt ← S(i).nxt
        S(p).active ← false
        if S(i).nxt > −1 then
            S(S(i).nxt).prv ← p
        end if
    end if
end if
```

Figure 5.7: *Updating the skyline on the left-hand side during execution of the BL algorithm.*

The packing of items is complicated by the possibility of overhangs (see Figure 5.9) occurring during the packing process. Baker *et al.* [6] do not deal with this specific case in their paper. However, they do state that the item "is first placed into the lowest possible location, and then it is left-justified at this vertical position" [6, p. 847]. Although the packing of items into the left-hand corner will generally yield a left-justified packing, the possibility of overhangs requires an additional step attempting to move the item further left. In order to determine how far to the left an item may be moved, the location of all $i-1$ previously packed items have to be inspected for possible overlapping. This is achieved by the procedure shown in Figure 5.10.

If the item may move further toward the left-hand side of the strip, this must be taken into account during the update of the skyline. If the item has moved further left than the item is long, then the skyline requires no updating. The item now finds itself completely under the overhang and hence under an existing skyline segment. However, in the case where packing the item causes a change to the skyline and the difference between the item width and the move space is greater than any single skyline segment over which it appears, some segments of the skyline may be deleted, and some may require modification. Once the leftmost skyline segment s has been identified where the item i will change the skyline, the procedure in Figure 5.11 may be used to alter the skyline appropriately.

```
if S(i).nxt > −1 then
    n ← S(i).nxt
    if S(n).Y = S(i).Y then
        S(i).W ← S(i).W + S(n).W
        S(i).nxt ← S(n).nxt
        S(n).active ← false
        if S(n).nxt > −1 then
            S(S(n).nxt).prv ← i
        end if
    end if
end if
```

Figure 5.8: *Updating the skyline on the right-hand side during execution of the BL algorithm.*

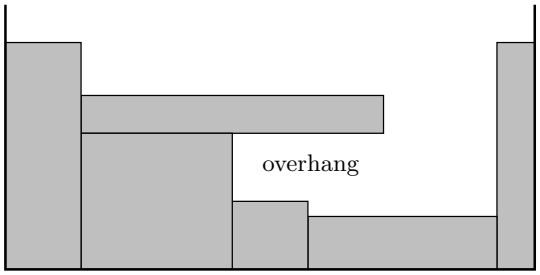


Figure 5.9: *An illustration of an overhang.*

5.1.4 Golan's Split Algorithm

In 1981 Golan [62] published the *split* (SP) algorithm. In this algorithm items are first ordered according to decreasing width. The strip is segmented into various regions R_j, depending on which items have been packed. A list $\mathcal{J}$ contains the regions that contain an item $\mathcal{Z}_j$ (the last item packed into that region) that may accommodate the next item in the list on its right-hand side. If there are no such regions, the region with the lowest height is chosen and the item is packed above the item already in it. The heights of the regions are stored in variables a_j and b_j, where b_j is the height from the bottom of the strip to the bottom edge of $\mathcal{Z}_j$ and $a_j = b_j + h(\mathcal{Z}_j)$. The bottom edge of region R_j is raised to the height of the bottom edge of the recently packed item and the region below that is relabelled M_j (the inactive regions of the strip). If $\mathcal{J}$ is not an empty set, the item is packed in the lowest region in $\mathcal{J}$ and the packing results in the splitting of the region R_j into two further regions R_{j+1} and R_{j+2}. This process continues until all items have been packed. The result is a guillotineable packing and a pseudocode listing of the SP algorithm may be found in Algorithm 5.4.

Worked Example

By sorting the items in Table 3.1 in order of decreasing width, the list $\mathcal{I} = \{\mathcal{I}_{13}, \mathcal{I}_{11}, \mathcal{I}_5, \mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_4, \mathcal{I}_6, \mathcal{I}_9, \mathcal{I}_{12}, \mathcal{I}_7, \mathcal{I}_1, \mathcal{I}_8, \mathcal{I}_2\}$ results. The algorithm begins with only one region; R_1, the entire strip. There are no regions in which $\mathcal{I}_{13}$ may be packed next to another item; hence $\mathcal{J} = \emptyset$. Therefore, item $\mathcal{I}_{13}$ is packed in the bottom left corner of the lowest region R_1. Now that $\mathcal{I}_{13}$ is packed into R_1, $a_1 = 5$ and $\mathcal{Z}_1 = \mathcal{I}_{13}$, but the set $\mathcal{J}$ of regions in which the current item $\mathcal{I}_{11}$

```
if i = 0 then
    MoveSpace ← 0
    Exit
end if
MoveSpace ← W
for j = 1 to i
    if I(j).X + I(j).W ≤ I(i).X and I(j).Y < I(i).Y + I(i).H
            and I(j).Y + I(j).H > I(i).Y then
        if MoveSpace > I(i).X − (I(j).X + I(j).W) then
            Found ← True
            MoveSpace ← I(i).X − (I(j).X + I(j).W)
            if MoveSpace = 0 then
                Exit
            end if
        end if
    end if
next
```

Figure 5.10: *Determining how far left an item may be moved from its current position during execution of the BL algorithm.*

Algorithm 5.4 Split (SP) algorithm

Input: A list $\mathcal{I}$ of n items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items and the strip width W.

Output: A packing of the items in $\mathcal{I}$ into a strip of width W.

1: sort the items according to decreasing width
2: initialise the first region R_1, $r \leftarrow 1$, $a_1 \leftarrow 0$, $b_1 \leftarrow 0$
3: **for** $i = 1$ **to** n **do**
4: let $\mathcal{J} = \{R_j \mid w(R_j) \geq w(\mathcal{Z}_j) + w(\mathcal{I}_i)\}$
5: **if** $\mathcal{J} \neq \emptyset$ **then**
6: determine R_ℓ, such that $b_\ell \leq b_j$ for all $R_j \in \mathcal{J}$
7: $r \leftarrow r + 1$, $b_r \leftarrow b_\ell$, $a_r \leftarrow b_\ell + h(\mathcal{I}_i)$, $w(R_r) \leftarrow w(R_\ell) - w(\mathcal{I}_i)$
8: $w(R_\ell) \leftarrow w(\mathcal{Z}_\ell)$, $b_\ell \leftarrow a_\ell$, $\mathcal{Z}_\ell \leftarrow \mathcal{I}_i$
9: **else**
10: determine R_ℓ, such that $a_\ell \leq a_j$ for all $1 \leq j \leq r$
11: $b_\ell \leftarrow a_\ell$, $a_\ell \leftarrow a_\ell + h(\mathcal{I}_i)$, $\mathcal{Z}_\ell \leftarrow \mathcal{I}_i$
12: **end if**
13: **end for**

fits next to the last item packed into it remains empty. Therefore, $\mathcal{I}_{11}$ is packed into the region with the lowest a value, namely region R_1, and $a_1 = 14$ and $b_1 = 5$. The same procedure is performed for $\mathcal{I}_5$ with similar results; it is packed onto $\mathcal{I}_{11}$ in R_1, resulting in the values $b_1 = 14$, $a_1 = 23$ and $\mathcal{Z}_1 = \mathcal{I}_{11}$.

There is still only one region, but this is added to the set $\mathcal{J}$, because $\mathcal{I}_{10}$ has the same width as the space between $\mathcal{I}_5$ and the boundary of the strip. Item $\mathcal{I}_{10}$ is placed into the space which results in the splitting of the region into two regions. That part of the strip below the top edge of $\mathcal{I}_{11}$ is used to generate the region M_1 and two new regions are generated: R_2 is the region

```
RemainingW ← I(i).W − MoveSpace − S(s).W
n ← s
do while RemainingW ≥ 0 and n <> −1
    n ← S(n).nxt
    if n > −1 then
        RemainingW ← RemainingW − S(n).W
        if RemainingW ≥ 0 then
            S(n).active ← False
        end if
    end if
loop
if n > −1 then
    S(n).prv ← s
    S(n).X ← S(n).X + S(n).W + RemainingW
    S(n).W ← −1 × RemainingW
end if
S(s).Y ← I(i).Y + I(i).H
S(s).nxt ← n
S(s).W ← I(i).W − MoveSpace
```

Figure 5.11: *Modifying the skyline after an item has been packed, such that the difference between its width and any movement to the left is greater than a number of low skyline segments.*

above $\mathcal{I}_5$ and R_3 is the region above $\mathcal{I}_{10}$. Now $b_2 = b_3 = 14$, $a_2 = 23$, $a_3 = 21$, $\mathcal{Z}_2 = \mathcal{I}_5$ and $\mathcal{Z}_3 = \mathcal{I}_{10}$. Neither region has any space to the right of the last item packed into it, with the result that $\mathcal{J} = \emptyset$. Therefore, $\mathcal{I}_3$ is packed into the region with the lowest a value, namely R_3. Now $b_3 = 21$, $a_3 = 27$ and $\mathcal{Z}_3 = \mathcal{I}_3$, while the corresponding values for R_2 remain unchanged.

Continuing in this manner, $\mathcal{I}_4$, $\mathcal{I}_6$ and $\mathcal{I}_9$ are packed into R_2 and $\mathcal{I}_{12}$ is packed into R_3, resulting in the values $b_2 = 26$, $a_2 = a_3 = 29$, $b_3 = 27$, $\mathcal{Z}_2 = \mathcal{I}_9$ and $\mathcal{Z}_3 = \mathcal{I}_{12}$. The item $\mathcal{I}_7$ has a width equal to the space remaining to the left of $\mathcal{I}_9$ in R_2, which leads to the nonempty set $\mathcal{J} = \{R_2\}$. Therefore, item $\mathcal{I}_7$ is packed adjacent to $\mathcal{I}_9$ and the region R_2 is split into two regions: R_4 and R_5, with values $b_4 = b_5 = 26$, $a_4 = a_5 = 29$, $\mathcal{Z}_4 = \mathcal{I}_9$ and $\mathcal{Z}_5 = \mathcal{I}_7$. The area of R_2 below items $\mathcal{I}_9$ and $\mathcal{I}_7$ becomes the area M_2. Item $\mathcal{I}_1$ does not fit into the space between $\mathcal{I}_{12}$ and the right-hand boundary of R_3, resulting in its location[1] in R_4. Item $\mathcal{I}_8$ is packed onto $\mathcal{I}_7$, while $\mathcal{I}_2$ fits adjacent to $\mathcal{I}_1$ in R_4 and is packed there, resulting in the packing shown in Figure 5.12(a).

Known Performance Bounds

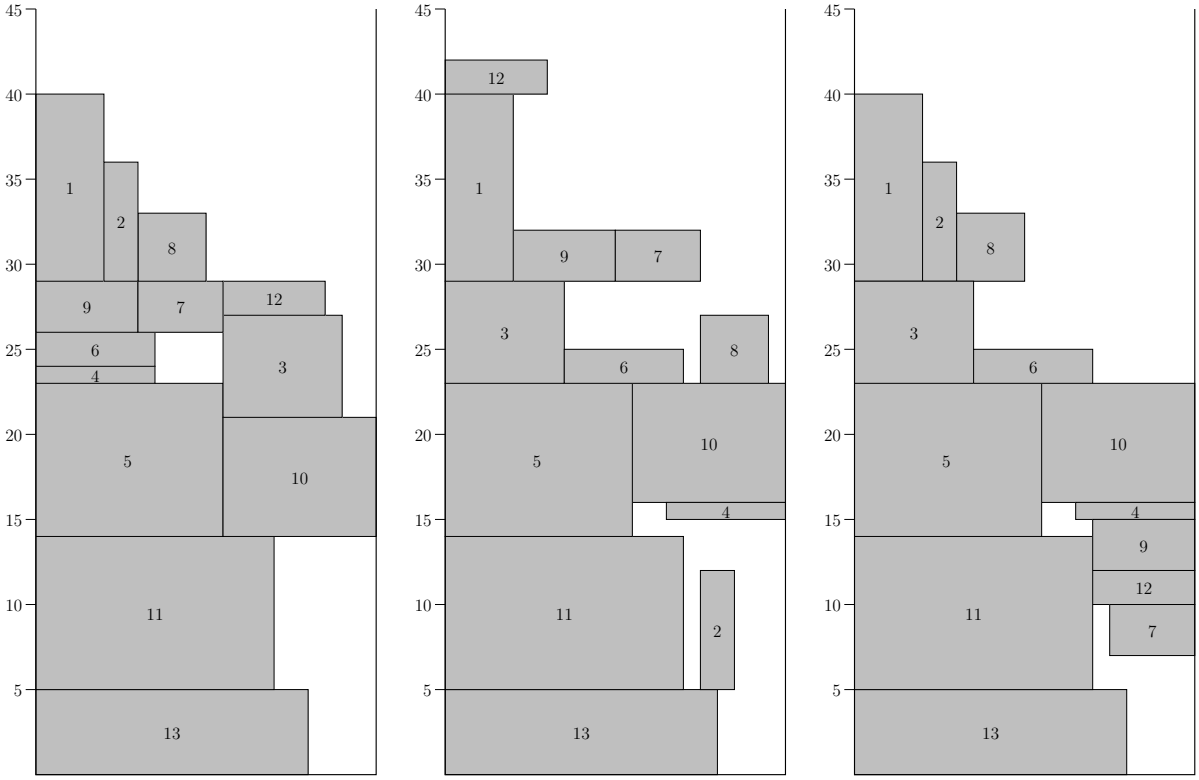
Golan [62] established the asymptotic performance bound

$$\mathtt{SP}(\mathcal{L}) \leq 2\,\mathtt{OPT}(\mathcal{L}) + 1$$

for a list $\mathcal{L}$ containing items with a maximum width and height of 1, where $\mathtt{SP}(\mathcal{L})$ is the packing height of the SP algorithm for a list of items $\mathcal{L}$, and $\mathtt{OPT}(\mathcal{L})$ denotes the optimal packing height for those items. He also established the worst-case performance bound

$$\mathtt{SP}(\mathcal{L}) \leq 3\,\mathtt{OPT}(\mathcal{L})$$

[1] Golan [62, p. 572] remarks that ties in height may be resolved in any manner. In this example ties were resolved by packing the item into the leftmost region of that height.



(a) SP ($\mathtt{H} = 40$)

(b) M ($\mathtt{H} = 42$))

(c) UD ($\mathtt{H} = 40$)

Figure 5.12: *Results obtained when packing the items in Table 3.1 into a strip of width 20 using the known plane strip packing algorithms described in §5.1.4–§5.1.6. The resulting packing heights* H *are also shown.*

for the SP algorithm. If the list $\mathcal{L}$ contains only squares, the worst-case performance bound may be improved to

$$\mathtt{SP}_{\mathrm{S}}(\mathcal{L}) \leq 2\,\mathtt{OPT}(\mathcal{L}).$$

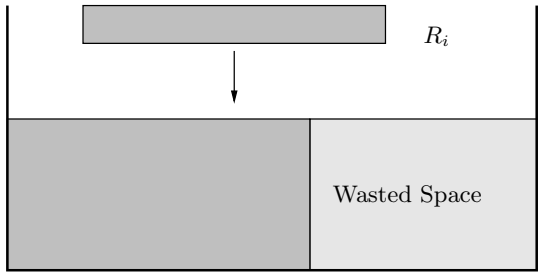
Worst-case Time Complexity

The sorting procedure in line 1 of Algorithm 5.4 may be performed by the merge-sort algorithm which has a worst-case time complexity of $\mathcal{O}(n \log n)$. The initialisation of the regions in line 2 has a constant time complexity. The search for regions that may be added to the list $\mathcal{J}$ in line 4 has a time complexity of $\mathcal{O}(n)$. So too does the subsequent search for the lowest appropriate packing region (see lines 6 and 10). All other operations within the *for*-loop spanning lines 3–13 have a constant time complexity. Therefore the algorithm has an overall worst-case time complexity of $\mathcal{O}(n \log n) + \mathcal{O}(1) + \mathcal{O}(n) \times (\mathcal{O}(n) + \max\{\mathcal{O}(n) + \mathcal{O}(1), \mathcal{O}(n) + \mathcal{O}(1)\}) = \mathcal{O}(n^2)$.

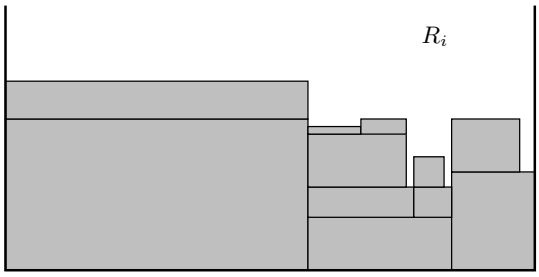
Algorithmic Variations and Practical Considerations

Golan's algorithm wastes some space when the set $\mathcal{J}$ is empty. In this case the algorithm packs an item onto the incumbent item $\mathcal{Z}_i$ in $\mathcal{R}_i$ and raises the value of b_i in order to allow items to be packed adjacent to the new item. A space as wide as the difference in widths of $\mathcal{Z}_i$ and $\mathcal{R}_i$ and as tall as the height of $\mathcal{Z}_i$ remains unused.

This space may be filled by means of a recursive subroutine that attempts to stack items that remain unpacked into the space. The first unpacked item that fits may be packed into the bottom-left corner of the space. The procedure may call itself with the remaining width and the same height. If all unpacked items are wider than the space that remains after an item is packed, then an attempt may be made to stack the remaining items. During the stacking procedure the algorithm should reject unpacked items that are either too wide or whose height would yield a stack of items taller than the height of the item next to which the items are being stacked. This concept is illustrated in Figure 5.13.



(a) A large space may remain empty if the SP algorithm is used to pack consecutive items with a combined width larger than $w(R_i)$



(b) The resulting SPm algorithms attempt to fill this space with smaller items before the next item is packed

Figure 5.13: *An illustration of the proposed modification to the SP algorithm. An attempt is made to pack smaller items adjacent to an item before another item is packed above it.*

In order to achieve the modification mentioned above, a subroutine such as the one listed in Procedure 5.4.1 is required. This subroutine is to be called after line 10 of Algorithm 5.4. The *for*-loop spanning lines 3–13 of Algorithm 5.4 should be changed to a *while*-loop that terminates when no unpacked items remain. In order to render the algorithm more efficient when items may be packed out of sequence, linked lists may again be used to represent the order of items. The resulting modified SP algorithm is labelled the SPmG algorithm, because the results are guaranteed to be guillotineable. This additional procedure increases the worst-case time complexity for the SPmG algorithm to $\mathcal{O}(n^4)$, because Procedure 5.4.1 has a worst-case time complexity of $\mathcal{O}(n^3)$. For every item that is packed, an attempt is made to pack the remaining items above it and next to it.

In an attempt to pack items more densely according to an SP-like algorithm, further changes may be made to the SPmG algorithm. Before any item is packed, the positions of all previously packed items may be noted so that the lowest possible location for the new item may be found. This "gravity" effect may render the solution non-guillotinable. The resulting algorithm is therefore named the SPmF algorithm. An attempt to lower the item is not made when an item is packed due to an empty set $\mathcal{J}$, because the item is packed such that its lower edge is coincident with the incumbent item $\mathcal{Z}_j$ in the region Rj. When the procedure was first programmed, the subroutine used a *for*-loop that inspected every item in the list in order to determine whether it was to be packed. If the item was packed, then the procedure would investigate whether the item had an influence on the item that was to be packed. However, it proves faster to maintain a list of packed items $\mathcal{A}$, noting the index value that was the last to be entered. A search of the entire list of items is then no longer required. Instead, only those items that are already packed are inspected, thereby increasing the speed of the algorithm when the few items are packed. The pseudocode required to perform this task is shown in Figure 5.14, increasing the worst-case time complexity of the entire procedure to $\mathcal{O}(n^5)$.

Procedure 5.4.1 FILLNOJ($\mathcal{I}, i, t, X, Y,$ `Width`, `Height`)

1: let t be the index value of the thinnest unpacked item
2: let $\mathcal{I}_j$ be the first unpacked item after $\mathcal{I}_i$
3: `PackedH` $\leftarrow 0$
4: **while** unpacked items remain **and** `Width` $\geq w(\mathcal{I}_t)$ **do**
5: **if** $h(\mathcal{I}_j) \leq$ `Height` $-$ `PackedH` **and** $w(\mathcal{I}_j) \leq$ `Width` **then**
6: pack $\mathcal{I}_j$ with coordinates X and $Y +$ `PackedH`
7: **if** no unpacked items remain **then**
8: **exit**
9: **end if**
10: **if** `Width` $- w(\mathcal{I}_j) \geq w(\mathcal{I}_t)$ **then**
11: **call** FILLNOJ $(\mathcal{I}, j, t, X + w(\mathcal{I}_j), Y +$ `PackedH`, `Width` $- w(\mathcal{I}_j)$, `Height` $-$ `PackedH`$)$
12: **end if**
13: **if** an item was packed to the right of $\mathcal{I}_j$ **then**
14: `Width` $\leftarrow w(\mathcal{I}_j)$
15: **end if**
16: `PackedH` $\leftarrow$ `PackedH` $+ h(\mathcal{I}_j)$
17: **end if**
18: let $\mathcal{I}_j$ be the next unpacked item
19: **end while**

```
MoveSpace ← Y
for j = 0 to NumPacked
    k ← A(j)
    if I(k).Y + I(k).H <= I(i).Y and I(k).X < I(i).X + I(i).W and
            I(k).X + I(k).W > I(i).X then
        if MoveSpace > I(i).Y − (I(k).Y + I(k).H) then
            MoveSpace ← I(i).Y − (I(k).Y + I(k).H)
            if MoveSpace = 0 then
                exit
            end if
        end if
    end if
next
```

Figure 5.14: *Determining how far items may be moved down during execution of the SPmF algorithm.*

5.1.5 Golan's Mixed-Algorithm

Golan [62] also designed the *mixed algorithm* (M algorithm). Consider a list of items $\mathcal{L}$, each with a maximum width and height one. Initially the items are sorted into five groups, namely

$$\mathcal{A} = \left\{ \mathcal{L}_i \;\middle|\; w(\mathcal{L}_i) > \frac{1}{2} \right\}, \quad \mathcal{B} = \left\{ \mathcal{L}_i \;\middle|\; \frac{1}{3} < w(\mathcal{L}_i) \leq \frac{1}{2} \right\}, \quad \mathcal{C} = \left\{ \mathcal{L}_i \;\middle|\; \frac{1}{4} < w(\mathcal{L}_i) \leq \frac{1}{3} \right\},$$

$$\mathcal{D}^1 = \left\{ \mathcal{L}_i \;\middle|\; \frac{5}{24} < w(\mathcal{L}_i) \leq \frac{1}{4} \right\} \quad \text{and} \quad \mathcal{D}^2 = \left\{ \mathcal{L}_i \;\middle|\; w(\mathcal{L}_i) \leq \frac{5}{24} \right\}.$$

All items in $\mathcal{A}$ are sorted by decreasing width. All items are stacked on top of one another until no unpacked items remain in $\mathcal{A}$. The height to which they are packed is labelled L_2. The

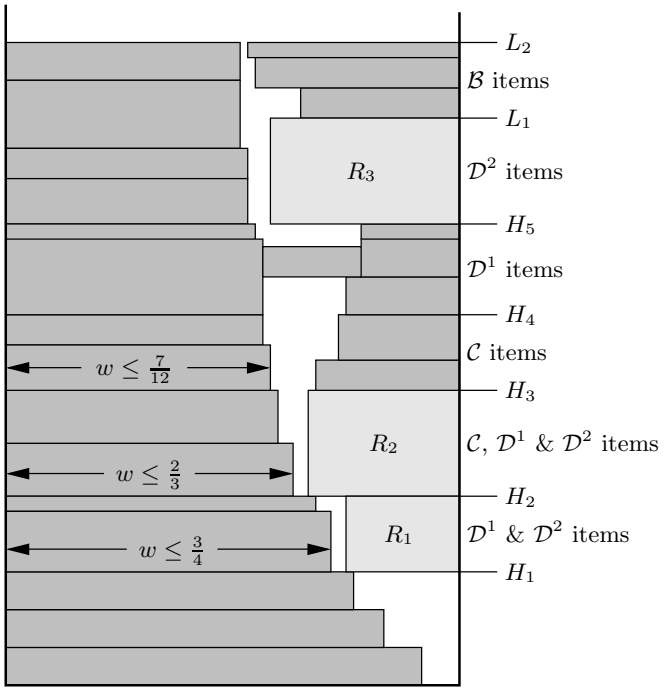


Figure 5.15: *An illustration of the regions packed by the M algorithm.*

items in $\mathcal{B}$ are sorted by decreasing width and an attempt is made to pack these items next to the items from $\mathcal{A}$. This is achieved by stacking the items from L_2 downwards so that the bottom edge of the last item in $\mathcal{B}$ that fits is at a height of L_1. The space below the packed items in $\mathcal{B}$ and to the right of the items in $\mathcal{A}$ is then split into further regions. The first region is between the heights of the top edges of the highest items in $\mathcal{A}$ of width greater than 3/4 (this height is labelled H_1) and the highest item of width greater than 2/3 (this height is labelled H_2). This region (labelled R_1) has a width of 1/4 and is filled with items in $\mathcal{D}^1$ and $\mathcal{D}^2$ using the FFDH algorithm (see §3.2.2). The height H_3 is defined as the minimum of L_1 and the height of the top edge of the highest item of width greater than 7/12. If H_2 and H_3 are distinct, then a region R_2 of width 1/3 is created between these heights in which items from $\mathcal{C}$, $\mathcal{D}^1$ and $\mathcal{D}^2$ are packed according to the FFDH algorithm. If H_3 is lower than L_1 then any unpacked items in $\mathcal{C}$ are stacked into this region, such that their right-hand edges are adjacent to the right-hand boundary of the strip. The height of the top edge of those items in $\mathcal{C}$ is labelled H_4. If there is sufficient space between H_4 and L_1 for any unpacked items from $\mathcal{D}^1$, then these are packed in that space by means of the bottom-up right-justified algorithm (an adaption of the BL algorithm presented in §5.1.3). The height of the highest point of these items is labelled H_5. If sufficient space remains between H_5 and L_1 for unpacked items in $\mathcal{D}^2$, then a region R_3 of width 5/12 is created into which items in $\mathcal{D}^2$ are packed by means of the FFDH algorithm. A graphical representation of such a packing may be found in Figure 5.15.

Any unpacked items from $\mathcal{B}$ are packed above the items in $\mathcal{A}$. The levels are rearranged so that any levels with width greater than 3/4 are below the others. The height of the top edge of the highest item from $\mathcal{B}$ is labelled L_4 and the vertical coordinate of the lowest level of width at most 3/4 is labelled L_3. The remaining space between L_3 and L_4, with a width of 1/4, is filled with any unpacked items by means of the FFDH algorithm. If any items from $\mathcal{C}$, $\mathcal{D}^1$ or $\mathcal{D}^2$ remain unpacked, they are packed above L_4 in an FFDH manner. A pseudocode listing of the M-algorithm may be found in Algorithm 5.5.

Worked Example

After partitioning the items in Table 3.1 into their respective sets and sorting those sets in the appropriate order, the lists $\mathcal{A} = \{\mathcal{I}_{13}, \mathcal{I}_{11}, \mathcal{I}_5\}$, $\mathcal{B} = \{\mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_4, \mathcal{I}_6\}$, $\mathcal{C} = \{\mathcal{I}_9, \mathcal{I}_{12}\}$, $\mathcal{D}^1 = \{\mathcal{I}_7\}$ and $\mathcal{D}^2 = \{\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_8\}$ result. The items in $\mathcal{A}$ are stacked onto one another, so that their left-

Algorithm 5.5 Mixed-algorithm

Input: A list $\mathcal{I}$ of n items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items and the strip width $\mathtt{W}$.

Output: A packing of the items in $\mathcal{I}$ into a strip of width $\mathtt{W}$.

place all items where $w(\mathcal{I}_i) > \mathtt{W}/2$ in $\mathcal{A}$, let n_A denote the number of items in $\mathcal{A}$
place all items where $\mathtt{W}/3 < w(\mathcal{I}_i) \leq \mathtt{W}/2$ in $\mathcal{B}$, let $n_B = |\mathcal{B}|$
place all items where $\mathtt{W}/4 < w(\mathcal{I}_i) \leq \mathtt{W}/3$ in $\mathcal{C}$, let $n_C = |\mathcal{C}|$
place all items where $\mathtt{W} \times 5/24 < w(\mathcal{I}_i) \leq \mathtt{W}/4$ in $\mathcal{D}^1$, let $n_{D1} = |\mathcal{D}^1|$
place all items where $w(\mathcal{I}_i) \leq \mathtt{W} \times 5/24$ in $\mathcal{D}^2$, let $n_{D2} = |\mathcal{D}^2|$
if $n_A > 0$ **then**
  sort all items in $\mathcal{A}$ according to decreasing width
  stack all items in $\mathcal{A}$ adjacent to the left-hand boundary of the strip
  let L_2 denote the height to which items in $\mathcal{A}$ have been packed
  stack items from $\mathcal{B}$ in a downward manner from L_2, update n_B
  let L_1 denote the height at which the bottom-most item from $\mathcal{B}$ is packed
  let $j = \max_{1\leq j\leq n_A}(0, j \mid w(\mathcal{A}_j) > \mathtt{W} \times 3/4)$
  set H_1 equal to the vertical coordinate of the top edge of $\mathcal{A}_j$, if $H_1 > L_1$ then $H_1 \leftarrow L_1$
  let $k = \max_{1\leq k\leq n_A}(0, k \mid w(\mathcal{A}_k) > \mathtt{W} \times 2/3)$
  set H_2 equal to the vertical coordinate of the top edge of $\mathcal{A}_k$, if $H_2 > L_1$ then $H_2 \leftarrow L_1$
  let $m = \max_{1\leq m\leq n_A}(m \mid w(\mathcal{A}_m) > \mathtt{W} \times 7/12)$
  set H_3 equal to the vertical coordinate of the top edge of $\mathcal{A}_m$, if $H_3 > L_1$ then $H_3 \leftarrow L_1$
  if $H_1 < H_2$ **and** $(n_{D1} > 0$ **or** $n_{D2} > 0)$ **then**
    $w(R_1) \leftarrow \mathtt{W}/4$, $h(R_1) \leftarrow H_2 - H_1$, locate R_1 as shown in Figure 5.15
    pack items from $\mathcal{D}^1$ and $\mathcal{D}^2$ into R_1 with the FFDH algorithm, update n_{D1} and n_{D2}
  end if
  if $H_2 < H_3$ **and** $(n_C > 0$ **or** $n_{D1} > 0$ **or** $n_{D2} > 0)$ **then**
    $w(R_2) \leftarrow \mathtt{W}/3$, $h(R_2) \leftarrow H_3 - H_2$, locate R_2 as shown in Figure 5.15
    pack items from $\mathcal{C}$, $\mathcal{D}^1$ and $\mathcal{D}^2$ into R_2 by means of the FFDH algorithm
    update n_C, n_{D1} and n_{D2}
  end if
  if $n_C > 0$ then stack unpacked items from $\mathcal{C}$ right-aligned above H_3, update n_C
  let $H_3 \leq H_4 \leq L_1$ be the height to which the items in $\mathcal{C}$ are stacked
  if $H_4 < L_1$ **and** $n_{D1} > 0$ **then**
    pack $\mathcal{D}^1$ items in the remaining space with the bottom-up right-justified algorithm
    update n_{D1}
  end if
  let $H_4 \leq H_5 \leq L_1$ be the height to which items are packed in the right-hand column
  if $H_5 < L_1$ **and** $n_{D2} > 0$ **then**
    $w(R_3) \leftarrow \mathtt{W} \times 5/12$, $h(R_2) \leftarrow L_1 - H_5$, locate R_3 as shown in Figure 5.15
    pack items from $\mathcal{D}^2$ into R_3 by means of the FFDH algorithm, update n_{D2}
  end if
end if{*continued...*}

Algorithm 5.5 (*continued...*) Mixed-algorithm

if $n_B > 0$ **then**
  pack any unpacked items in $\mathcal{B}$ beginning at L_2 by means of the FFDH algorithm
  let L_4 be the height of the topmost item in $\mathcal{B}$
  rearrange the levels such that those of width $> \mathtt{W} \times 3/4$ are below the others
  let L_3 be the height of the boundary between the two groups of levels
  create a region R_4 of width $\mathtt{W} \times 3/4$ between L_3 and L_4
  if $n_{D1} > 0$ **or** $n_{D2} > 0$ **then**
    pack the items into R_4, update n_{D1} and n_{D2}
  end if
end if
if $n_C > 0$ **or** $n_{D1} > 0$ **or** $n_{D2} > 0$ **then**
  pack the remaining items above L_4 with the FFDH algorithm
end if

hand edges coincide with the left-hand boundary of the strip. The resulting heights are $H_1 = 5$, $H_2 = H_3 = 14$ and $L_2 = 23$. An attempt is made to stack the items in $\mathcal{B}$ in a downward manner from L_2. Items $\mathcal{I}_{10}$ and $\mathcal{I}_4$ fit into the space between the items in $\mathcal{A}$ and the left-hand strip boundary and result in $L_1 = 15$, but $\mathcal{I}_3$ and $\mathcal{I}_6$ do not fit and hence remain unpacked.

Now a region R_1 of width $\mathtt{W}/4$ is created between H_1 and H_2, and is right-justified against the right-hand boundary of the strip. Of the items in $\mathcal{D}^1$ and $\mathcal{D}^2$ only $\mathcal{I}_2$ fits into R_1 and is packed there. The heights H_2 and H_3 are equal; hence region R_2 is not created. The heights $H_4 = H_3 = 14$ result and an attempt is made to pack items from $\mathcal{C}$ into the space between H_4 and L_1. None of the items fit into the space, nor do any of the items in $\mathcal{D}^1$ or $\mathcal{D}^2$.

The unpacked items in $\mathcal{B}$ are packed above L_2 by means of the FFDH algorithm. This means that $\mathcal{I}_3$ is packed against the left-hand strip boundary and $\mathcal{I}_6$ is packed adjacent to the right-hand edge of $\mathcal{I}_3$. There is only one level of items from $\mathcal{B}$ and the width of the level is less than three quarters of the strip width; hence $L_3 = L_2 = 23$ and $L_4 = 29$. Therefore, a region R_4 of width $\mathtt{W}/4$ is created between L_3 and L_4 and an attempt is made to pack some of the remaining items into that region. Only $\mathcal{I}_8$ fits and is packed there. The remaining items are packed into the strip above L_4 by means of the FFDH algorithm, resulting in the packing shown in Figure 5.12(b).

Known Performance Bounds

Golan [62] established the asymptotic performance bound

$$\mathtt{M}(\mathcal{L}) \leq \frac{4}{3}\mathtt{OPT}(\mathcal{L}) + 7\frac{1}{18}$$

for the M-algorithm (packing a list with a maximum item width and height of 1), where $\mathtt{M}(\mathcal{L})$ denotes the packing height of the M algorithm for a list of items $\mathcal{L}$, and $\mathtt{OPT}(\mathcal{L})$ denotes the optimal packing height for those items.

Worst-case time complexity

The assignment of an item to its respective set, spanning lines 1–5 in Algorithm 5.5, has a time complexity of $\mathcal{O}(n)$. If there are items wider than half the strip width, then these items are sorted in line 7. This step has a time complexity of $\mathcal{O}(n \log n)$ if an algorithm such as merge-sort is used. The stacking of the items in line 8 has a linear time complexity. In order for the items in $\mathcal{B}$ to be stacked in a downward manner, care must be taken that no items overlap. Hence, for each attempt at packing an item from $\mathcal{B}$, the list of items in $\mathcal{A}$ is searched for an item that may overlap with the item from $\mathcal{B}$. Therefore the step in line 10 has a quadratic time complexity in the worst case. Determining the heights of the parameters L and H in lines 9 and 11–17 may be performed during the packing procedures and therefore only contribute operations of constant time complexity to the algorithm. The creation of regions in the *if*-statement spanning lines 18–21 has a constant time complexity, while the FFDH algorithm used to pack the items in $\mathcal{D}^1$ and $\mathcal{D}^2$ in line 20 has a worst-case time complexity of $\mathcal{O}(n^2)$. The same applies to the *if*-statement spanning lines 22–26. The stacking of items from $\mathcal{C}$ between H_3 and L_1 in line 27 has a linear time complexity. Line 30 contains a call to the bottom-up right-justified algorithm, which has been shown in §5.1.3 to have a worst-case time complexity of $\mathcal{O}(n^3)$. Region R_3 is filled by means of the FFDH algorithm in line 36. This step has a worst-case time complexity of $\mathcal{O}(n^2)$. If any unpacked items remain in $\mathcal{B}$, then they are packed above L_2 according to the FFDH algorithm in line 40 (which has a $\mathcal{O}(n^2)$ worst-case time complexity). Once all the items in $\mathcal{B}$ have been packed, the levels are rearranged in line 42 by means of a procedure that has a linear time complexity. If there are levels of items from $\mathcal{B}$ that have a width of at most three quarters of the width of the strip, region R_4 is created (constant time complexity) and is filled with any remaining items from $\mathcal{D}^1$ and $\mathcal{D}^2$ by means of the FFDH algorithm, which has a quadratic time complexity. Finally, all remaining items are packed above L_4 by means of the FFDH algorithm. Thus, the overall time complexity of the M-algorithm is $\mathcal{O}(n^3)$.

Practical Considerations

The items in the set $\mathcal{A}$ are the only items that are not packed into a restricted area throughout execution of the algorithm. If there are items in $\mathcal{A}$, then the items in $\mathcal{B}$, $\mathcal{C}$, $\mathcal{D}^1$ and $\mathcal{D}^2$ may be packed into restricted regions. Therefore, items may not be packed in the order in which they appear once sorted, as some items may not fit into one of these regions. The use of a linked list implementation allows for fast removal of items from the various sets.

The fact that items from up to three groups may be packed into a single region (R_2, for example) means that the FFDH algorithm should be modified to accept items from three groups. This reduces the number of computationally expensive generations of new item lists of combined sets of items that must be re-sorted into their respective sets when the FFDH algorithm has completed filling a region. The first part of the algorithm sorts the three sets of items according to decreasing height and places the tallest item in the three sets that fits into the region into the bottom-left corner of the region. In the second part of the algorithm, the items are packed according to a procedure of the form illustrated in Procedure 5.5.1.

The purpose of Procedure 5.5.1 is to determine which item should be packed into the region. There are two conditions according to which a list of items is no longer considered for packing. The first occurs when there are no unpacked items in the list (when the variables `NumC`, `NumD1` and `NumD2` are zero). The second condition occurs when all items in a list have been considered for packing, but some items have not been packed because they do not fit into the region. This is indicated by the indices i, j and k being equal to -1. They will be -1 because of the way the

Procedure 5.5.1 FFDH3 (Part 2)

```
1: Packed ← False, NumLevels ← 1
2: L_1 ← W − width of the first item packed (in Part 1)
3: RemainingH ← height of the region − height of the first item packed (in Part 1)
4: while not Packed and (i or j or k <> −1) and (NumC or NumD1 or NumD2 > 0) do
5:   if (NumC < 1 or i < 0) and (NumD1 < 1 or j < 0) and (NumD2 < 1 and k < 0) then
6:     exit
7:   else if (NumD1 < 1 or j < 0) and (NumD2 < 1 or k < 0) then
8:     call PackMain (I, L, C, R, i, RemainingH, NumC, NumLevels, FirstC)
9:   else if (NumC < 1 or i < 0) and (NumD2 < 1 or k < 0) then
10:    call PackMain (I, L, D^1, R, j, RemainingH, NumD1, NumLevels, FirstD1)
11:  else if (NumC < 1 or i < 0) and (NumD1 < 1 or j < 0) then
12:    call PackMain (I, L, D^2, R, k, RemainingH, NumD2, NumLevels, FirstD2)
13:  else if (NumD2 < 1 or k < 0) and i > −1 and j > −1 then
14:    if h(C_i) < h(D^1_j) then
15:      call PackMain (I, L, D^1, R, j, RemainingH, NumD1, NumLevels, FirstD1)
16:    else
17:      call PackMain (I, L, C, R, i, RemainingH, NumC, NumLevels, FirstC)
18:    end if
19:  else if (NumD1 < 1 or j < 0) and i > −1 and k > −1 then
20:    if h(C_i) < h(D^2_k) then
21:      call PackMain (I, L, D^2, R, k, RemainingH, NumD2, NumLevels, FirstD2)
22:    else
23:      call PackMain (I, L, C, R, i, RemainingH, NumC, NumLevels, FirstC)
24:    end if
25:  else if (NumC < 1 or i < 0) and j > −1 and k > −1 then
26:    if h(D^1_j) < h(D^2_k) then
27:      call PackMain (I, L, D^2, R, k, RemainingH, NumD2, NumLevels, FirstD2)
28:    else
29:      call PackMain (I, L, D^1, R, j, RemainingH, NumD1, NumLevels, FirstD1)
30:    end if
31:  else
32:    if h(C_i) ≥ h(D^1_j) and h(C_i) ≥ h(D^2_k) then
33:      call PackMain (I, L, C, R, i, RemainingH, NumC, NumLevels, FirstC)
34:    else if h(D^1_j) > h(C_i) and h(D^1_j) ≥ h(D^2_k) then
35:      call PackMain (I, L, D^1, R, j, RemainingH, NumD1, NumLevels, FirstD1)
36:    else if h(D^2_k) > h(C_i) and h(D^2_k) > h(D^1_j) then
37:      call PackMain (I, L, D^2, R, k, RemainingH, NumD2, NumLevels, FirstD2)
38:    end if
39:  end if
40: end while
```

linked lists are created. The ends of the list are noted by values of -1 in order to prevent looping, as discussed in the *Practical Considerations* section of §3.4. The many *if*-statements ensure that no index values violate the bounds of the array. A pseudocode listing of the procedure labelled PackMain may be found in Procedure 5.5.2.

If there are unpacked items in $\mathcal{B}$ that are to be packed above L_2, they are packed according to

Procedure 5.5.2 PACKMAIN($\mathcal{I}, \mathcal{L}, \mathcal{D}, R, i$, `RemainingH`, `NumD`, `NumLevels`, `FirstD`)

```
1: j ← 1, Packed ← False
2: while not Packed do
3:   if w(D_i) ≤ w(L_j) then
4:     pack D_i into level j, Packed ← True
5:     NumD ← NumD − 1, w(L_j) ← w(L_j) − w(D_i)
6:   else if w(R) < w(D_i) then
7:     Packed ← True
8:   else
9:     j ← j + 1
10:    if j > NumLevels and RemainingH ≤ w(D_i) then
11:      NumLevels ← NumLevels + 1, RemainingH ← RemainingH − h(D_i)
12:      w(L_j) ← w(R) − w(D_i), h(L_j) ← h(D_i), Packed ← True, NumD ← NumD − 1
13:    else if j > NumLevels and RemainingH > w(D_i) then
14:      Packed ← True
15:    end if
16:  end if
17: end while
18: set i equal to the index of the next unpacked item in D
```

the FFDH algorithm. In order for this to be possible, the array representing the set $\mathcal{B}$ must be sorted according to decreasing height. However, in such a case the linked list will no longer be valid after the sorting procedure; hence a procedure is required to repair the linked list. The level into which an item is packed is saved in the `.lvl` characteristic of the decuple that represents an item. Therefore, it is possible to rearrange the levels according to a procedure such as the one in Figure 5.16.

In addition to rearranging the levels of the items in $\mathcal{B}$ that remain unpacked, the procedure determines the heights L_3 and L_4. This is important for the penultimate step, during which any remaining unpacked items are packed into the region R_4.

5.1.6 The Up-Down Algorithm

In 1981 Baker *et al.* [5] described the *Up-Down* (UD) algorithm for two-dimensional strip packing. It partitions the strip into five regions $R_1, \ldots, R_5$ of heights $h_1, \ldots, h_5$. Items with a width in the range $(1/(i+1), 1/i]$ are assigned into their respective sets $\mathcal{L}^i$ where $1 \leq i \leq 4$, and packed into the regions R_1 to R_4 using a combination of simple stacking and the BL algorithm. Once all items allocated to a region R_i have been packed, items of $\mathcal{L}^j$, $i+1 \leq j \leq 4$ are packed in a column from h_i towards h_{i-1} in order of decreasing width before they are packed into their region by means of the BL algorithm. Once this part of the algorithm is completed, the remaining items in $\mathcal{L}^5$ (the set of items of width no greater than $1/5$) are packed into the spaces between the items on the right and the items on the left, using a generalised version of the NFDH algorithm. The *generalised next-fit decreasing height* (GNFDH) algorithm allows items to be packed into a region with left and right boundaries that may change with height. Any remaining items in $\mathcal{L}^5$ are packed into R_5 using the NFDH algorithm. The resulting packing is not guaranteed to be guillotineable and a pseudocode listing of the UD algorithm may be found in Algorithm 5.6.

```
Height ← 0
for i = 1 to NumLevels
    if L(i).W > 3 * W/4 then
        LY(i) ← Height
        Height ← Height + L(i).H
    end if
end for
L3 ← Height
for i = 1 to NumLevels
    if L(i).W ≤ 3 * W/4 then
        LY(i) ← Height
        Height ← Height + L(i).H
    end if
end for
L4 ← Height
i ← FirstB
do while i <> −1
    B(i).Y ← LY(B(i).lvl)
    NumB ← NumB − 1
    i ← B(i).nxt
end while
```

Figure 5.16: *Rearranging levels and items during execution of the M algorithm.*

Worked Example

After partitioning the items in Table 3.1 into their respective sets and sorting those sets appropriately, the lists $\mathcal{L}^1 = \{\mathcal{I}_{13}, \mathcal{I}_{11}, \mathcal{I}_5\}$, $\mathcal{L}^2 = \{\mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_4, \mathcal{I}_6\}$, $\mathcal{L}^3 = \{\mathcal{I}_9, \mathcal{I}_{12}\}$, $\mathcal{L}^4 = \{\mathcal{I}_7\}$ and $\mathcal{L}^5 = \{\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_8\}$ result. The algorithm begins by packing the items in $\mathcal{L}^1$ into the strip in a BL manner, *i.e.* $\mathcal{I}_{13}$ is packed into the bottom-left corner, and $\mathcal{I}_{11}$ and $\mathcal{I}_5$ are packed onto it, their left-hand edges coinciding with the left-hand strip boundary. An attempt is now made to stack items from the other sets (excluding $\mathcal{L}^5$) down from the ceiling at a height of $H_1 = 23$. Items $\mathcal{I}_{10}$ and $\mathcal{I}_4$ from $\mathcal{L}^2$, $\mathcal{I}_9$ and $\mathcal{I}_{12}$ from $\mathcal{L}^3$ and $\mathcal{I}_7$ from $\mathcal{L}^4$ all fit into the space between the items in $\mathcal{L}^1$ and the right-hand boundary of the strip. Items $\mathcal{I}_3$ and $\mathcal{I}_6$ are packed into a new level by means of the BL algorithm to a height of $H_2 = 29$. No items remain in $\mathcal{L}^3$ and $\mathcal{L}^4$; hence no items are packed from the ceiling. The only items remaining are in the list $\mathcal{L}^5$ and an attempt is made to pack them into the free space in the two existing levels. Item $\mathcal{I}_1$ does not fit into any of the spaces; hence the items are all packed above H_2 by means of the NFDH algorithm, yielding the packing in Figure 5.12(c).

Known Performance Bounds

Baker *et al.* [5] established the performance bound

$$\texttt{UD}(\mathcal{L}) \leq \frac{5}{4}\,\texttt{OPT}(\mathcal{L}) + \frac{53}{8}H$$

for the UD algorithm, where $\texttt{UD}(\mathcal{L})$ denotes the packing height of the UD algorithm for a list of items $\mathcal{L}$, $\texttt{OPT}(\mathcal{L})$ denotes the optimal packing height for those items, and H denotes the height

Algorithm 5.6 Up-down algorithm

Input: A list $\mathcal{I}$ of n items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items and the strip width $\mathtt{W}$.

Output: A packing of the items in $\mathcal{I}$ into a strip of width $\mathtt{W}$.

place all items where $w(\mathcal{I}_i) > \mathtt{W}/2$ in $\mathcal{L}^1$, let n_1 denote the number of items in $\mathcal{L}^1$
place all items where $\mathtt{W}/3 < w(\mathcal{I}_i) \leq \mathtt{W}/2$ in $\mathcal{L}^2$, let $n_2 = \left|\mathcal{L}^2\right|$
place all items where $\mathtt{W}/4 < w(\mathcal{I}_i) \leq \mathtt{W}/3$ in $\mathcal{L}^3$, let $n_3 = \left|\mathcal{L}^3\right|$
place all items where $\mathtt{W}/5 < w(\mathcal{I}_i) \leq \mathtt{W}/4$ in $\mathcal{L}^4$, let $n_4 = \left|\mathcal{L}^4\right|$
place all items where $w(\mathcal{I}_i) \leq \mathtt{W}/5$ in $\mathcal{L}^5$, let $n_5 = \left|\mathcal{L}^5\right|$
define `Left` to be the left-hand boundary of each level L
`Left` (ℓ, h) is the horizontal coordinate of the boundary for level L_ℓ at height h
define `Right` to be the right-hand boundary of each level L
$h_0 \leftarrow 0$, $h_1 \leftarrow 0$
if $n_1 > 0$ **then**
  sort all items in $\mathcal{L}^1$ according to decreasing width
  stack all items in $\mathcal{L}^1$ adjacent to the left-hand boundary of the strip
  let h_1 denote the height to which items in $\mathcal{L}^1$ have been packed
end if
for sets $\mathcal{L}^2$, $\mathcal{L}^3$ and $\mathcal{L}^4$ **do**
  if $n_2/n_3/n_4 > 0$ **then**
    sort all items in $\mathcal{L}^2/\mathcal{L}^3/\mathcal{L}^4$ according to decreasing width
    for $i = 1$ **to** $n_2/n_3/n_4$ **do**
      `Packed` $\leftarrow$ False
      while not `Packed` **and** $j \leq 1/2/3$ **do**
        if $h_j > h_{j-1}$ **then**
          attempt to stack $\mathcal{L}_i^2/\mathcal{L}_i^3/\mathcal{L}_i^4$ in a downward manner from h_j
          if successful **then**
            `Packed` $\leftarrow$ True, update `Right` (j)
          end if
        end if
      end while
      if not `Packed` **then**
        pack $\mathcal{L}_i^2/\mathcal{L}_i^3/\mathcal{L}_i^4$ into level $L_2/L_3/L_4$ by means of the BL algorithm
        update $h_2/h_3/h_4$ and `Left` (2) /`Left` (3) /`Left` (4) if necessary
      end if
    end for
  end if
end for
if $n_5 > 0$ **then**
  call GNFDH $\left(\mathcal{L}^5, L, \mathtt{Left}, \mathtt{Right}, n_5\right)$
  pack any unpacked items in $\mathcal{L}^5$ above h_4 by means of the NFDH algorithm
end if

of the tallest item in the list $\mathcal{L}$. Moreover, they proved that the asymptotic bound of 5/4 is tight.

Procedure 5.6.1 GNFDH $(\mathcal{E}, L, \texttt{Left}, \texttt{Right}, n_5)$

```
1: sort the items in L^5 according to decreasing height
2: i ← 1, ℓ ← 1
3: while ℓ ≤ 4 and i ≤ n_5 do
4:   if h(L_ℓ) > 0 then
5:     Lower ← 0, Upper ← h(L_ℓ), Top ← Lower + h(L_i^5), Current ← Left(Lower)
6:     while i ≤ n_5 and Top ≤ Upper do
7:       if Current + w(L_i^5) ≤ Right(Top) then
8:         pack L_i^5 into the space, Current ← Current + w(L_i^5), i ← i + 1
9:       else
10:        if Left(Upper) < Current then
11:          H ← min{H̄ | Left(H̄) < Current}
12:        else
13:          H ← Upper
14:        end if
15:        Lower ← max(Top, H), Top ← Lower + h(L_i^5), Current ← Left(Lower)
16:      end if
17:    end while
18:  end if
19:  ℓ ← ℓ + 1
20: end while
```

Worst-case Time Complexity

The UD algorithm begins by assigning each item in $\mathcal{L}$ to one of five sets of items according to the item's width (see lines 1–5). If there are items in $\mathcal{L}^1$, then they are sorted in line 11 according to decreasing width by means of the merge-sort algorithm, which has a worst-case time complexity of $\mathcal{O}(n \log n)$. The items are then stacked onto each other (see line 12) according to a procedure that has a time complexity of $\mathcal{O}(n)$. If there are items in $\mathcal{L}^2$, $\mathcal{L}^3$ and $\mathcal{L}^4$ then they too are sorted according to decreasing width by means of the merge-sort algorithm (see line 17). If the difference between the heights of the lower levels is greater than zero, then an attempt is made to stack items into these levels. In order to determine whether an item fits below the ceiling stack of a lower level, a procedure with a linear worst-case time complexity is required for each of the levels. If the item fits into such a lower level, then the right-hand boundary must be updated. This step (see line 24) has a constant time complexity. If the item does not fit into a lower level, line 29 dictates that it must be packed into the current level j by means of the BL algorithm [6] presented in §5.1.3. In the *worst-case time complexity* section of §5.1.3 it was found that the worst-case time complexity of the BL algorithm is $\mathcal{O}\left(n^3\right)$. Therefore, the *for*-loop spanning lines 15–32 has a worst-case time complexity of $\mathcal{O}\left(n^4\right)$. Thereafter, the GNFDH algorithm is called.

The GNFDH algorithm, the pseudocode of which may be found in Procedure 5.6.1, begins by sorting all items in $\mathcal{L}^5$ according to decreasing height. If this is done by means of the merge-sort algorithm, the time complexity of line 1 is $\mathcal{O}(n \log n)$. The algorithm enters a *while*-loop spanning lines 3–20, the contents of which is executed at most 4 times; once for each of the lower levels. Determining the value for `Current` in line 5 has constant time complexity, because the horizontal coordinate of the lowest left-hand boundary is simply the last item in the list of boundary values. The contents of the *while*-loop spanning lines 6–17 is executed $\mathcal{O}(n_5) = \mathcal{O}(n)$ times. In order to determine the horizontal position of the right-hand boundary in line 7, an

operation with a worst-case time complexity of $\mathcal{O}(n)$ is required. If all item sizes are guaranteed to be integer, then this may be performed in constant time. If the item does fit, then it may be packed in constant time (see line 8), but if it is not packed, then the left-hand boundary at the height `Upper` must be investigated. This may be found in linear time for non-integer data sets. If the left-hand boundary is less than the current horizontal coordinate of the left-hand boundary, then the lowest relevant boundary position (see line 11) may be found in linear time. Once the relevant value for `H` has been found there are some constant-time computations and another search for the horizontal coordinate of the left-hand boundary at a specific height (linear time for non-integer data). Therefore, the combined time complexity of the GNFDH algorithm is $\mathcal{O}(n^2)$.

Any remaining items are packed by means of the NFDH algorithm. No further sorting is required as this has been done during the GNFDH algorithm. Thus the time complexity of line 37 of Algorithm 5.6 is $\mathcal{O}(n)$. The time complexity of $\mathcal{O}(n^4)$ of the *for*-loop spanning lines 15–32 dominates the time complexity of the other steps of the UD algorithm, giving it a worst-case time complexity of $\mathcal{O}(n^4)$.

Practical Considerations

In the previous section it was claimed that the search for the horizontal coordinate of a boundary at a specific height is linear if the item sizes are integer values. If this is the case, then an array such as the one in Figure 4.2(a) may be used to represent the boundaries. However, in order to accommodate noninteger item dimensions, arrays such as those used for the skylines of the BL and BF algorithms, are rather used to represent the two boundaries for each level. In order to create an integer array representing the boundaries, a total of $\mathcal{O}(nH_{\text{ave}})$ operations would be required to create the boundary, where H_{ave} is the average item height. Instead, the use of a skyline construct allows for linear time boundary construction and a linear time search for the horizontal coordinate at a fixed height.

5.1.7 Chazelle's Bottom-Left Bin Packing Algorithm

In 1983, only two years after Baker *et al.* [6] published their BL algorithm, Chazelle [25] suggested improvements to the algorithm. He identified an opportunity in the BL algorithm to pack the items more densely. The BL algorithm attempts to pack items from above, possibly leaving empty spaces (called *holes*) under some items that remain empty during the entire packing. Chazelle's algorithm (called the bottom-left fill algorithm by Hopper [75], and therefore abbreviated as BLF in the remainder of this dissertation) attempts to pack items into the lowest of these holes (if they fit), before packing onto the packed items in the strip (if they do not fit). The algorithm begins by initialising the strip, making it a dynamic hole. As items are packed into this hole, the topmost two points of the hole are moved upwards as the height of the strip increases, in order to remain twice the height of the tallest item above the height of the items packed into the strip. As items are packed, new holes may form. All holes are searched for a valid packing position for each subsequent item. The hole that yields the lowest packing position, or in the case of equal packing heights, the packing position furthest to the left is selected as the location for the item.

There are some structures in a hole that affect how the algorithm finds possible packing positions in a hole. The first is the *leftmost edge* (see Figure 5.17), which is common to all holes. It is defined as a vertical edge with two horizontal edges on either end that are to the right of the

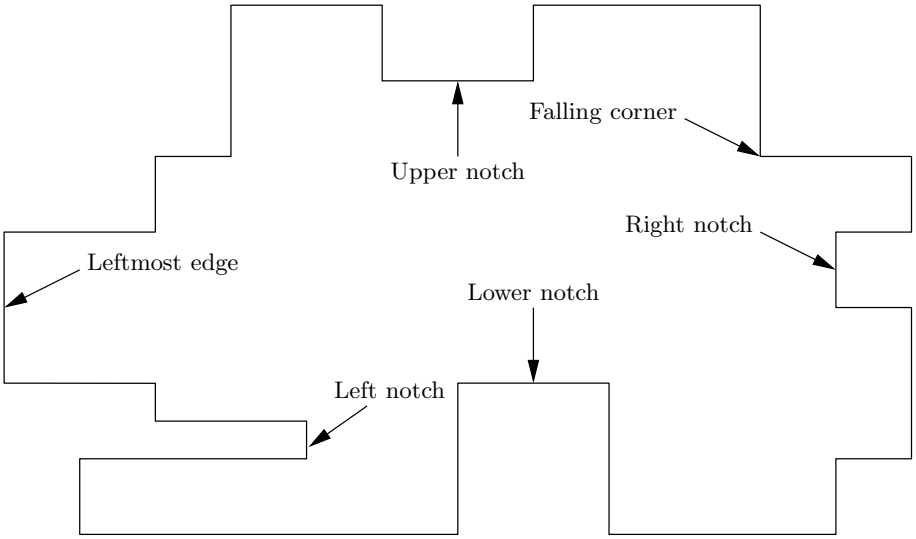


Figure 5.17: *Important edges for the BLF algorithm (adapted from Fig. 4 in [25]).*

edge. The *rightmost edge* is similar, but the horizontal edges are to the left of the vertical edge. There are no edges in a hole that are further to the right than the rightmost edge. All holes have these two structures. A *lower notch* is a horizontal edge that has two vertical edges adjacent to it that are below it. A *left notch* is an edge that has its two adjacent edges to the left. There is one more leftmost edge than left notch. The final important structure is a so-called *falling corner*. This is formed by a vertical edge that drops down from the ceiling, and a horizontal edge that is to the right of the vertical edge and is the top horizontal edge of the structure that defines a rightmost edge.

Two of the structures shown in Figure 5.17 cannot occur when holes appear during the packing process. Chazelle [25, p. 698] proved that an *upper notch* cannot occur, because it would break the rule of the algorithm that forces items to be packed as low as possible. More specifically, an item that creates an upper notch may be packed lower; its lower edge may be located at the same height as the highest point of the floor of the hole directly below the item. Furthermore, there can be no *right notch*, as it would break the rule stating that items are to be packed as far to the left as possible. Any item creating a right notch may be moved further left until it encounters an edge that forms part of a lower notch or the left-hand boundary. The same reasoning may be used to prove that there may be only one falling corner. Consider a packing with two falling corners. The upper falling corner may be restricted from being shifted to the left by the item that forms the ceiling of the hole. The item forming the lower falling corner has no such restriction and may be moved left until its left edge encounters another vertical edge.

Due to the fact that there are no right notches, there may be only one rightmost edge. However, there may be more than one leftmost edge and each leftmost edge requires the creation of a *subhole*. A subhole is a region of a hole that is treated as a individual hole when investigating possible locations for packing. Subholes are created in the following manner: the top corner of each left notch Q_i is used as a starting point from which two further points are identified. The point QN_i is the point on the boundary directly above the left notch Q_i and QW_i is defined as the point on the right-hand boundary immediately to the right of the notch (see Figure 5.18). The edges created by the Q_iQN_i and Q_iQW_i pairs may be combined with the existing boundaries to create subholes with only one leftmost edge each and no left notches. The Q_iQW_i edges form part of the the lower boundary for the leftmost edge L_i, while the Q_iQN_i edges form

part of the boundary for the subhole related to the leftmost edge L_{i-1}.

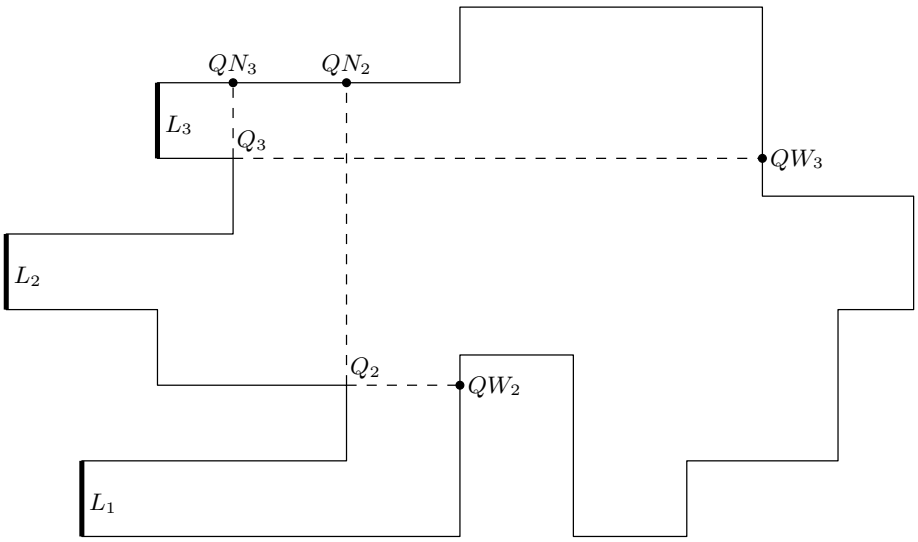


Figure 5.18: *The points used in the BLF algorithm to split a hole into subholes.*

A horizontal bar of length ℓ_b equal to the width of the item to be packed may be used to traverse the lower and upper boundaries of these subholes, and the holes that have a single leftmost edge. The process begins with the bar being placed a distance ℓ_b away along the horizontal axis from the bottom corner of the leftmost edge. The bar is moved to the right until either a vertical edge is encountered, the top end of which is higher than the horizontal edge on which the bar is sliding, or the bar drops to a lower height. A note is made at each change of height of the bar. This process is described in further detail by Chazelle [25, p. 703] and continues until the right-hand point of the bar encounters the rightmost edge. In order to find points along the top boundary of the hole that define permitted locations for packing an item, the same bar is allowed to slide along the boundary, moving upwards with each increase in boundary height, while the right-hand side of the bar does not come into contact with the rightmost edge or vertical edge of the falling corner. This process is described in further detail by Chazelle [25, p. 705]. It is possible that the two lines created by joining all the points in each list may overlap (see Figure 5.19). Once all points have been found, another array of points is created from the two lists of points. This list comprises x and y values of the lists describing the valid packing points along the bottom and top edges of the hole, and a boolean value for each point that is *true* if there is sufficient space between the top and bottom boundaries for the item to be packed. The algorithm that generates this list is described in detail by Chazelle [25, p. 701].

Once every subhole and hole has been investigated for the valid packing location, the item is packed into the hole with the lowest packing position. If there are two or more holes that have a packing location at the same height, the leftmost hole is selected in which to pack the item. A pseudocode listing of the algorithm may be found in Algorithm 5.7.

Worked example

By sorting the items in Table 3.1 in order of decreasing width, the list $\mathcal{I} = \{\mathcal{I}_{13}, \mathcal{I}_{11}, \mathcal{I}_5, \mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_4, \mathcal{I}_6, \mathcal{I}_9, \mathcal{I}_{12}, \mathcal{I}_7, \mathcal{I}_1, \mathcal{I}_8, \mathcal{I}_2\}$ results. The strip is initialised as a hole by setting the following corners to the hole: $(0, 0)$, $(20, 0)$, $(20, 22)$, $(0, 22)$. The value 22 is twice the height of the tallest item $\mathcal{I}_{11}$. The left edge is identified as $((0, 0), (0, 22))$ and there is no falling corner. The search

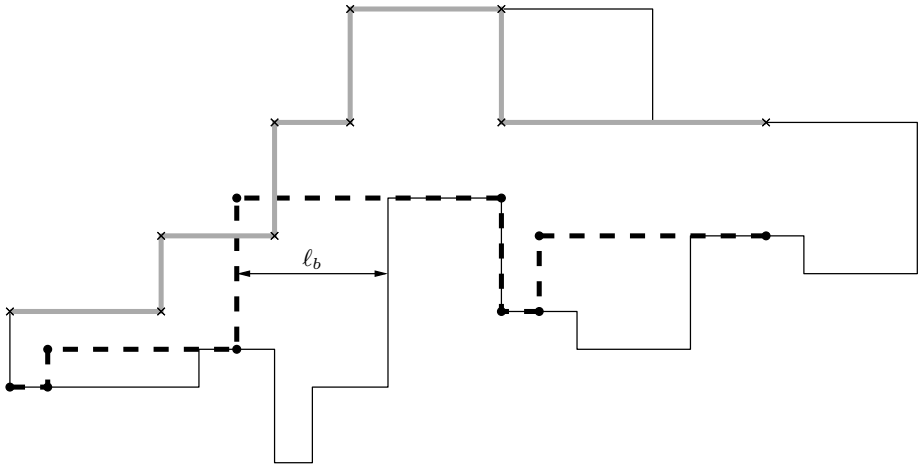


Figure 5.19: *Determining valid packing positions in a hole in the BLF algorithm. The dashed line indicates valid positions for a bar of length ℓ_b along the bottom boundary, while the light grey line indicates all valid positions below which the top of an item may be located.*

for valid packing points yields two locations: $(0, 0)$ and $(4, 0)$. The lowest, leftmost position is selected and $\mathcal{I}_{13}$ is packed with its bottom-left corner at $(0, 0)$. The hole is updated to yield the corners $(0, 5)$, $(16, 5)$, $(16, 0)$, $(20, 0)$, $(20, 27)$, $(0, 27)$ and the leftmost edge at $((0, 5), (0, 27))$.

The search for valid packing locations for $\mathcal{I}_{11}$ yields two locations: $(0, 5)$ and $(14, 5)$. Therefore, $\mathcal{I}_{11}$ is packed onto $\mathcal{I}_{13}$, its left-hand edge coinciding with the left-hand strip boundary. The hole is updated to yield the corners $(0, 14)$, $(14, 14)$, $(14, 5)$, $(16, 5)$, $(16, 0)$, $(20, 0)$, $(20, 36)$, $(0, 36)$ and the leftmost edge at $((0, 14), (0, 36))$. A similar process is undertaken to locate $\mathcal{I}_5$ on top of $\mathcal{I}_{11}$. The search for valid packing locations for $\mathcal{I}_{10}$ yields three positions: $(0, 23)$, $(11, 23)$ and $(11, 14)$. The last location is the lowest position and $\mathcal{I}_{10}$ is packed there to yield two holes. The first has the corners $(0, 23)$, $(11, 13)$, $(11, 21)$, $(20, 21)$, $(20, 45)$ and $(0, 45)$, and the leftmost edge $((0, 23), (0, 45))$, while the second hole has the corners $(14, 5)$, $(16, 5)$, $(16, 0)$, $(20, 0)$, $(20, 14)$ and $(14, 14)$, and the leftmost edge $((14, 5), (14, 14))$.

Item $\mathcal{I}_3$ does not fit into the smaller, lower hole and is therefore located at the lowest position in the top hole; left-aligned on top of $\mathcal{I}_{10}$. Similarly, items $\mathcal{I}_4$ finds its lowest location on top of $\mathcal{I}_5$ and $\mathcal{I}_6$ is packed onto $\mathcal{I}_4$. Item $\mathcal{I}_9$ fits into the lower hole and is packed onto $\mathcal{I}_{13}$, adjacent to $\mathcal{I}_{11}$, thereby creating two new square holes; one below $\mathcal{I}_9$ and to the right of $\mathcal{I}_{13}$, and the other above $\mathcal{I}_9$, below $\mathcal{I}_{10}$ and to the right of $\mathcal{I}_{11}$. Item $\mathcal{I}_{12}$ is too wide to fit into the lowest hole, but it does fit into the middle hole and is packed there. The same applies to $\mathcal{I}_7$. Item $\mathcal{I}_1$ is too tall to fit into the two lower holes and is packed in the lowest possible position within the top hole, above $\mathcal{I}_5$ in the space between $\mathcal{I}_4$ and $\mathcal{I}_3$. Item $\mathcal{I}_8$ is both narrow and short enough to fit into the lowest hole and is packed there. The final item, $\mathcal{I}_2$, is too tall to fit into either of the lower holes and is packed into the lowest position into which it fits in the top hole, the space above $\mathcal{I}_{10}$ and to the right of $\mathcal{I}_3$. The final packing height is 34 and a graphical representation of the packing may be found in Figure 5.20(a).

Worst-case Time Complexity

The BLF algorithm begins by sorting all items according to decreasing width by means of the merge-sort algorithm, which has a worst-case time complexity of $\mathcal{O}(n \log n)$ (see line 1). The step in line 2 is required to determine the height at which the top boundary of the topmost hole

Algorithm 5.7 Chazelle's algorithm

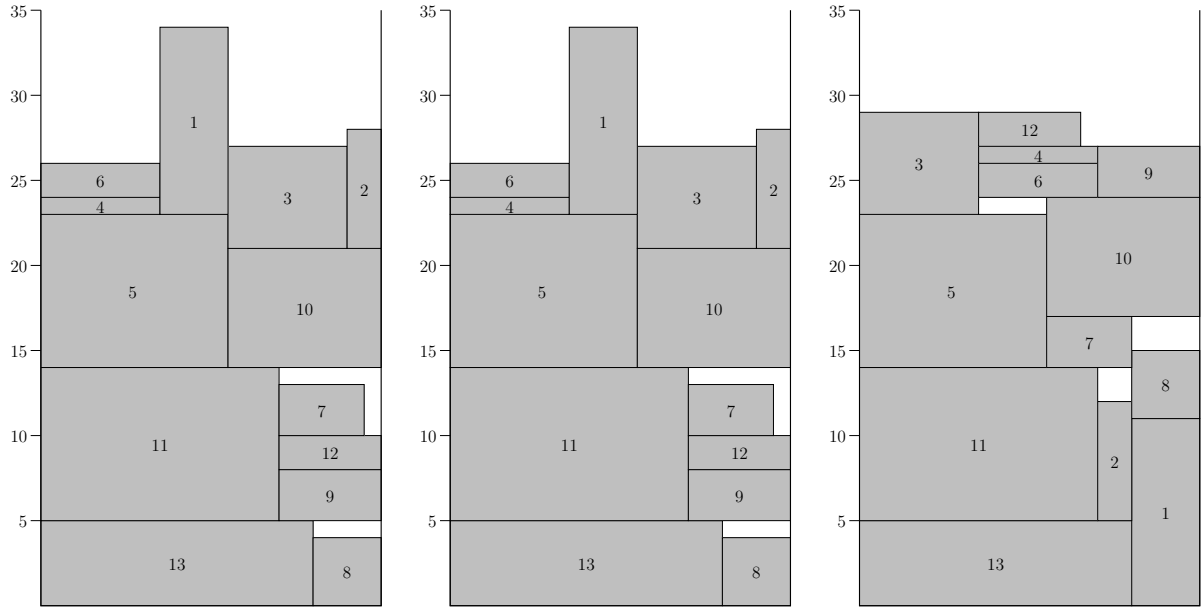
Input: A list $\mathcal{I}$ of items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items and the strip width `W`.

Output: A packing of the items in $\mathcal{I}$ into a strip of width `W`.

```
1: sort items according to decreasing width
2: determine the maximum item height in the list I
3: initialise the strip as a hole
4: for all items do
5:   for all holes do
6:     if the hole has two or more leftmost edges then
7:       for all subholes do
8:         attempt to find a valid packing position for the item in the subhole
9:         if the subhole has a valid packing position then
10:          let X and Y be the coordinates at which the item may be packed
11:          if Y < BestY or (X < BestX and Y = BestY) then
12:            BestX ← X, BestY ← Y, set BestH equal to the index of the hole
13:          end if
14:        end if
15:      end for
16:    else
17:      attempt to find a valid packing position for the item in the hole
18:      if the hole has a valid packing position then
19:        let X and Y be the coordinates at which the item may be packed
20:        if Y < BestY or (X < BestX and Y = BestY) then
21:          BestX ← X, BestY ← Y, set BestH equal to the index of the hole
22:        end if
23:      end if
24:    end if
25:  end for
26:  pack the item into hole BestH at the coordinates (BestX, BestY)
27:  fix the list of points that represent the hole, create new holes if necessary
28:  if the packing has increased the height of the strip then
29:    change the position of the top boundary of the topmost hole to reflect this
30:  end if
31:  update the lists describing the horizontal and vertical edges of the holes and subholes
32: end for
```

must be placed after an item has been packed into that hole. The height of each item must be investigated in this step in order to find the maximum height; hence this line has a time complexity of $\mathcal{O}(n)$. The initialisation of the strip as a hole has a constant time complexity, because the number of steps required to perform this procedure is independent of the number of items in the list.

In order to find a packing location for an item in a hole or subhole, the list of packing locations along the bottom and top boundaries must be determined. A procedure of $\mathcal{O}(e_b)$ complexity, where e_b is the number of vertical edges along the bottom boundary, is required in order to determine the valid packing locations along the boundary (see Chazelle [25, p. 704] for further details). Similarly, a procedure of $\mathcal{O}(e_t)$ complexity, where e_t is the number of vertical edges along the top boundary, is required to determine the valid points along the boundary. Once



(a) BLF (H = 34) (b) GCS (H = 34) (c) BFLM (H = 29)

Figure 5.20: *Results obtained when packing the items in Table 3.1 into a strip of width 20 using the known plane strip packing algorithms described in §5.1.7–§5.1.9. The resulting packing heights* H *are also shown.*

the valid points along each boundary have been found, a procedure with a worst-case time complexity of $\mathcal{O}(e_b + e_t)$ is employed to find the points below which the item may be packed. The number of edges along the top and bottom boundaries is proportional to the number of points p that describe a hole or subhole. Therefore, the time complexity of the procedure to find a suitable location for the item has a worst-case time complexity of $\mathcal{O}(p)$.

Saving the best location has a constant time complexity, so too does the packing procedure (line 26). Unfortunately, Chazelle does not provide detail on how he updates the points in a hole once an item has been packed. In this implementation the first step is a search for the first point along the boundary that has a larger horizontal coordinate than the right-hand edge of the item (a worst-case time complexity of $\mathcal{O}(p)$). This is followed by an anti-clockwise search for the first point on the boundary that comes into contact with any of the item's edges ($\mathcal{O}(p)$ time complexity). Once that point is known, the next point to coincide with the item is sought (another $\mathcal{O}(p)$ operation). If the two points are adjacent, the search for the next point coinciding with the item begins from the second point. If the two points are not adjacent, the new hole that has been created is saved (an $\mathcal{O}(p)$ operation). This process continues until the search has yielded a point either equal to the first starting point, or a point with a vertical coordinate larger than the vertical coordinate of the top edge of the item. The hole is then updated with the new item in place (another $\mathcal{O}(p)$ operation). This is followed by updating the list of vertical and horizontal edges that constitute the top and bottom boundaries of the new holes and/or subholes (line 31), an $\mathcal{O}(s\,p)$ operation. The process of fixing a hole by finding the new list of points that describe its shape is therefore a sum of many $\mathcal{O}(p)$ operations for a total worst-case time complexity of $\mathcal{O}(p)$.

Therefore, the combined worst-case time complexity of the BLF algorithm is $\mathcal{O}(n(s\,p + p + s\,p))$ $= \mathcal{O}(s\,p\,n)$, where s is the combined number of holes and/or subholes that are investigated for

a packing position (contributed in the first term by the *for*-loop spanning lines 5–25). The factor n outside the inner set of parentheses is due to the *for*-loop spanning lines 4–32. If there are no subholes, then a function of $\mathcal{O}(s\,p)$ is $\mathcal{O}(n)$, because no point in a hole may appear in more than one other hole. This means that the searches for packing locations in all the holes will not require more than a number of operations that is a linear function of n. In this case the time complexity of the algorithm would be $\mathcal{O}(n^2)$. However, if there are many subholes, then a function of $\mathcal{O}(s\,p)$ will be at worst $\mathcal{O}(n^2)$, as the same points may be investigated for each subhole within a hole. Therefore, Chazelle's algorithm has a worst-case time complexity of $\mathcal{O}(n^3)$ if one investigates the same points in the hole for each subhole, or $\mathcal{O}(n^2)$ if one can prevent this search for the right-hand boundary from requiring such a search. Chazelle claims a worst-case time complexity of $\mathcal{O}(n^2)$ [25, p. 697].

Practical Considerations

An important aspect of this algorithm is its memory management. There is a large amount of data that requires efficient storage during execution of the algorithm. In order to store the holes, an approximately $n/2 \times 2n$ array is required. The first dimension is used to store the number of holes, while the second dimension is used to store the points that represent each hole. Furthermore, an array of the same size is required to save the details of the subholes. It is impossible for the entire array to be used, but the array must be large enough to accommodate many holes with few points per hole, or few holes with many points per hole. Another four approximately $n/2 \times n$ arrays are required to save the vertical and horizontal edges along the top and bottom boundaries of each hole, and four more for the same edges in each subhole. Further $\mathcal{O}(n)$ arrays are required to save information such as the number of leftmost edges in each hole, the occurrence of a falling corner in a hole/subhole, *etc.* If one considers that the data in an array may consist of quintuples (as the structures that save points and edges are), the amount of memory required grows quickly as n increases.

It would be possible to reduce the volume of data that have to be stored by recalculating the falling corners, subholes and horizontal/vertical edges of each hole/subhole before finding a suitable packing location. However, this would require more processing time. Instead, it is possible to use dynamic arrays whose dimensions change as required. In Visual Basic .NET [118] this is achieved by employing the `ArrayList` class [116]. These are dynamic arrays which may contain any data type. This is useful, because two-dimensional ArrayLists may be created by adding an existing ArrayList to another, so that the first ArrayList occupies a single location in the second ArrayList. In this manner only the space actually required by the data structures representing the packed items as they are at any stage of the packing is required. This reduces the memory footprint considerably, making it possible to solve large problem instances (containing 5 000 items) that could not be solved with the use of static arrays.

5.1.8 The Guillotine Cutting Stock Algorithm

Ten years after Chazelle put forward his bottom-left packing algorithm [25], MacLeod *et al.* [109] published an algorithm that may be used to solve guillotine cutting stock problems. The aim of their work was to find an algorithm that would find a layout of items on a stock sheet with minimum waste, where the items are a subset of a finite set of items. The authors approached the problem by randomly sorting the list of items and packing those that fit onto the stock sheet by means of their *guillotine cutting stock* (GCS) algorithm. They compared the solutions obtained when this was done 100 and 500 times and the minimum waste solution was retained

as the final layout. The algorithm is a plane-packing algorithm and is suitable for finding guillotineable solutions for strip packing problems.

At the heart of the algorithm lies a procedure that determines whether a given layout of items on a stock sheet may be disassembled by means of guillotine cuts. During the packing process three lists maintain the cuts that are required to remove items from the sheet. The first of these cuts are horizontal cuts and the data structure that describes these cuts contains the horizontal and vertical coordinates of the blocking point (the point at which the cut is interrupted by the perpendicular edge of an item), an indicator that saves the position of the blocking point (*i.e.* whether it is a blocking point on the left-hand or right-hand side of the cut), the item that is removed from the layout by the cut, the edge of the item that is cut and a pointer to the blocking point on the other side of the cut (if it exists). This list of blocking points of horizontal cuts is labelled $\mathcal{H}$ and the items in the list are sorted according to increasing horizontal coordinate. The data structure that describes vertical cuts is similar, but instead of left and right blocking points, they are top or bottom blocking points. This list of blocking points of horizontal cuts is labelled $\mathcal{V}$ and are sorted according to vertical coordinate. The third list of cuts, called *minimal cuts* by MacLeod *et al.* [109, p. 402], saves those cuts that have no blocking points, *i.e.* cuts that are guillotine cuts. These cuts have a simpler structure, namely the position, the relevant item removed and edge of the cut and its orientation (horizontal or vertical). The list of minimal cuts is labelled $\mathcal{M}$ and the cuts are not sorted.

The *guillotine slicing* (GS) procedure is employed for each possible location for a new item and cuts along all minimal cuts, deleting any invalid blocking points as it does so. To clarify the deletion process, MacLeod *et al.* [109, p. 404] use the example of a vertical minimal cut, with the item anywhere to the left of the cut. A similar procedure is used for horizontal cuts. Consider a horizontal cut anywhere below the possible item location. The list of cuts in $\mathcal{V}$ is investigated from the front to the back until the horizontal coordinate of a horizontal blocking point is greater than the coordinate of the minimal cut. The cut separates the layout into two regions; an active region and an inactive region. The active region is the layout on the same side of the cut as the item to be packed. One of four cases may occur:

1 If the blocking point is a top blocking point, the point and its partner may be temporarily deleted. The entire cut is in the inactive region and may be ignored.

2 If it is a bottom blocking point and has no partner, the cut is added to the list of minimal cuts.

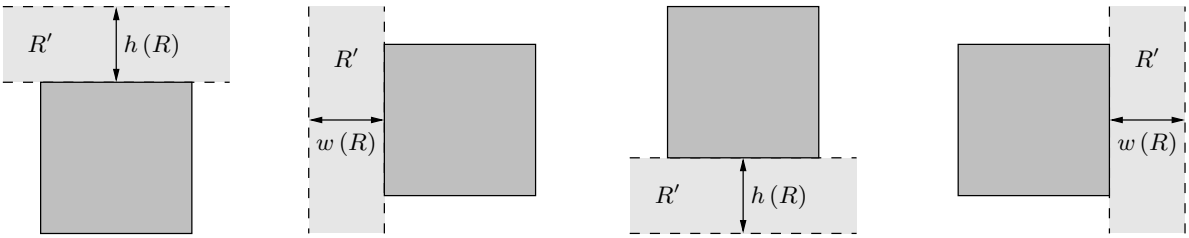
3 If it is a bottom blocking point and the partner blocking point is below the minimal cut, both blocking points may be deleted temporarily as they are both in the inactive region.

4 If it is a bottom blocking point that has a partner blocking point above the minimal cut, the lower blocking point may be deleted, noting the change in the top blocking point. The top blocking point may not be deleted because it is in the active region.

This procedure continues until all minimal cuts have been performed. If no cuts remain, then the tentative location of the item results is guillotine feasible. However, if either of the sets $\mathcal{H}$ or $\mathcal{V}$ are nonempty, then the location of the item does not yield a guillotine layout and should be ignored.

MacLeod *et al.* [109, p. 405] observed that every waste piece in a layout, into which subsequent items may be packed, "must completely contain, along one of its four borders, the entire face of some already placed rectangle." They define an *exposed border* of a layout to be an edge of

a packed item that does not coincide with an item of any other edge anywhere along its edge. Therefore, the GCS algorithm investigates all exposed borders of the items already packed as possible locations for an item. It does this by creating a critical region adjacent to the exposed border, using the GS procedure to determine whether the placement of the critical region results in a guillotine feasible layout. The size of the critical region is determined by the new item's dimensions and the items that have already been packed. If the exposed border is horizontal (top or bottom edge of the item), then the critical region's height is equal to the height of the new item. If the exposed border is vertical (left or right edge of the packed item), then the width of the new item determines the width of the critical region. The other dimension is determined by the other items that have already been packed (see Figure 5.21). If the critical region is too small for the new item to fit into, then this location for the item is ignored and the next exposed border is considered.



(a) Top border exposed (b) Left-hand border exposed (c) Bottom border exposed (d) Right-hand border exposed

Figure 5.21: *Construction of the critical region during execution of the GCS algorithm.*

If the critical region is large enough for the item to fit into it, then the GS procedure is used to determine whether the critical region may be disassembled from the layout by means of guillotine cuts. If the result is true, then the item may be packed in the bottom-left corner of the critical region. However, if the critical region is larger than the item to be packed, but is not guillotine feasible, then a list $\mathcal{E}$ of cuts is created passing through the critical region before it was created in order of increasing horizontal (vertical) coordinates. First, an attempt is made to pack the item on the left-hand (bottom) border. If the GS procedure returns the result that the location is guillotine infeasible, then the item is packed so that its left-hand (bottom) edge coincides with the first cut in $\mathcal{E}$. This process continues until a guillotine feasible location is found or the horizontal (vertical) coordinate plus the width (height) of the item exceeds the right-hand (top) boundary of the critical region. If no guillotine feasible location has been found, the algorithm continues by temporarily packing the item so that its right-hand (top) edge coincides with the right-hand (top) boundary of the critical region. The item is temporarily packed so that its right-hand (top) edge coincides with each of the cuts in $\mathcal{E}$ until either there are no more cuts, or the horizontal (vertical) coordinate of the left-hand (bottom) edge is less than the horizontal (vertical) coordinate of the left-hand (bottom) boundary of the critical region. If one or more of the positions yield a guillotine feasible location, the leftmost (lowest) position is chosen as the final best location for the new item alongside the current exposed border. Once the lowest leftmost location is found for the item, the exposed borders of the items are updated; the edges of items that coincide with the edges of the new item are marked as unexposed, as are those edges on the new item. This process continues until all items have been packed into guillotine feasible locations. A pseudocode listing of the GCS algorithm may be found in Algorithm 5.8.

Algorithm 5.8 Guillotine cutting stock (GCS) algorithm

Input: A list $\mathcal{I}$ of items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items and the strip width `W`.

Output: A packing of the items in $\mathcal{I}$ into a strip of width `W`.

```
1: sort the items if desired
2: pack the first item in the bottom-left corner and add the first two M cuts
3: initialise the top and left-hand edges as exposed borders
4: for i = 1 to n do
5:   for j = 1 to i − 1 do
6:     for each exposed border do
7:       create the critical region CritRegion adjacent to the exposed border of I_j
8:       if CritRegion is large enough for I_i and may yield a lower packing then
9:         call GS(H, V, M, CritRegion)
10:        if layout is guillotine feasible and location is lower than current best then
11:          save the current position as the best to this point
12:        else if layout is not guillotine feasible then
13:          if packing I_i against the left-hand/bottom boundary is guillotine feasible then
14:            pack the item in the bottom-left corner of the critical region
15:          else
16:            call GCS-E(I, H, V, M, CritRegion)
17:          end if
18:        end if
19:      end if
20:    end for
21:  end for
22:  save the cuts associated with the best location to the list of cuts
23:  pack the item into its best location and update the exposed borders
24: end for
```

Worked example

By sorting the items in Table 3.1 in order of decreasing width, the list $\mathcal{I} = \{\mathcal{I}_{13}, \mathcal{I}_{11}, \mathcal{I}_5, \mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_4, \mathcal{I}_6, \mathcal{I}_9, \mathcal{I}_{12}, \mathcal{I}_7, \mathcal{I}_1, \mathcal{I}_8, \mathcal{I}_2\}$ results. The first item, $\mathcal{I}_{13}$, initialises the strip by being packed in the bottom-left corner. The top and right-hand edges are marked as exposed borders, one horizontal minimal cut is created at a vertical coordinate of 5 and a vertical minimal cut is created at a horizontal coordinate of 16. The list $\mathcal{M} = \{(5, H, T, 13), (16, V, R, 13)\}$ results[2].

Item $\mathcal{I}_{11}$ follows in the list and the first exposed border that is investigated is the top edge of $\mathcal{I}_{13}$. A tentative critical region is created at a vertical coordinate of five and a horizontal coordinate of zero, with a height equal to the height of $\mathcal{I}_{11}$ and a width equal to the strip width. No packed items occur in this critical region; hence the tentative critical region becomes the final critical region. The list of cuts is updated: $\mathcal{M}_1$ remains, while $\mathcal{M}_2$ is deleted with the addition of the blocking point[3] $(16, 5, R, 1, T, -1)$ to $\mathcal{V}$. Moreover, two cuts are added to the list $\mathcal{M}$: one for the bottom edge of the critical region and another for its top edge. Once

[2]The data structure for minimal cuts takes the following form: coordinate, **H**orizontal/**V**ertical, edge cut (**T**op, **B**ottom, **L**eft or **R**ight), item (or item index) cut.

[3]The notation for blocking points takes the following form: horizontal coordinate, vertical coordinate, edge cut (**T**op, **B**ottom, **L**eft or **R**ight), item (or item index) cut, location of the blocking point (**T**op, **B**ottom, **L**eft or **R**ight), partner blocking point (or its index, -1 if there is no partner point).

Procedure 5.8.1 GS($\mathcal{H}, \mathcal{V}, \mathcal{M}$, `CritRegion`)

update the cuts with the critical region in place
let f_h and l_h (f_v and l_v) be the first and last $\mathcal{H}$ ($\mathcal{V}$) cuts, respectively
let m be the number of $\mathcal{M}$ cuts, $i \leftarrow 1$
while $i \leq m$ **do**
  if $\mathcal{M}_i$ is a vertical cut and to the left of `CritRegion` **then**
    $j \leftarrow f_h$
    while horizontal coordinate of $\mathcal{H}_j <$ horizontal coordinate of $\mathcal{M}_i$ **do**
      process the cut in a manner as described in the text
      $j \leftarrow$ the item after j in $\mathcal{H}$, add to $\mathcal{M}$ and m if necessary
    end while
  else if $\mathcal{M}_i$ is a vertical cut and to the right of `CritRegion` **then**
    $j \leftarrow l_h$
    while horizontal coordinate of $\mathcal{H}_j >$ horizontal coordinate of $\mathcal{M}_i$ **do**
      process the cut in a manner as described in the text
      $j \leftarrow$ the item before j in $\mathcal{H}$, add to $\mathcal{M}$ and m if necessary
    end while
  else if $\mathcal{M}_i$ is a horizontal cut and below `CritRegion` **then**
    $j \leftarrow f_v$
    while vertical coordinate of $\mathcal{V}_j <$ vertical coordinate of $\mathcal{M}_i$ **do**
      process the cut in a manner as described in the text
      $j \leftarrow$ the item after j in $\mathcal{V}$, add to $\mathcal{M}$ and m if necessary
    end while
  else if $\mathcal{M}_i$ is a horizontal cut and above `CritRegion` **then**
    $j \leftarrow v_l$
    while vertical coordinate of $\mathcal{V}_j >$ vertical coordinate of $\mathcal{M}_i$ **do**
      process the cut in a manner as described in the text
      $j \leftarrow$ the item before j in $\mathcal{V}$, add to $\mathcal{M}$ and m if necessary
    end while
  end if
end while

Procedure 5.8.2 GCS-E($\mathcal{I}, \mathcal{H}, \mathcal{V}, \mathcal{M}$, `CritRegion`)

find the set $\mathcal{E}$ of e cuts that pass through the critical region, $k \leftarrow 1$
while no packing has been found **and** $k \leq e$ **do**
  update the cuts $\mathcal{H}$, $\mathcal{V}$ and $\mathcal{M}$ for the new critical region
  test guillotine feasibility if $\mathcal{I}_i$ is packed left-aligned at $\mathcal{E}_k$, $k \leftarrow k + 1$
  if a potentially better packing location has been found, save it
end while
if no feasible location has been found **then**
  reset the cuts in $\mathcal{E}$
  for $k = e$ **to** 1 **do**
    update the cuts for the new critical region
    test guillotine feasibility if $\mathcal{I}_i$ is packed right-aligned at $\mathcal{E}_k$
    if there is a potential better packing location, save it
  end for
end if

the cuts have been updated, the GS procedure in employed to determine whether the critical region in its current location yields a guillotine feasible layout. The first cut in $\mathcal{M}$ is the cut along the top edge of $\mathcal{I}_{13}$. There is only one blocking point that forms part of a vertical cut: it is a top blocking point and the critical region is above $\mathcal{M}_1$ — hence it is deleted, because the cut is in the inactive region. No further cuts remain in the lists $\mathcal{H}$ and $\mathcal{V}$. Therefore, the location of the critical region yields a guillotine layout and $\mathcal{I}_{11}$ may tentatively be packed with its bottom-left edge at the coordinates $(0, 5)$, followed by an update of the cuts for the item at this location (the change yields a blocking point $(14, 5, R, 11, B, -1)$ and the return of the minimal cut $(16, V, R, 13)$ as the critical region no longer blocks the cut). The right-hand edge of $\mathcal{I}_{13}$ is the last remaining exposed border, but there is insufficient space for the item to be packed there, which results in $\mathcal{I}_{11}$ being packed permanently at its tentative location. The top edge of $\mathcal{I}_{13}$ is removed from the list of exposed borders and the top and right-hand edges of $\mathcal{I}_{11}$ are added to the list.

The right-hand edge of $\mathcal{I}_{13}$ is the first location for a critical item based on the dimensions of $\mathcal{I}_5$. However, the space between $\mathcal{I}_{13}$ and the right-hand boundary of the strip is too small for $\mathcal{I}_5$. This is followed by an attempt to place $\mathcal{I}_5$ on top of $\mathcal{I}_{11}$ which yields a guillotine feasible layout. The space between $\mathcal{I}_{11}$ and the boundary of the strip is insufficient for $\mathcal{I}_5$ to be packed; hence its final location is on top of $\mathcal{I}_{11}$. This results in the removal of the top edge of $\mathcal{I}_{11}$ from the list of exposed edges, and the addition of the top and right-hand edges of $\mathcal{I}_5$ to the list. Item $\mathcal{I}_{10}$ does not fit adjacent to the exposed borders of $\mathcal{I}_{13}$ or $\mathcal{I}_{11}$, but it does produce guillotine feasible layouts when placed adjacent to the top and right-hand edges of $\mathcal{I}_5$. Placing $\mathcal{I}_{10}$ adjacent to the right-hand edge of $\mathcal{I}_5$ yields the lowest packing; hence its temporary placement there is made permanent. The remaining exposed borders are the right-hand edges of $\mathcal{I}_{13}$ and $\mathcal{I}_{11}$ and the top edges of $\mathcal{I}_5$ and $\mathcal{I}_{10}$.

An attempt is made to pack $\mathcal{I}_3$ adjacent to $\mathcal{I}_{13}$ and $\mathcal{I}_{11}$, but there is insufficient space between those items and the right-hand boundary of the strip. The top edge of $\mathcal{I}_5$ is the next exposed border in the list and the creation of a critical region above the item yields a guillotine feasible layout. Therefore $\mathcal{I}_3$ is temporarily packed onto $\mathcal{I}_5$. However, placing $\mathcal{I}_3$ on top of $\mathcal{I}_{10}$ also yields a guillotine layout; this packing location is lower than the top of $\mathcal{I}_5$ and it is the final exposed border; hence $\mathcal{I}_3$ is permanently packed onto $\mathcal{I}_{10}$, as far to the left as possible. The list of exposed borders is now $(13, R)$, $(11, R)$, $(5, T)$, $(3, T)$ and $(3, R)$, where the first entry of a pair is the item and the second entry is the edge that is an exposed border (**T**op, **B**ottom, **L**eft or **R**ight).

Item $\mathcal{I}_4$ is too wide to be packed adjacent to $\mathcal{I}_{13}$ and $\mathcal{I}_{11}$, but it does yield a guillotine feasible layout when packed onto $\mathcal{I}_5$. The exposed border on top of $\mathcal{I}_3$ is ignored because it would yield a packing higher than the current best position on $\mathcal{I}_5$. The final exposed border is also ignored because $\mathcal{I}_4$ does not fit into the space between $\mathcal{I}_3$ and the right-hand boundary of the strip. Thus $\mathcal{I}_5$ remains on top of $\mathcal{I}_5$. The list of exposed borders increases to $(13, R)$, $(11, R)$, $(3, T)$, $(3, R)$, $(4, T)$ and $(4, R)$. Item $\mathcal{I}_6$ has the same width as $\mathcal{I}_4$ and hence does not fit to the right of $\mathcal{I}_{13}$, $\mathcal{I}_{11}$ and $\mathcal{I}_3$. Packing it on top of $\mathcal{I}_3$ yields a guillotine feasible layout, but packing it on top of $\mathcal{I}_4$ yields a guillotine layout that is lower; hence its final position there. The list of exposed borders is now $(13, R)$, $(11, R)$, $(3, T)$, $(3, R)$, $(4, R)$, $(6, T)$ and $(6, R)$.

The widest unpacked item is $\mathcal{I}_9$ and it too does not fit between $\mathcal{I}_{13}$ and the right-hand boundary of the strip. However, it does yield a guillotine feasible, non-overlapping layout when the second exposed border is investigated. It also fits onto items $\mathcal{I}_6$ and $\mathcal{I}_3$, but these are at higher vertical coordinates and are thus ignored. The remaining exposed borders are all right-hand edges that do not yield a critical region large enough to accommodate $\mathcal{I}_9$. Therefore $\mathcal{I}_9$ is permanently

packed to the right of $\mathcal{I}_{11}$. This requires the removal of the edge from the list of exposed borders and only the top edge of $\mathcal{I}_9$ is added as the left-hand edge coincides with item $\mathcal{I}_{11}$, the bottom edge coincides with $\mathcal{I}_{13}$ and the right-hand edge coincides with the right-hand boundary of the strip. Item $\mathcal{I}_{12}$ has the same width and fits onto $\mathcal{I}_9$. For similar reasons as for $\mathcal{I}_9$, this is where $\mathcal{I}_{12}$ is best packed, and $\mathcal{I}_7$ is best packed on top of $\mathcal{I}_{12}$. The exposed borders that remain are $(13, R)$, $(3, T)$, $(3, R)$, $(4, R)$, $(6, T)$, $(6, R)$, $(7, T)$ and $(7, R)$.

Item $\mathcal{I}_1$ may be narrow enough to fit between $\mathcal{I}_{13}$ and the strip boundary, but the critical region is bound from above by $\mathcal{I}_9$, which yields a critical region too short for $\mathcal{I}_1$. Item $\mathcal{I}_1$ does fit onto $\mathcal{I}_3$, yields a guillotine packing in that position and is tentatively packed there. The space between $\mathcal{I}_3$ and the strip boundary is not wide enough for $\mathcal{I}_1$ to be packed there, but the creation of a critical region adjacent to the right-hand edge of $\mathcal{I}_4$ yields a guillotine feasible layout and a packing lower than on $\mathcal{I}_3$. Therefore $\mathcal{I}_1$ is tentatively packed between $\mathcal{I}_4$ and $\mathcal{I}_3$. The item does yield a guillotine feasible layout when packed adjacent to the two remaining exposed borders (the top and right-hand sides of $\mathcal{I}_6$). However, these are locations higher than when packed adjacent to $\mathcal{I}_4$ and the location there is made permanent. This leaves the exposed borders $(13, R)$, $(3, T)$, $(3, R)$, $(6, T)$, $(7, T)$, $(7, R)$ and $(1, T)$.

Item $\mathcal{I}_8$ is the first item that may be packed adjacent to $\mathcal{I}_{13}$ and is tentatively placed there. If placed onto items $\mathcal{I}_3$, $\mathcal{I}_6$ or $\mathcal{I}_1$ it would also yield guillotine feasible results, but these are at higher vertical coordinates and are therefore ignored. The item does not fit between $\mathcal{I}_3$ and the strip boundary, nor does it fit above or to the right of $\mathcal{I}_7$. Therefore its location adjacent to $\mathcal{I}_{13}$ is made permanent. The list of exposed borders is now $(3, T)$, $(3, R)$, $(6, T)$, $(7, T)$, $(7, R)$, $(1, T)$ and $(8, T)$. Item $\mathcal{I}_2$ yields a guillotine feasible layout when packed onto $\mathcal{I}_3$ and is tentatively placed there. However, it is narrow enough to fit between $\mathcal{I}_3$ and the right-hand strip boundary and is tentatively placed there due to a lower packing location. Although it could have been packed onto $\mathcal{I}_6$ or $\mathcal{I}_1$, those positions are ignored as they are located at a higher vertical coordinate. Moreover, the item is too large to be packed adjacent to either of $\mathcal{I}_7$'s exposed borders, or onto $\mathcal{I}_8$; hence its position to the right of $\mathcal{I}_3$ is finalised, resulting in the packing shown in Figure 5.20(b).

Worst-case Time Complexity

MacLeod *et al.* [109, p. 404] proved that the GS procedure has $\mathcal{O}(n)$ worst-case time complexity. Initially the procedure updates all the cuts with a critical region or item in place (see line 1), which is an $\mathcal{O}(n)$ operation. There are $\mathcal{O}(n)$ cuts in total, spread over at most three lists: $\mathcal{M}$, $\mathcal{H}$ and $\mathcal{V}$. Each cut comprises at most two blocking points and none of these blocking points are considered more than three times. The largest number of operations that can take place on a cut is when its partner blocking point is deleted, when the cut is made a minimal cut and when the minimal cut is applied.

The part of the algorithm that moves the new item along the list $\mathcal{E}$ of cuts (Procedure 5.8.2 here) is claimed to have linear time complexity by MacLeod *et al.* [109, p. 407]. First the cuts that are passing through the critical region have to be found (see line 1); this has a time complexity of $\mathcal{O}(n)$ because the location of each existing cut has to be compared to the location of the critical region. Thereafter, the item is placed left-aligned against each cut in $\mathcal{E}$ until either a guillotine feasible packing location has been found, or insufficient space remains for the item. As the procedure moves the item from one cut in $\mathcal{E}$ to the next, MacLeod *et al.* [109] claim that the number of cuts that the GS procedure needs to evaluate diminishes accordingly and the blocking points that are covered as the item is moved will only be changed once. Therefore,

the *while*-loop that spans lines 2–6 has a linear time complexity. For the same reasons the *for*-loop spanning lines 9–13 has a linear time complexity. Thus, the overall time complexity for Procedure 5.8.2 is $\mathcal{O}(n)$.

The GS algorithm begins by sorting the list of items (see line 1). If this is done by means of the merge-sort algorithm, the time complexity is $\mathcal{O}(n \log n)$. The packing of the first item in line 2 has a constant time complexity, as does the initialisation of the exposed borders in line 3. The two *for*-loops spanning lines 4–24 and 5–21 contain the creation of the critical region in line 7; an $\mathcal{O}(n)$ operation. If the region is large enough, then Procedure 5.8.1 is called in line 9, which has been shown to have a time complexity of $\mathcal{O}(n)$. If the layout is guillotine feasible, the item's position is saved by a procedure that has a constant time complexity. If the layout is not guillotine feasible, then Procedure 5.8.2, which has been shown to have an $\mathcal{O}(n)$ time complexity, is called. Once all exposed borders have been investigated (*i.e.* when the *for*-loop spanning lines 5–21 is exited), the item is packed into the best position (constant time complexity), the lists of cuts are updated (linear time complexity) and the exposed borders of the items are updated (linear time complexity). Therefore, the GCS algorithm has a time complexity of

$$\mathcal{O}(n \log n) + \mathcal{O}(1) + \mathcal{O}(n) \times (\mathcal{O}(n) \times (\mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n)) + \mathcal{O}(n) + \mathcal{O}(n)) = \mathcal{O}\left(n^3\right).$$

Practical Considerations

Linked lists are useful when applied to the GCS algorithm. Many cuts are generated and deleted (when moved from one list to another) throughout execution of the algorithm, and linked lists prevent unnecessary investigations of cuts that are no longer valid. The list of exposed borders may be linked by means of a linked list data structure, since there are typically many edges that are not exposed and checking them may waste much time. The flexibility of the ArrayList class [116] in Visual Basic .NET [118] is useful when attempting to keep the memory use as low as possible in implementations of the GCS algorithm. A cut is represented by a structure consisting of many variables which take up a large amount of space. When this cut is deleted, the ArrayList location may be set equal to `Nothing`, thereby freeing the memory.

5.1.9 The Best-Fit Algorithm

In 2004 Burke *et al.* [22] introduced the *best-fit* (BF) algorithm. Although it was originally created for packing problems allowing rotation, Ntene [125] showed how it can be applied to oriented strip packing. Initially the items are sorted according to decreasing width. The algorithm finds the lowest gap, finds the widest item that fits into the gap, and packs it according to the packing policy decided upon. The BF algorithm either packs the item into the gap in the *leftmost* (LM) position, adjacent to the *tallest neighbour* (TN) or adjacent to the *shortest neighbour* (SN). An array records the height at each unit interval after each packing in order to accelerate the identification of the lowest point. The weakness of this method is that the dimensions of all rectangles included in the problem (including the strip width) must be integer values. This was discussed in greater detail in §4.1. The result is not guaranteed to be guillotineable. A pseudocode listing of the BFLM algorithm may be found in Algorithm 5.9.

Algorithm 5.9 Best-Fit Left-Most (BFLM) algorithm

Input: A list $\mathcal{I}$ of items to be packed, the dimensions $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$ of the items and the strip width `W`.

Output: A packing of the items in $\mathcal{I}$ into a strip of width `W`.

```
sort items according to decreasing width, resolving equalities by sorting according to decreasing height
initialise the skyline
while there are unpacked items do
    identify the lowest part of the skyline
    if there is an item that fits into the space then
        pack the first item that fits into the space
        update the skyline
    else
        raise that skyline part to the height of the lowest of its neighbours
    end if
end while
```

Worked Example

By sorting the items in Table 3.1 in order of decreasing width, the list $\mathcal{I} = \{\mathcal{I}_{13}, \mathcal{I}_{11}, \mathcal{I}_5, \mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_4, \mathcal{I}_6, \mathcal{I}_9, \mathcal{I}_{12}, \mathcal{I}_7, \mathcal{I}_1, \mathcal{I}_8, \mathcal{I}_2\}$ results for the BFLM algorithm. There is only one skyline segment, that part that spans the floor of the strip. The best item to fill this space is $\mathcal{I}_{13}$ and it is packed into the bottom-left corner of the strip creating two sections of skyline: the first spanning the top edge of $\mathcal{I}_{13}$ and the other spanning the length of the floor not under $\mathcal{I}_{13}$. The lowest skyline segment has a width of 4. The first item in the list that has the same width is $\mathcal{I}_1$ and it is packed into the space, resulting in two skyline segments; one above $\mathcal{I}_{13}$ at a height of 5 and another above $\mathcal{I}_1$ at a height of 11. The lowest skyline segment has a width of 16 and the first item to fit into this space is $\mathcal{I}_{11}$. Therefore $\mathcal{I}_{11}$ is packed left-justified onto $\mathcal{I}_{13}$ yielding three skyline segments; the lowest of which has a width of 2 and is at a height of 5. The only item that fits into this space is $\mathcal{I}_2$ and it is packed between $\mathcal{I}_{11}$ and $\mathcal{I}_1$. The lowest skyline segment is the top edge of $\mathcal{I}_1$ and has a width of 4. Item $\mathcal{I}_8$ has the same width as is packed onto $\mathcal{I}_1$, yielding a new skyline profile in which the top edge of $\mathcal{I}_2$ is the lowest segment. This segment is narrower than any unpacked item. Therefore the skyline segment is raised to the same height as the skyline segment above $\mathcal{I}_{11}$. This segment now has a width of 16 and is large enough to accommodate $\mathcal{I}_5$. The packing now consists of four segments and the lowest segment remains the section above $\mathcal{I}_2$. No items fit into the space between $\mathcal{I}_{11}$ and $\mathcal{I}_8$, resulting in the raising of the segment to the same height as the segment immediately to the right of $\mathcal{I}_5$. This segment has the same width as $\mathcal{I}_7$, which results in its packing there. At this stage the skyline consists of three segments: the top edges of $\mathcal{I}_5$, $\mathcal{I}_7$ and $\mathcal{I}_8$. The segment above $\mathcal{I}_8$ is the lowest, but narrower than any of the unpacked items. Raising the segment to the height of the segment above $\mathcal{I}_7$ yields a segment of the same width as $\mathcal{I}_{10}$. Once $\mathcal{I}_{10}$ is packed onto this segment, the skyline is reduced to two segments. The segment above $\mathcal{I}_5$ is the lowest and wide enough to accommodate $\mathcal{I}_3$. This yields a segment too narrow for any unpacked items. Once the segment is raised to the height of the top edge of $\mathcal{I}_{10}$, items $\mathcal{I}_6$ and $\mathcal{I}_9$ may be packed into the remaining space. Item $\mathcal{I}_4$ may be packed onto $\mathcal{I}_6$ and finally $\mathcal{I}_{12}$ is packed onto $\mathcal{I}_4$ to yield a packing height of 29; the lowest packing height achieved by any reviewed algorithm for this set of items. The packing is shown graphically in Figure 5.20(c).

Worst-case Time Complexity

If the merge-sort algorithm is used to sort the items, then the time complexity of line 1 is $\mathcal{O}(n \log n)$. The skyline initialisation step in line 2 has a time complexity of $\mathcal{O}(n)$, as a skyline segment is required for every item. The contents of the *while*-loop spanning lines 3–11 is executed at least once for every item and for instances where the lowest skyline segment is too narrow for the remaining unpacked items to be packed. The identification of the lowest skyline segment in line 4 has a time complexity of $\mathcal{O}(n)$. A constant time operation exists to discover whether there is an item that may fit onto the skyline segment. This may be achieved by comparing the width of the skyline segment to the width of the last unpacked item. If the item width is less than the width of the skyline segment, then at least one item fits into this space. The identification of a suitable item for packing in line 6 has a worst-case time complexity of $\mathcal{O}(n)$ as every unpacked item may be compared for suitability if the last unpacked item is the only one that does fit. When an item has been packed, the skyline must be updated (see line 7). This operation has a worst-case time complexity of $\mathcal{O}(n)$ as a suitable index value in the skyline must be found to represent the new segment of the skyline. If there is no item that fits into the skyline segment, the skyline is raised by means of a procedure that has a constant time complexity. Therefore, the overall worst-case time complexity of the BF algorithm is

$$\mathcal{O}(n \log n) + \mathcal{O}(n) \times (\mathcal{O}(n) + \max\{\mathcal{O}(n) + \mathcal{O}(1), \mathcal{O}(1)\}) = \mathcal{O}\left(n^2\right).$$

Algorithmic Variations

Burke *et al.* [22, 23] designed the algorithm to take into account rotations of items during the packing process. Their algorithm rotates the items in such a way that the width is greater than, or equal to the height before the sorting takes place. During the packing process the items may be rotated in order to fill the low spaces in the skyline in such as manner as to fill the gap or leave the smallest gap possible after having been packed. After all the items have been packed, the algorithm calls Procedure 5.9.1 in an attempt to reduce the impact of the items that were packed such that their heights are greater than their widths.

Procedure 5.9.1 Postprocessing stage of the BF algorithm

```
1: while optimisation is not complete do
2:    find the item I_h whose top edge is highest
3:    if w(I_h) ≥ h(I_h) then
4:       optimisation is complete
5:    else
6:       remove I_h and rotate it by 90°
7:       fix the skyline until the lowest space is large enough for I_h
8:       pack the item into this space according to the placement policy
9:       if the packing is not better then
10:         return the item to its previous orientation and position
11:         the optimisation is complete
12:      end if
13:   end if
14: end while
```

It became clear to the author during the testing phase that the algorithm does not convert well to the oriented packing problem. The algorithm does not pack very efficiently if there

are tall items that are very thin. These would often be packed last as the items are sorted according to their width. In an attempt to improve the results, a new version of the algorithm was designed by the author where the items are sorted according to decreasing area, resolving any equalities by sorting those items according to decreasing width. When a low segment of skyline is identified, the first unpacked item in the list is packed in an attempt to fill the space. No attempt is made to pack in a best-fit manner. Although the name is perhaps no longer appropriate as the algorithm no longer makes use of the best-fit practice of leaving the smallest gap possible after a packing, these forms of the best-fit leftmost, best-fit tallest neighbour and best-fit shortest neighbour algorithms are denoted as BFmLM(DADW), BFmTN(DADW) and BFmSN(DADW), respectively.

Practical Considerations

These algorithms may be implemented efficiently by making use of linked lists for both the items and the skyline. The BFLM algorithms will always pack items in the left-hand corner of a skyline segment by means of the procedure illustrated in Figure 5.6. However, unlike in the BL and BFLM algorithms, the BFTN and BFSN algorithms may pack items in the right-hand corner of a skyline segment. The procedure to correct the skyline in this case has the form illustrated in Figure 5.22. Once the item has been added to the skyline, the same procedures as those shown in Figures 5.7 and 5.8 may be used to ensure that any two segments are joined if they are adjacent and have the same height.

```
S(n).active ← True
S(n).Y ← S(o).Y.I(i).H
S(n).X ← S(o).X + S(o).W − I(i).W
S(n).W ← I(i).W
S(n).nxt ← S(o).nxt
S(n).prv ← o
if S(o).nxt > −1 then
    S(S(o).nxt).prv ← n
end if
S(o).W ← S(o).W − I(i).W
S(o).nxt ← n
```

Figure 5.22: *Adding to the skyline during execution of the BF algorithms when an item has been packed in the right-hand corner of a skyline segment.*

5.2 A New Categorisation of Plane-Packing Heuristics

If one compares the algorithms in §5.1, it becomes clear that the algorithms may be partitioned into two classes. The first class of algorithms is sorting-dependent, while the second class is sorting-independent.

5.2.1 Sorting-Dependent Algorithms

The class of sorting-dependent algorithms includes all level and pseudolevel algorithms, Sleator's algorithm, the split-fit algorithm, the split algorithm, the mixed algorithm and the up-down algorithm. These are called sorting-dependent, because their packing efficiency either depends heavily on the manner in which the items are sorted, or because the items are sorted into groups and each group is sorted in a specific manner. For example, consider any level-packing algorithm. If the items are not sorted according to decreasing height, the result necessarily will be worse than if they were sorted according to decreasing height. If an item initialises a level and the following item is taller, a new level must be created. This new level may have been avoided if the taller item appeared first in the list.

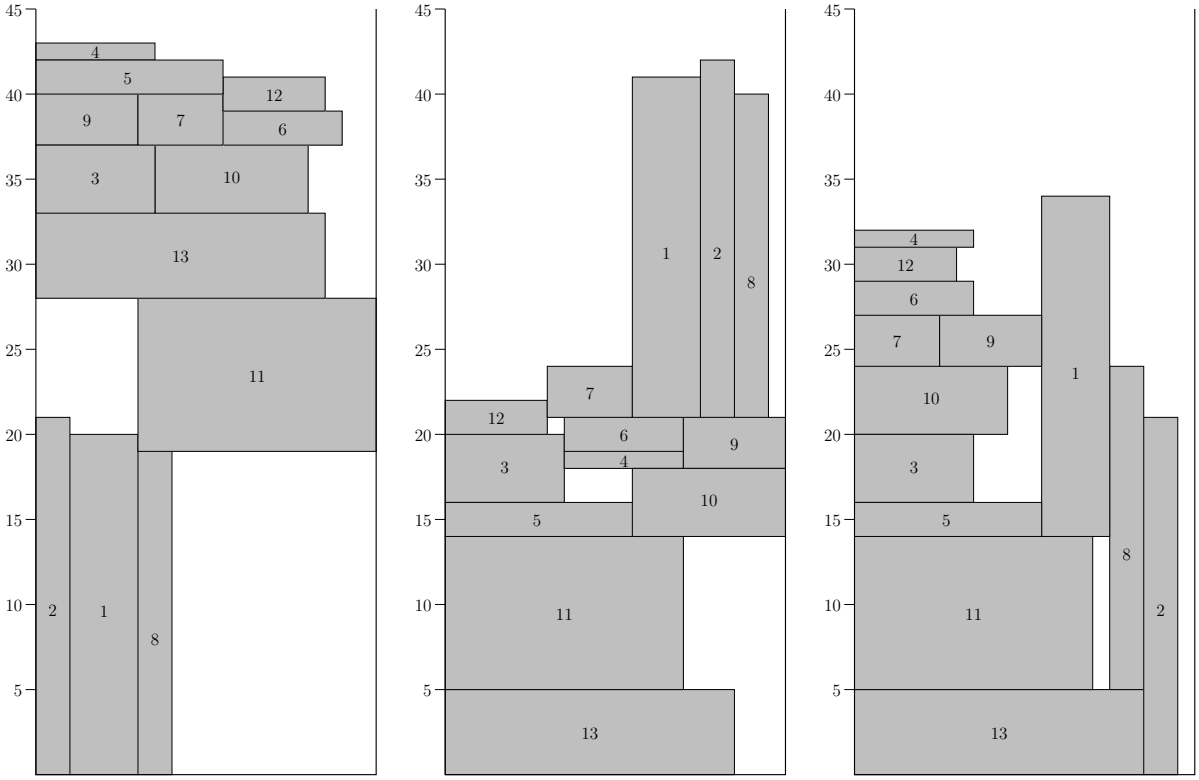
Next, consider the M algorithm. The items are assigned into groups, and each of these groups are sorted in their own manner. If the items were not sorted in the manner that the algorithm dictates, the resulting regions may not have the correct size, shape or location for the regions that are filled with items from subsequent groups. The SP algorithm requires that all items are no wider than the item preceding them in the packing order. If this were not the case, then it would be difficult to create regions when later items are wider than any of the regions.

5.2.2 Sorting-Independent Algorithms

The second class of algorithms is sorting-independent and includes the BL algorithm, Chazelle's algorithm, the GCS algorithm and the modified versions of the BF group of algorithms. These algorithms may be given a list of items that is sorted in any manner, including randomly sorted lists of items. While sorting the items according to increasing height or width has a high probability of yielding a bad result, the items may be sorted according to decreasing height, decreasing width or decreasing area, with ties resolved in any manner deemed appropriate. These algorithms may even be given lists of randomly arranged items in order to find the best packing after many packings of the same items in different orders, as MacLeod *et al.* [109] did for their cutting stock problem.

The fact that these algorithms can find a packing independently of the sorting order of the items allows one to experiment with the manner in which items are sorted. Lists that are sorted according to decreasing height may result in a packing that is sparse below a single wide item, thereby preventing further packing below it, particularly for the BL algorithm which does not allow packing to occur below a packed item, as Chazelle's algorithm and the GCS algorithm do. Lists sorted according to decreasing width may pack tall items last, yielding a packing with, say, one single pronounced vertical spike that results in a bad packing. If the item had been packed earlier, the resulting packing may have been lower. An attempt is made to clarify this point in Figure 5.23.

The solution shown in Figure 5.23(c) is the result of an attempt to rectify the problem encountered in the cases shown in Figure 5.23(a) and Figure 5.23(b), where sorting items according to decreasing height or width, respectively, yields an inefficient packing when the items are packed according to the BL algorithm. The idea is inspired by algorithms such as Sleator's algorithm [148], the SF algorithm by Coffman *et al.* [32], Golan's M algorithm [62] and the SAS algorithm by Ntene and Van Vuuren [125, 127]. In order to achieve this result, the items were first sorted according to decreasing width and then partitioned into two groups; those items $\mathcal{W}$ that are wider than half the strip width, and the remaining items $\mathcal{N}$. The list $\mathcal{W}$ is sorted according to decreasing width and the list $\mathcal{N}$ is sorted according to decreasing height, with ties



(a) DH ($\mathtt{H} = 43$)

(b) DW ($\mathtt{H} = 42$)

(c) $\frac{1}{2}$WDWDH ($\mathtt{H} = 34$)

Figure 5.23: *An illustration of a new sorting method for sorting independent algorithms; in this case the BL algorithm. Note that the items shown in these figures are not those listed in Table 3.1.*

resolved by sorting the items of equal height according to decreasing width. This sorting is denoted by $\frac{1}{2}$WDWDH (or 50WDWDH), where the fraction (or percentage) denotes the fraction (or percentage) of the strip width (W) at the splitting point, *i.e.* the width at which the two groups are separated. In this case, the items that are more than half (or 50%) the width of the strip width are sorted according to decreasing width (DW), and the remaining items are sorted according to decreasing height (DH). This may, for example, be changed to read $\frac{9}{20}$WDWDA which indicates that those items that have a width wider than 45% of the strip width are sorted according to decreasing width, while the remaining items are sorted according to decreasing area.

A natural modification would be to split the items into groups according to the number of items in the list. For example, one could sort the list of items by width and sort the last half of the items according to height. This would be denoted by $\frac{1}{2}$RDWDH or 50RDWDH, where the fraction or number represents the fraction or percentage of items that are, in this case, sorted by decreasing width. The remaining items would be sorted according to decreasing height in this example. The abbreviation R (for "rectangle") is used instead of I to prevent the possible confusion between the letter "I" and the number "1".

5.3 Chapter Summary

Previously known pseudolevel heuristics for the strip packing problem were reviewed in this chapter, in fulfilment of Dissertation Objective IV(c), as stated in §1.3. The algorithms were presented in chronological order, beginning with Sleator's algorithm in §5.1.1. This was followed by the split-fit algorithm in §5.1.2, for which two modifications were presented (one guaranteeing a guillotine solution, the other not), and the BL algorithm by Baker *et al.* [6]. Golan's two algorithms [62] were presented next, namely the split algorithm in §5.1.4, and the mixed algorithm in §5.1.5. Golan's algorithms were followed by the up-down algorithm in §5.1.6 and Chazelle's BLF algorithm in §5.1.7. The final two plane-packing algorithms considered in this chapter were the guillotine cutting stock algorithm in §5.1.8 and the best-fit algorithm (and its modifications) in §5.1.9. A summary of these algorithms may be found in Table 5.1.

Algorithm	Year	Source	G	Performance Bound	Complexity	H
Sleator	1980	[148]	×	$\texttt{S}(\mathcal{L}) \leq 2\,\texttt{OPT}(\mathcal{L}) + \frac{1}{2}h_{\text{tall}}$	$\mathcal{O}(n \log n)$	41
SF	1980	[32]	✓	$\texttt{SF}(\mathcal{L}) \leq \frac{3}{2}\,\texttt{OPT}(\mathcal{L}) + 2$	$\mathcal{O}(n^2)$	38
BL	1980	[6]	×	$\texttt{BL}(\mathcal{L}) \leq 3\,\texttt{OPT}(\mathcal{L})$	$\mathcal{O}(n^3)$	40
SP	1981	[62]	✓	$\texttt{SP}(\mathcal{L}) \leq 2\,\texttt{OPT}(\mathcal{L}) + 1$	$\mathcal{O}(n^2)$	40
M	1981	[62]	×	$\texttt{M}(\mathcal{L}) \leq \frac{4}{3}\,\texttt{OPT}(\mathcal{L}) + 7\frac{1}{18}$	$\mathcal{O}(n^3)$	42
UD	1981	[5]	×	$\texttt{UD}(\mathcal{L}) \leq \frac{5}{4}\,\texttt{OPT}(\mathcal{L}) + \frac{53}{8}H$	$\mathcal{O}(n^4)$	40
BLF	1983	[25]	×	unknown	$\mathcal{O}(n^2)$	34
GCS	1993	[109]	✓	unknown	$\mathcal{O}(n^3)$	34
BF	2004	[22]	×	unknown	$\mathcal{O}(n^2)$	29

Table 5.1: *A summary of the plane-packing heuristics considered in this chapter. The column labelled* G *indicates whether or not the solution is guaranteed to be a guillotine layout. The complexity column shows the worst-case time complexity if n items are packed, and the column labelled* H *contains the packing heights achieved by the algorithms for the simple example item set listed in Table 3.1.*

In an attempt to improve on the known plane-packing algorithms in fulfilment of Dissertation Objective V(c), the known algorithms of §5.1 were partitioned into two classes; sorting-dependent algorithms and sorting-independent algorithms. There is a possibility that the new sorting procedures discussed in §5.2.2 may improve the packing densities of the sorting-independent algorithms. This will be investigated in the following chapter.

CHAPTER 6

An Appraisal of the Strip Packing Algorithms

Contents

A set of benchmark problem instances used to test the algorithms of §3–§5 are presented in this chapter, followed by a summary and interpretation of the results obtained by the various strip packing algorithms when applied to these benchmark instances.

6.1 Benchmark Problem Instances

In order to evaluate the effectiveness of the new algorithms for the strip packing problem (see §3.3 and §4.3), the results obtained by these algorithms are compared to the results obtained by

applying the known algorithms (described in §3.2, §4.2 and §5.1) to a large number of benchmark problem instances. The benchmark problem instances used for this purpose are introduced in this section.

A number of repositories containing benchmark problem instances for various packing problems are available online. These include Beasley's *OR-library* [9], Cui's *CutWeb* [36], the DEIS Operations Reseach Group's library of instances [39], the EURO Special Interest Group on Cutting and Packing (ESICUP) repository [46], Fekete and Van der Veen's *PackLib*^{2} [50], Hifi's *library of instances* [71], the test instances by Scheithauer *et al.* [147] and the repository for strip packing problems by Van Vuuren and Ortmann [154]. The benchmark problem instances used in this dissertation to evaluate the strip packing algorithms are listed in Table 6.1.

Authors	Year	Reference	Number	Guillotineable	Optimal
Christofides & Whitlock	1977	[26]	3	Random	1 Known
Bengtsson	1982	[14]	10	Random	All Known
Beasley	1985	[7]	13	Random	2 Known
Beasley	1985	[8]	12	Random	All Known
Berkey & Wang	1987	[16]	300	Random	None known
Jakobs	1996	[83]	2	0 Guillotineable	Both Known
Dagli & Poshyanonda	1997	[38]	11	Random	None Known
Martello & Vigo	1998	[112]	200	Random	None Known
Ratanapan & Dagli	1998	[141]	1	Random	Not Known
Hifi	1998	[69]	25	Random	10 Known
Hifi	1999	[70]	9	Random	None Known
Burke & Kendall	1999	[21]	1	1 Guillotineable	Known
Hopper & Turton	2000	[75, 79]	21	14 Guillotineable	All Known
Hopper & Turton	2000	[75, 80]	70	35 Guillotineable	All Known
Wang & Valenzuela	2001	[156]	480	All	All Known
Burke, Kendall & Whitwell	2004	[22]	12	All	All Known
Total			1 170		621 Known

Table 6.1: *Benchmark problem instances used to evaluate the strip packing algorithms in §3, §4 and §5. The* Guillotineable *column indicates whether the benchmarks were designed such that an optimal packing can be disassembled by means of guillotine cuts. Ten of the benchmark sets [7, 8, 14, 16, 26, 38, 69, 70, 112, 141] were randomly generated subject to certain area and dimensional constraints, but not from an initial rectangle in the same manner that the the others [22, 75, 79, 80, 83, 156] were generated (which allows one to deduce an optimal packing). Known optimal solutions to some of these instances are listed by Martello et al. [110] and Kenmochi et al. [90].*

1977 Christofides and Whitlock (cgcut)

Christofides and Whitlock [26] generated their benchmark problem instances from an initial rectangle $\mathcal{R}_0$ of area $A(\mathcal{R}_0)$. A further m random rectangles $\mathcal{R}_1, \ldots, \mathcal{R}_m$ were generated by drawing $A(\mathcal{R}_i)$ from a uniform distribution in the range $(0, A(\mathcal{R}_0)/4)$ where $h(\mathcal{R}_i)$ is an integer from a uniform distribution in the range $(0, A(\mathcal{R}_i))$ and $w(\mathcal{R}_i) = \lceil A(\mathcal{R}_i)/h(\mathcal{R}_i) \rceil$. These rectangles were initially generated as a test case for the constrained cutting problem, where a limited number of each rectangle may be used to find a layout in a single bin that minimises the wasted area. These benchmark instances have previously been used in the context of the strip packing problem by Monaci [119], Martello *et al.* [110], Ntene [125], Alvarez-Valdes *et al.* [4], Bekrar and Kacem [10], Ntene and Van Vuuren [127], Wei *et al.* [158] and Kenmochi *et al.* [90], and were obtained from [46].

1982 Bengtsson (beng)

Bengtsson [14] generated his benchmark problem instances by taking rectangle lengths as the nearest integer of the form $12r + 1$ and widths equal to the nearest integer of the form $8r + 1$, where r is a random number drawn from a uniform distribution in the range $(0, 1)$. These instances have previously been used by various authors, including Monaci [119], Martello *et al.* [110], Ntene [125], Alvarez-Valdes *et al.* [4], Bekrar and Kacem [10], Wei *et al.* [158] and Kenmochi *et al.* [90] to test algorithms designed for the strip packing problem.

1985 Beasley (gcut and ngcut)

Beasley's first set of benchmark problem instances (gcut) [7] was generated in a manner similar to that of Christofides and Whitlock [26] and was intended for the unconstrained, two-dimensional, guillotine cutting problem. The height and width distributions are different to those employed by Christofides and Whitlock. The height is an integer taken from a uniform distribution in the range $[h(\mathcal{R}_0)/4, 3h(\mathcal{R}_0)/4]$ and the width is taken from a uniform distribution in the range $[w(\mathcal{R}_0)/4, 3w(\mathcal{R}_0)/4]$. The second set of benchmark problem instances (ngcut) [8] was generated using the same restrictions as implemented by Christofides and Whitlock, except that the height of a rectangle was taken as an integer from a uniform distribution in the range $[1, h(\mathcal{R}_0)]$. These benchmark instances have been used for the strip packing problem by various authors, including Monaci [119], Martello *et al.* [110], Ntene [125], Alvarez-Valdes *et al.* [4], Bekrar and Kacem [10], Ntene and Van Vuuren [127], Wei *et al.* [158] and Kenmochi *et al.* [90]. These benchmark instances were obtained from [46].

1987 Berkey and Wang

Berkey and Wang [16] generated their benchmark problem instances in order to test their algorithms for the single bin size bin packing problem. The items were generated in three groups with the heights and widths of rectangles randomly selected from a uniform distribution of integer values. The dimensions of the items in the first group were generated in the range $[1, 10]$, while the range for the second group was $[1, 35]$ and the range for the third group was $[1, 100]$. Items in the first group are to be packed into two bin sizes; namely 10×10 and 30×30 bins, while items in the second group are to be packed into bins of dimensions 40×40 and 100×100, and items in the third group were used to test the two cases where the bins have dimensions 100×100 and 300×300. This means that there are a total of six problem instances, each containing 100 items.

Martello and Vigo [112] expanded these problem instances for use in the context of bin packing algorithms. Ten sets of 20, 40, 60, 80 and 100 items were generated for each size range/bin size pair. This resulted in a total of 300 test instances. Monaci [119] adapted these instances for the strip packing problem by taking the strip width equal to the bin width. These instances have been used by Lodi *et al.* [107], Bortfeldt [18], Alvarez-Valdes *et al.* [4], Belov *et al.* [12] and Bekrar and Kacem [10] in the context of the strip packing problem, and were obtained from [39].

1996 Jakobs

Jakobs [83] began with a stock rectangle of height 15 and width 40, and randomly cut it into smaller pieces. One problem instance comprises 25 rectangles and the other 50 rectangles. Optimal solutions to these problem instances are not guillotineable. Monaci [119] and Bortfeldt

[18] have used these instances in the context of the strip packing problem. These benchmark instances were obtained from [46].

1997 Dagli and Poshyanonda

Dagli and Poshyanonda [38] and Ratanapan and Dagli [141] provide no detail regarding the generation of their problem instances and optimal solutions are not known for any of their benchmark problem instances, which were obtained from [46].

1998 Burke and Kendall

The benchmark instance of Burke and Kendall [21] is based on Figure 4 in the paper by Christofides and Whitlock [26, p. 42], which was declared an optimal cutting pattern for one of their problems. Burke and Kendall [21] simply doubled the dimensions of the items in the figure. This benchmark instance was obtained from [46].

1998 Martello and Vigo

Martello and Vigo [112] generated four classes of instances to accompany those of Berkey and Wang [16] described in §6.1. They generated ten problem instances for each value of $n \in \{20, 40, 60, 80, 100\}$ in each class, considering 100×100 bins for each class. The four classes of items were selected from four types of times. The first type of item was generated choosing a rectangle height from a uniform distribution in the range $[1, 50]$, and a rectangle width from the range $[67, 100]$. The heights for the second type of item was selected from a uniform distribution in the range $[67, 100]$ and the widths were selected from a uniform distribution in the range $[1, 50]$. For the third type the item heights were selected from a uniform distribution in the range $[50, 100]$ and widths from a uniform distribution in the range $[50, 100]$. The final type of item had heights selected from a uniform distribution in the range $[1, 50]$ and widths were selected from a uniform distribution in the range $[1, 50]$. The first class of items consists of 70% type 1 items and 10% each of the remaining types. The second class of items is comprised of 70% of type 2 items and 10% each of the other item types. The third class of items consists of 70% type 3 items and 10% of each of the remaining item types. Finally, the fourth class of items is comprised of 70% type 4 items and 10% of each of the remaining item types.

Monaci [119] adapted these instances for the strip packing problem by taking the strip width equal to 100 throughout. These instances have been used for the strip packing problem by Lodi *et al.* [107], Bortfeldt [18], Alvarez-Valdes *et al.* [4], Belov *et al.* [12] and Bekrar and Kacem [10], and were obtained from [39]; no optimal solutions are known.

1998 Hifi (SCP and SCPL)

Hifi does not give any details regarding the construction of the 25 problem instances (often labelled *SCP*) that he generated, other than to state that they are "random problem instances" [69, p. 935]. The second set (1999) was generated in a similar manner (often labelled *SCPL*) and are simply larger instances. These benchmark instances were obtained from Hifi [71] together the optimal packing heights for the SCP set of instances computed by him. Optimal solutions to the guillotine packing/cutting problem are known, but algorithms that do not adhere to the guillotine constraint may find a better packing solution. These instances have been used by Monaci [119] and Bekrar and Kacem [10] in the context of the strip packing problem.

2000 Hopper and Turton (T, N and C)

The methods used by Hopper [75] and by Hopper and Turton [79,80] to generate their benchmark problem instances are described in detail by Hopper [75]. Three algorithms were used to generate each problem instance subset. The first algorithm chooses a rectangle, assigns a random point in the rectangle, then splits the rectangle into four parts by means of a horizontal cut and a vertical cut through the point. The second algorithm randomly chooses an edge of an initial rectangle, assigns a random point on the edge, mirrors the point on the opposite edge and splits the rectangle though the line between the two points. Finally, the non-guillotineable benchmark generator selects an initial large rectangle, randomly assigns two points in the rectangle and generates a pattern of five smaller rectangles in a non-guillotine manner. Half of the instances in the larger benchmark set [75, 80] were generated by one of the guillotineable algorithms (the set often labelled *T*), and the other half by the non-guillotineable algorithm (the set often labelled *N*). The aspect ratios of rectangle $\mathcal{R}_i$ satisfies the constraint $1/7 \leq h(\mathcal{R}_i)/w(\mathcal{R}_i) \leq 7$. The smaller benchmark set [75, 79] (often labelled *C* or *CP*) consists of an equal number of instances generated by each of the three algorithms. Lesh *et al.* [99], Burke *et al.* [22, 23], Bortfeldt [18], Ntene [125], Belov *et al.* [12], Cui *et al.* [37], Alvarez-Valdes *et al.* [4], Ntene and Van Vuuren [127], Wei *et al.* [158] and Kenmochi *et al.* [90] have used these benchmark instances to test their algorithms for the strip packing problem and the optimal packing heights are known in each case.

2001 Wang and Valenzuela (Nice and Path)

Wang and Valenzuela's [156] benchmark generator allows one to place restrictions on the size (area ratio) and shape (aspect ratio) of rectangles generated. They randomly generated half of their benchmark instances (which they call the "pathological" set), placing restrictions on neither the aspect ratio, nor the area ratio. For the other half (the "nice" set), they enforced two constraints. The first is an area ratio constraint, restricting the largest rectangle to be no larger then 7 times the area of the smallest. The second is an aspect ratio restriction, requiring that $1/4 \leq h(\mathcal{R}_i)/w(\mathcal{R}_i) \leq 4$. In this manner they generated benchmark problem instances containing rectangles that are possibly either vastly different ("pathological" set) or fairly similar ("nice" set). One significant difference between this data set and the others, is that the item dimensions are all real numbers, while the other benchmark instances consist of rectangles having integer-valued dimensions. All the Wang and Valenzuela [156] benchmark instances have a strip width of 100 and optimal packing height of 100. Burke *et al.* [22, 23], Bortfeldt [18], Ntene [125], Alvarez-Valdes *et al.* [4] and Ntene and Van Vuuren [127] have made use of these benchmark instances in the context of the strip packing problem.

2004 Burke, Kendall and Whitwell

Burke *et al.* [22] used a benchmark generating algorithm that begins with a large rectangle (an optimal solution is therefore known), then randomly makes random horizontal or vertical cuts to randomly selected rectangles, such that some minimum dimension constraint is not violated. This process continues until the required number of rectangles has been produced. These benchmark instances have been used by Burke *et al.* [23], Ntene [125], Alvarez-Valdes *et al.* [4], Ntene and Van Vuuren [127] and Kenmochi *et al.* [90] in order to test their strip packing algorithms.

6.2 Results Obtained by Level-Packing Heuristics

In this section the results obtained via the level-packing algorithms will be presented. In the first subsection those algorithms are considered that pack items individually in the order that they are sorted. This includes the NFDH, FFDH, BFDH and WFDH algorithms and their variations. Thereafter, the results of those algorithms that perform further refinements (joining items, solving the knapsack problem or item substitution) are reported. Finally, a comparison of the best algorithms from each group is performed in order to find the best of the algorithms.

A few significance tests are used to test whether the new algorithms are significantly better than the algorithms from the literature. An analysis of variance[1] (ANOVA) may be performed in order to test whether the results obtained by algorithms are significantly different. The SAS software suite [145] is used to perform the ANOVA. However, the weakness of the ANOVA for these types of results is that it assumes that the data are normally distributed. This is not the case for the packing results, as may be seen in the box plots[2] in the remainder of this chapter. Therefore a nonparametric Friedman test [53], as recommended by Demšar [40], may be used to test for significance if the input is not normally distributed (it was implemented in MATLAB [113]). The Friedman test ranks the columns (algorithms) for each row (benchmark instance) and uses these ranks to test for significance.[3] Demšar goes on to recommend and describe the Nemenyi test as a post-hoc test for comparing all algorithms to one another, in order to discover which algorithms are not significantly different from one another. When the Friedman test has found the mean ranks of the algorithms, the Nemenyi test finds significance by means of a critical distance

$$CD = q_\alpha \sqrt{\frac{k\,(k+1)}{6N}}$$

between the ranks, where k is the number of columns (algorithms) and N is the number of rows (benchmark instances). The "critical values q_α are based on the Studentised range statistic divided by $\sqrt{2}$" [40, p. 12] and were provided to the author by Kourentzes [94]. The Nemenyi test was implemented in a spreadsheet and validated by means of a MATLAB implementation supplied by Kourentzes. For interest sake, the Bonferroni t test (as suggested by Nel [123])

[1] An analysis of variance may be used to compare the means of more than two data sets [73] in order to test whether these means are significantly different, and has previously been used to compare the results of strip packing algorithms by Ntene [125]. The F-distribution [73, Table A.3] is used for the ANOVA. A variance ratio F is calculated by dividing the variance between sets by the variance within sets. There are significant differences between the means when $F > F_c$, where F_c is the critical value at a chosen significance level (typically 95%). Many software suites may return a P-value (the significance level is $1 - P$), denoting the confidence with which $F > F_c$. The lower the value of P, the more significant the differences between the observed means.

[2] A box plot, first proposed by Tukey [152, pp. 39–41], is a method of graphically representing the distribution of a set of observations. In MATLAB [114], the box is created by drawing a line at the 25^{th} percentile (also called the first or lower quartile) and a line at the 75^{th} percentile (also called the third or upper quartile) and then joining the end points of these lines. Half of the points are distributed between these two lines and the difference between the upper and lower quartile is known as the *interquartile range* (IQR). A line is drawn through the box at the location of the median value. The lines extending past the box are known as *whiskers* and they are no longer than 1.5 times the IQR from the upper or lower quartile, and would not extend past the lowest or highest point if it fell within the range of the whisker. Any points that lie past the end of the whisker are called *outliers* as they would fall outside approximately 99.3% of the coverage if the data were normally distributed.

[3] The Friedman statistic,

$$\chi_F^2 = \frac{12N}{k(k+1)} \left[\sum_j \overline{r}_j^2 - \frac{k(k+1)^2}{4} \right],$$

is distributed according to the χ_F^2 distribution with $k-1$ degrees of freedom, where $\overline{r}_j$ is the mean rank for algorithm j, k is the number of columns (algorithms) and N is the number of rows (benchmark instances) [40, p. 11].

was implemented in SAS as the post-hoc test for the ANOVA for the packing heights and the computational solution times (the Tukey test [73] was also included for comparison purposes for the computational times on the suggestion of Lamont [96] and was also implemented in SAS). All tests were performed for a confidence interval of 95%.

6.2.1 The NFDH, FFDH, BFDH and WFDH Algorithms

In this subsection the results from the NFDH, FFDH, BFDH and WFDH are presented. These algorithms are grouped together, because they pack items in a very similar manner. The difference between the algorithms lies in the selection of the levels into which the items may be packed. The NFDH algorithms pack items into the topmost level only, the FFDH algorithms pack items into the lowest possible level, the BFDH algorithms pack items into the level with the least remaining horizontal space, and the WFDH algorithms pack items into the level with the largest remaining space.

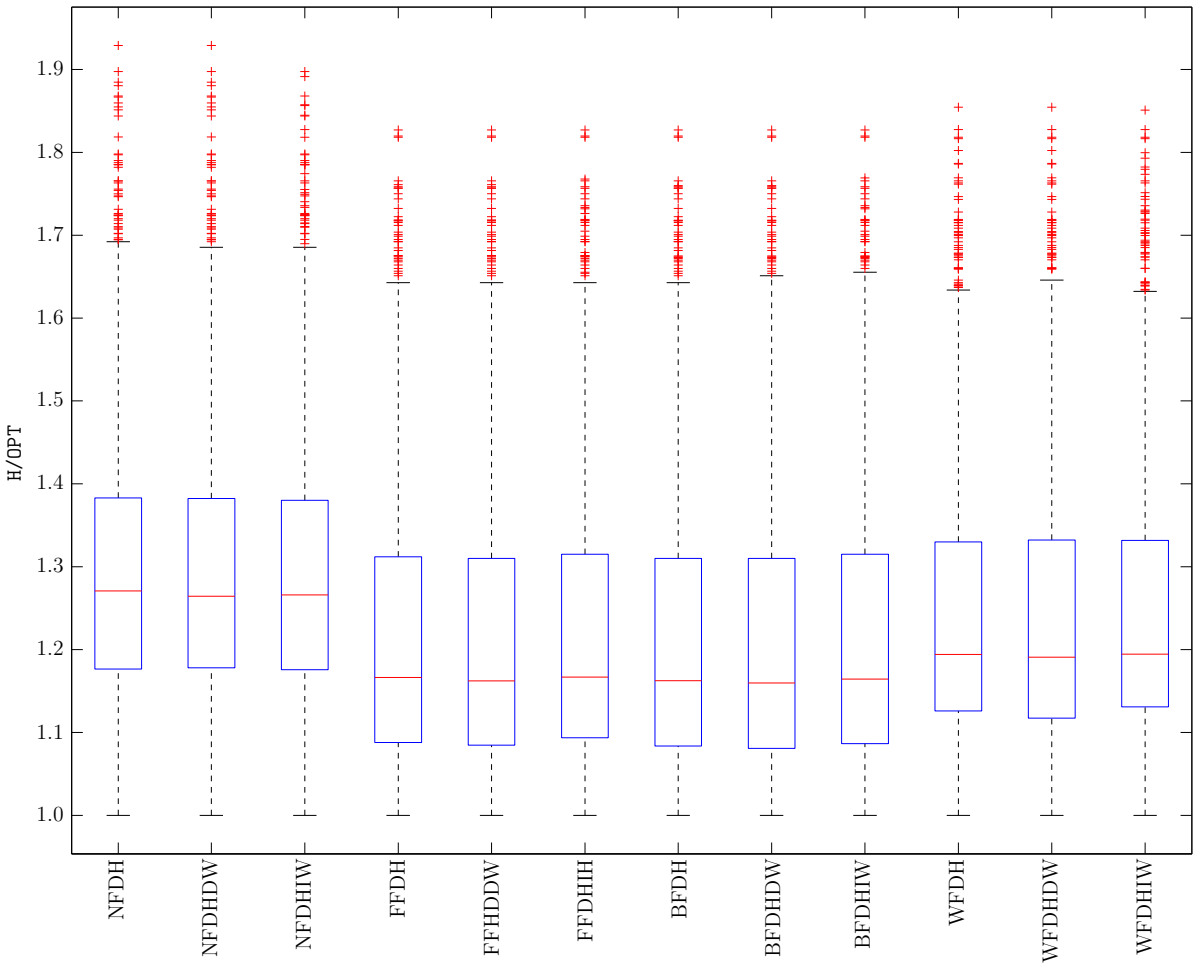


Figure 6.1: *Box plot of the distribution of results for the various NFDH, FFDH, BFDH and WFDH algorithms described in §3.2 and §3.3.1 when applied to the 1 170 strip packing problem benchmark instances described in §6.1.*

Figure 6.1 is a box plot of the solutions obtained by various algorithms. The bottom edge of the box is the location of the lower quartile and the top edge is located at the value of the upper quartile. The line passing through the box is the median of the packing heights relative to the

optimal packing height. Table 6.2 contains a summary of the results for these algorithms and their variations. It is clear from the box plots that the NFDH and WFDH algorithms pack the majority of benchmark instances to a higher relative packing height than do the FFDH and BFDH algorithms. This is also clear from Table 6.2, where the values of the lower quartiles of the NFDH and WFDH algorithms are higher than the corresponding values for the FFDH and BFDH algorithms. The median values of the BFDH and FFDH algorithms are lower than those of the other algorithms, as are the upper quartile values.

Applying ANOVA with a confidence level of 95% to the data yields $P < 0.0001$, suggesting that the null hypothesis of all algorithms being equal may be rejected. This is supported by the Friedman test which yields a P-value of 0. A Bonferroni t test performed on the four groups (NF, FF, BF and WF) indicates that the next-fit class of algorithms are significantly different from the other algorithms. The WFDH algorithms are the second worst group of algorithms in this set and the average packing heights of this class of algorithms are significantly different to the first-fit and best-fit algorithms (excluding the FFDHIW algorithm). The average rankings of the algorithms reflect the distributions shown in Figure 6.1. The Bonferroni t test indicates no significant difference between the six FFDH and BFDH algorithms (they all belong to group D). However, the BFDHDW has the lowest average ranking, followed by the BFDH and FFDH algorithms, which suggests that the BFDHDW algorithm may be the best choice of the algorithms reviewed in this section. This is supported by the nonparametric Nemenyi test, which ranks the BFDHDW algorithm as the best of the group, but with no significant difference to the FFDHDW and BFDH algorithms. In the row labelled *Sig. Class* in Table 6.2, the letters assigned to the algorithms indicate their rank. If the letters are different, then the Nemenyi test suggests that they are significantly different. The worst algorithm is assigned to class "A" and the algorithms are ranked alphabetically, the best algorithm in the group is labelled with the letter furthest from "A".

Performing the same significance test on the various algorithmic solution times for the benchmark instances with 5 000 items yields a significant difference ($P < 0.0001$ for only the "nice" items, only the "pathological" items and for the two sets combined at a confidence level of 95%) between the time it takes the NFDH algorithms to find solutions to such large problems, and the time it takes the FFDH, BFDH and WFDH algorithms to find solutions to the same problems (the time associated with the FFDH algorithm are not significantly different from the times required by the NFDHDW and NFDHIW algorithms). This result is expected due to the NFDH algorithms' better worst-case time complexity of $\mathcal{O}(n \log n)$ versus a worst-case time complexity of $\mathcal{O}(n^2)$ for the other algorithms in this set. The BFDHDW algorithm will be used for the purpose of further comparisons due to its performance in terms of packing height.

6.2.2 The KP Algorithms

In this subsection the results obtained via the KP and time-restricted KP algorithms described in §3.2.4 are presented. Fewer data sets are used for comparison purposes due to the time it requires to solve the integer programming knapsack problem. The nice data sets of 5 000 items by Wang and Valenzuela [156] are ignored, as is the twelfth instance by Burke *et al.* [22] of 500 items, due to the KP algorithm reaching the timeout restriction of an hour. Therefore 1 159 benchmark instances were used to compare these algorithms. A summary of the results may be found in Table 6.3.

The box plots of the corresponding results in Figure 6.2 do not show a clear difference between the solutions obtained by the various algorithms. The quartiles and medians listed in Table 6.3

	NFDH	NFDHDW	NFDHIW	FFDH	FFDHDW	FFDHIW	BFDH	BFDHDW	BFDHIW	WFDH	WFDHDW	WFDHIW
Low. Q. `H/OPT`	117.6%	117.8%	117.6%	108.8%	108.5%	109.4%	108.4%	108.1%	108.6%	112.6%	111.7%	113.1%
Med. `H/OPT`	127.1%	126.4%	126.6%	116.6%	116.2%	116.7%	116.3%	116.0%	116.4%	119.4%	119.1%	119.4%
Up. Q. `H/OPT`	138.3%	138.2%	138.0%	131.1%	131.0%	131.5%	131.0%	131.0%	131.5%	133.0%	133.2%	133.1%
IQR	20.7%	20.4%	20.4%	22.4%	22.5%	22.1%	22.6%	22.9%	22.9%	20.4%	21.5%	20.1%
Max. `H/OPT`	192.9%	192.9%	189.7%	182.7%	182.7%	182.7%	182.7%	182.7%	182.7%	185.4%	185.4%	185.1%
Mean Rank	10.40 (12)	10.36 (11)	10.25 (10)	4.26 (5)	3.92 (3)	4.81 (6)	3.78 (2)	3.56 (1)	4.26 (4)	7.54 (8)	7.17 (7)	7.70 (9)
Sig. Class	A(A)	A(A)	A(A)	E(CD)	EF(CD)	D(BCD)	EF(CD)	F(D)	E(CD)	BC(B)	C(BC)	B(B)
Nice 5 000 t	2.2964^{C}	2.2988^{BC}	2.2990^{BC}	2.3087^{AB}	2.3124^{A}	2.3132^{A}	2.3156^{A}	2.3164^{A}	2.3169^{A}	2.3175^{A}	2.3178^{A}	2.3179^{A}
Path 5 000 t	2.2820^{E}	2.2860^{D}	2.2859^{DE}	2.2923^{C}	2.2962^{BC}	2.2963^{B}	2.2984^{AB}	2.2992^{AB}	2.3003^{A}	2.3012^{A}	2.3007^{A}	2.3012^{A}
Bon. Class	C	BC	BC	AB	A	A	A	A	A	A	A	A

Table 6.2: *A summary of the results for the NFDH, FFDH, BFDH and WFDH algorithms and their variations described in Chapter 3 when applied to the* 1 170 *strip packing benchmark problem instances described in §6.1. The row labelled* 'Median `H/OPT`' *contains the median packing height for all benchmark instances listed in Table 6.1 as a percentage of the optimal packing height, or its lower bound if the optimal is not known. The row labelled* 'Low. Q. `H/OPT`' *contains the value of the lower quartile, the row labelled* 'Up. Q. `H/OPT`' *contains the values of the upper quartile and the interquartile range (in the row labelled* 'IQR'*) is the difference between the two. The row labelled* 'Max. `H/OPT`' *contains the worst result achieved by the algorithms for all benchmark instances. The row labelled* 'Sig. Class' *are results obtained by means of a Nemenyi test, with the results from a Bonferroni t test in parentheses. Algorithms in the same group (indicated by letters) do not produce results that are significantly different. The row labelled* 'Bon. Class' *contains results obtained by means of a Bonferroni t test on the average times required to solve all instances of* 5 000 *items. The row labelled* 'Mean Rank' *contains the mean rank achieved by the algorithms in this set (a rank of 1 indicates that the algorithm packed to the lowest height for an instance), with their ranks shown in parentheses. If algorithms yielded the same packing height for an instance, the mean of the ranks that would have been awarded is used. The rows labelled* 'Nice 5 000 t' *and* 'Path 5 000 t' *show the time (in seconds) required for instances of* 5 000 *items (for the "nice" and "pathological" benchmark problem instances [156]), with results from the Bonferroni t test for each set presented as superscripts..*

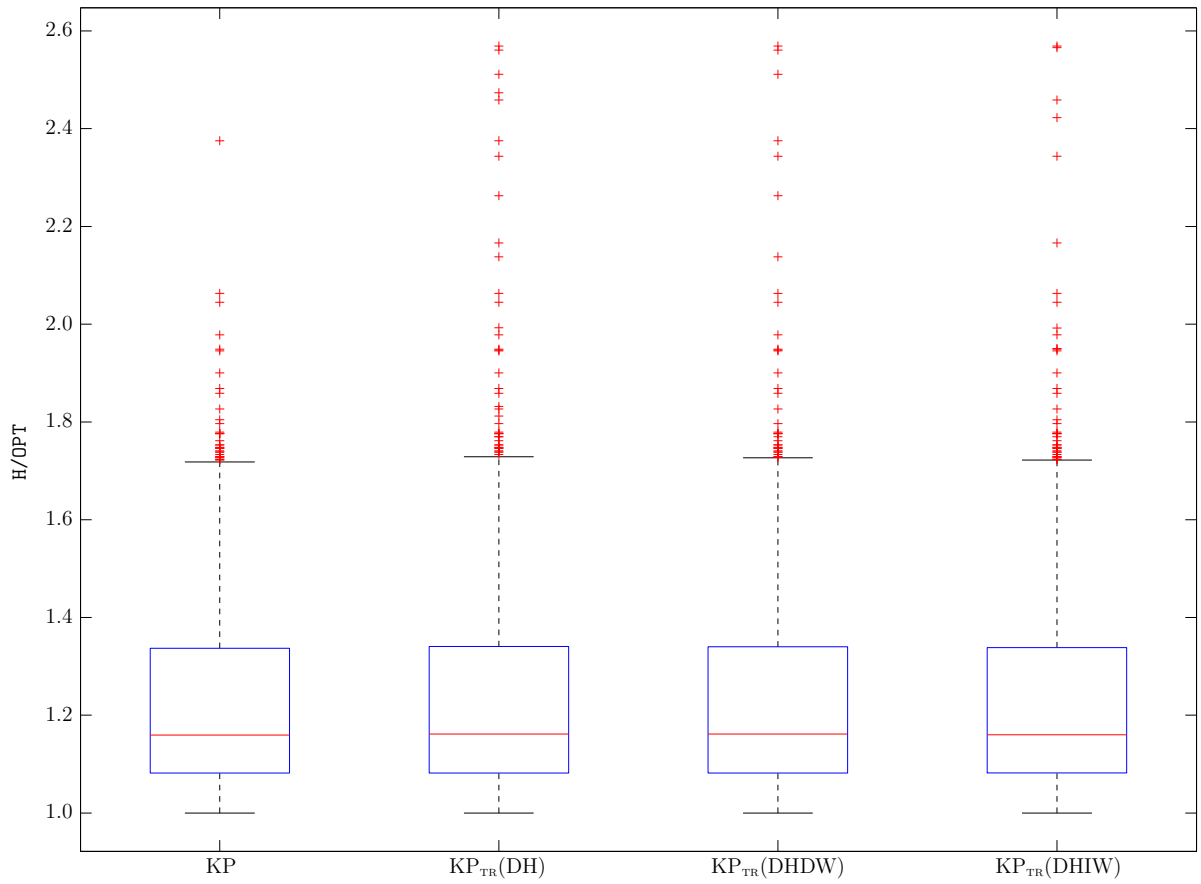


Figure 6.2: *Box plot of the distribution of results for the KP algorithm and the time-restricted KP algorithms described in §3.2.4 when applied to the 1 170 strip packing problem benchmark instances described in §6.1.*

also do not yield figures that indicate that any of these algorithms is better than the others. The KP algorithm shows a small advantage in average rank, but an ANOVA yields $P = 0.6563$, suggesting that the differences in average height are not significant at a confidence of 95%. However, the Friedman test yields $P = 1.51 \times 10^{-11}$, suggesting that there is a significant difference between the algorithms. The Nemenyi test suggests that there is a significant difference between the KP and KP$_{\text{TR}}$DHDW algorithms, but the test is not strong enough to determine whether the other two algorithms are significantly different from either the KP or KP$_{\text{TR}}$DHDW algorithms.

There is a significant difference between the KP algorithm and the time-restricted algorithms in terms of time taken to solve problems containing 2 000 items ($P < 0.0001$ for only the "nice" items, only the "pathological" items and for the two sets combined at a confidence level of 95%), with the time-restricted algorithms showing a clear advantage over the original algorithm. One would expect this for large data sets, but for small data sets the one second time restriction on the solution of the knapsack problem does not make a difference to the times. In fact, there are 673 benchmark instances (345 of 50 items or fewer, 283 of more than 50 items but fewer than 100 items, 37 of between 100 and 200 items and 8 of between 200 and 500 items) for which the KP algorithm was faster than the average of the time-restricted algorithms. The fact that there are cases where the KP algorithm finds solutions in less time than the time-restricted algorithms may be attributed to the fact that the time-restricted algorithms attempt

	KP	$KP_{TR}DH$	$KP_{TR}DHDW$	$KP_{TR}DHIW$
Low. Q. `H/OPT`	108.2%	108.2%	108.2%	108.2%
Med. `H/OPT`	115.9%	116.2%	116.2%	116.0%
Up. Q. `H/OPT`	133.7%	134.0%	134.0%	133.8%
IQR	25.5%	25.9%	25.8%	25.6%
Max. `H/OPT`	237.5%	256.9%	256.9%	256.9%
Mean Rank	2.40 (1)	2.53 (3)	2.55 (4)	2.51 (2)
Nem. Class	B	AB	A	AB
Nice 2 000 t	355.24^{A}	43.484^{B}	43.449^{B}	43.540^{B}
Path 2 000 t	140.91^{A}	31.000^{B}	30.965^{B}	31.194^{B}
Bon. Class	A	B	B	B

Table 6.3: *A summary of the results for the KP algorithm and its variations described in Chapter 3 when applied to the* 1 170 *strip packing benchmark problem instances described in §6.1. The row labelled* 'Median `H/OPT`' *contains the median packing height for all benchmark instances listed in Table 6.1 as a percentage of the optimal packing height, or its lower bound if the optimal is not known. The row labelled* 'Low. Q. `H/OPT`' *contains the value of the lower quartile, the row labelled* 'Up. Q. `H/OPT`' *contains the values of the upper quartile and the interquartile range (in the row labelled* 'IQR'*) is the difference between the two. The row labelled* 'Max. `H/OPT`' *contains the worst result achieved by the algorithms for all benchmark instances. The row labelled* 'Nem. Class' *are results obtained by means of a Nemenyi test. Algorithms in the same group (indicated by letters) do not produce results that are significantly different. The row labelled* 'Bon. Class' *contains results obtained by means of a Bonferroni t test on the average times required to solve all instances of* 2 000 *items. The row labelled* 'Mean Rank' *contains the mean rank achieved by the algorithms in this set (a rank of 1 indicates that the algorithm packed to the lowest height for an instance), with their ranks shown in parentheses. If algorithms yielded the same packing height for an instance, the mean of the ranks that would have been awarded is used. The rows labelled* 'Nice 2 000 t' *and* 'Path 2 000 t' *show the time (in seconds) required for instances of* 2 000 *items (for the "nice" and "pathological" benchmark problem instances [156]). The 10 benchmark instances containing* 5 000 *"nice" items by Wang and Valenzuela [156] and the 500-item instance by Burke et al. [22] are excluded because the KP algorithm timed out for these.*

to find an approximate solution to the knapsack problem before the problem is solved as an integer programming problem. The KP algorithm does not require this step as it only exits the solver if it requires more than an hour to solve the knapsack problem. Because of the risk of having to solve large problems and their superior time performance for these problems, any of the time-restricted algorithms may be used for further comparisons with other strip packing heuristics.

6.2.3 The JOIN Algorithms

The results from the JOIN algorithms described in §3.2.5 are presented in this subsection. A Friedman test on the data yields $P = 0$ (and an ANOVA yields $P < 0.0001$), suggesting that all algorithms do not necessarily perform similarly. If one compares the upper and lower parts of Table 6.4, or the left and right halves of Figure 6.3 it becomes clear that the horizontal joining of items is typically the more successful joining method of the two. Not only are the median values lower for the algorithms (with the same δ value) that join the items horizontally, but so too are the third quartile values and the interquartile ranges (IQR). The difference between the horizontal and vertical joining strategies is duplicated in the row indicating significantly different groups. The algorithms that join items horizontally and the algorithms that join items vertically for $\delta = 0$ are all not significantly different, but they are significantly different to the algorithms that join items vertically and have $\delta \geq 5$. The box plots indicate that an increase

in the value of δ causes a shift in the majority of the results further away from the optimum. This is common to all sorting strategies.

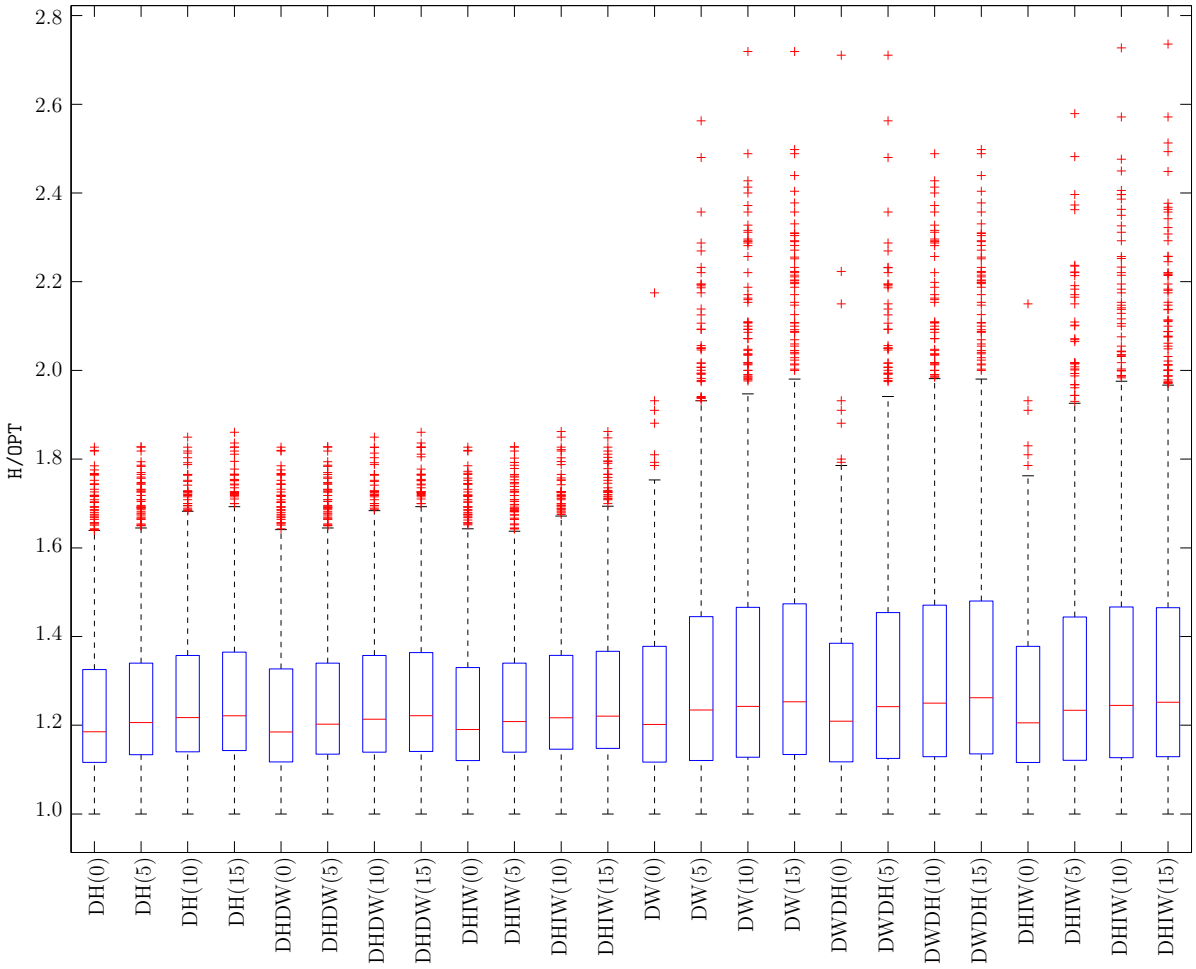


Figure 6.3: *Box plot of the distribution of results for the JOIN algorithms described in §3.2.5 when applied to the 1 170 strip packing problem benchmark instances described in §6.1.*

This observation is confirmed by the values of the mean ranks. The lower the value of δ, the lower the mean rank for the same sorting type. If one compares the mean ranks for algorithms with the same values for δ, the average rank is always lower for algorithms that join items horizontally than for algorithms that join items vertically. Therefore, one may deduce that an algorithm that would pack to the lowest height most consistently is likely to be an algorithm which joins items horizontally and uses $\delta = 0$. Of the three algorithms that conform to these restrictions, the JOIN(DHDW) algorithm achieves the best mean rank, and the JOIN(DHIW) algorithm achieves the worst mean rank. The Nemenyi test suggests that the three horizontally joining JOIN algorithms for $\delta = 0$ are not significantly different.

Performing an ANOVA on the large instances yields $P < 0.0001$ for only the "nice" item instances, only the "pathological" item instances and for the two sets combined at a confidence level of 95%, suggesting that the null hypothesis of all algorithms requiring similar times to find solutions may be rejected. The time required to solve the 5 000-item benchmark instances is not significantly different between the algorithms that join items horizontally. However, the improved packing achieved by algorithms with $\delta = 0$ and which join items vertically is at the cost of significantly longer solution times according to the Bonferroni and Tukey post-hoc

	DH(0)	DH(5)	DH(10)	DH(15)	DHDW(0)	DHDW(5)	DHDW(10)	DHDW(15)	DHIW(0)	DHIW(5)	DHIW(10)	DHIW(15)
Low. Q. H/OPT	111.7%	113.4%	114.0%	114.3%	111.7%	113.5%	113.9%	114.1%	112.1%	113.9%	114.6%	114.8%
Med. H/OPT	118.6%	120.6%	121.7%	122.1%	118.5%	120.2%	121.4%	122.2%	119.0%	120.8%	121.7%	122.1%
Up. Q. H/OPT	132.5%	134.0%	135.7%	136.5%	132.7%	134.0%	135.6%	136.4%	133.0%	134.0%	135.8%	136.6%
IQR	20.9%	20.6%	21.7%	22.2%	20.9%	20.5%	21.7%	22.2%	20.9%	20.1%	21.2%	21.9%
Max. H/OPT	182.7%	182.8%	185.0%	186.1%	182.7%	182.8%	185.0%	186.1%	182.7%	182.8%	186.2%	186.2%
Mean Rank	7.07 (2)	10.82 (8)	12.72 (11)	13.70 (16)	6.95 (1)	10.77 (7)	12.65 (10)	13.62 (15)	7.97 (3)	11.46 (9)	13.00 (12)	13.78 (17)
Sig. Class	K(B)	GI(B)	EF(B)	DEF(B)	K(B)	GIJ(B)	F(B)	DEF(B)	K(B)	G(B)	EF(B)	DE(B)
Nice 5 000 t	$2.3129^{\mathrm{C–F}}$	2.2834^{F}	2.3245^{CD}	2.2963^{DEF}	2.3166^{CDE}	$2.3013^{\mathrm{C–F}}$	$2.3282^{\mathrm{CD}}_{\mathrm{C}}$	2.2856^{EF}	2.3168^{CDE}	$2.3017^{\mathrm{C–F}}$	2.3288^{C}	2.2860^{EF}
Path 5 000 t	2.2775^{C}	2.2815^{C}	2.2784^{C}	2.2803^{C}	2.2797^{C}	2.2848^{C}	2.2840^{C}	2.2849^{C}	2.2809^{C}	2.2855^{C}	2.2854^{C}	2.2855^{C}
Bon. Class	C(CD)	C(D)	C(CD)	C(CD)	C(CD)	C(CD)	C(CD)	C(CD)	C(CD)	C(CD)	C	C(CD)

	DW(0)	DW(5)	DW(10)	DW(15)	DWDH(0)	DWDH(5)	DWDH(10)	DWDH(15)	DWIH(0)	DWIH(5)	DWIH(10)	DWIH(15)
Low. Q. H/OPT	111.7%	112.1%	112.8%	113.4%	111.8%	112.5%	112.9%	113.5%	111.6%	112.1%	112.7%	112.9%
Med. H/OPT	120.2%	123.4%	124.2%	125.3%	120.9%	124.2%	125.0%	126.2%	120.5%	123.4%	124.5%	125.2%
Up. Q. H/OPT	137.7%	144.5%	146.6%	147.4%	138.5%	145.3%	147.1%	147.9%	137.8%	144.4%	146.7%	146.5%
IQR	26.0%	32.4%	33.8%	34.0%	26.7%	32.8%	34.1%	34.4%	26.1%	32.3%	34.0%	33.6%
Max. H/OPT	217.5%	256.3%	271.9%	271.9%	271.1%	271.1%	248.9%	249.8%	215.0%	257.9%	272.7%	273.6%
Mean Rank	9.73 (4)	13.52 (13)	15.14 (20)	16.05 (23)	10.24 (6)	14.26 (18)	15.78 (22)	16.68 (24)	9.83 (5)	13.53 (14)	15.08 (19)	15.64 (21)
Sig. Class	J(B)	DEF(A)	BC(A)	AB(A)	IJ(A)	CD(A)	AB(A)	A(A)	IJ(A)	DEF(A)	BC(A)	AB(A)
Nice 5 000 t	3.6412^{A}	2.7964^{B}	2.8139^{B}	2.8001^{B}	3.6391^{A}	2.8066^{B}	2.8042^{B}	2.7841^{B}	3.6366^{A}	2.8071^{B}	2.8057^{B}	2.7848^{B}
Path 5 000 t	3.6048^{A}	2.7834^{B}	2.7841^{B}	2.7801^{B}	3.6089^{A}	2.7886^{B}	2.7891^{B}	2.7854^{B}	3.6086^{A}	2.7887^{B}	2.7897^{B}	2.7856^{B}
Bon. Class	A	B	B	B	A	B	B	B	A	B	B	B

Table 6.4: *A summary of the results for the JOIN algorithms and their variations described in §3.2.5 when applied to the* 1 170 *strip packing benchmark problem instances described in §6.1. The upper (lower) set of results are for the JOIN algorithm in which the adjacent items with similar heights (widths) are joined horizontally (vertically). The row labelled* 'Median H/OPT' *contains the median packing height for all benchmark instances listed in Table 6.1 as a percentage of the optimal packing height, or its lower bound if the optimal is not known. The row labelled* 'Low. Q. H/OPT' *contains the value of the lower quartile, the row labelled* 'Up. Q. H/OPT' *contains the values of the upper quartile and the interquartile range (in the row labelled* 'IQR') *is the difference between the two. The row labelled* 'Max. H/OPT' *contains the worst result achieved by the algorithms for all benchmark instances. The row labelled* 'Sig. Class' *are results obtained by means of a Nemenyi test, with the results from a Bonferroni t test in parentheses. Algorithms in the same group (indicated by letters) do not produce results that are significantly different. The row labelled* 'Bon. Class' *contains results obtained by means of a Bonferroni t test on the average times required to solve all instances of* 5 000 *items. The row labelled* 'Mean Rank' *contains the mean rank achieved by the algorithms in this set (a rank of 1 indicates that the algorithm packed to the lowest height for an instance), with their ranks shown in parentheses. If algorithms yielded the same packing height for an instance, the mean of the ranks that would have been awarded is used. The rows labelled* 'Nice 5 000 t' *and* 'Path 5 000 t' *show the time (in seconds) required for instances of* 5 000 *items (for the "nice" and "pathological" benchmark problem instances [156]), with results from the Bonferroni t test for each set presented as superscripts (results from the Tukey test appear as subscripts if the two tests differ).*

tests.[4] The algorithms joining items vertically are significantly slower than those that join items horizontally, strengthening the argument that the best JOIN algorithm is one that joins items horizontally.

6.2.4 The B2F Algorithms

Results obtained via the B2F algorithms described in §3.3.2 are presented in this subsection. The results are shown in Tables 6.5 and 6.6, which were not treated separately when determining whether differences in packing heights and solution times for instances of 5 000 items are significant. A box plot representation of the solution data may be found in Figures 6.4 and 6.5. These figures show that the results obtained via the various B2F algorithms are very similar. A Friedman test on the results yields $P = 0$, suggesting that there are significant differences between some algorithms. Applying an ANOVA to the results yields $P = 0.5934$, which would suggest that the algorithms are not significantly different, but the Friedman test is better suited to the nature of these results and therefore more likely to be accurate.

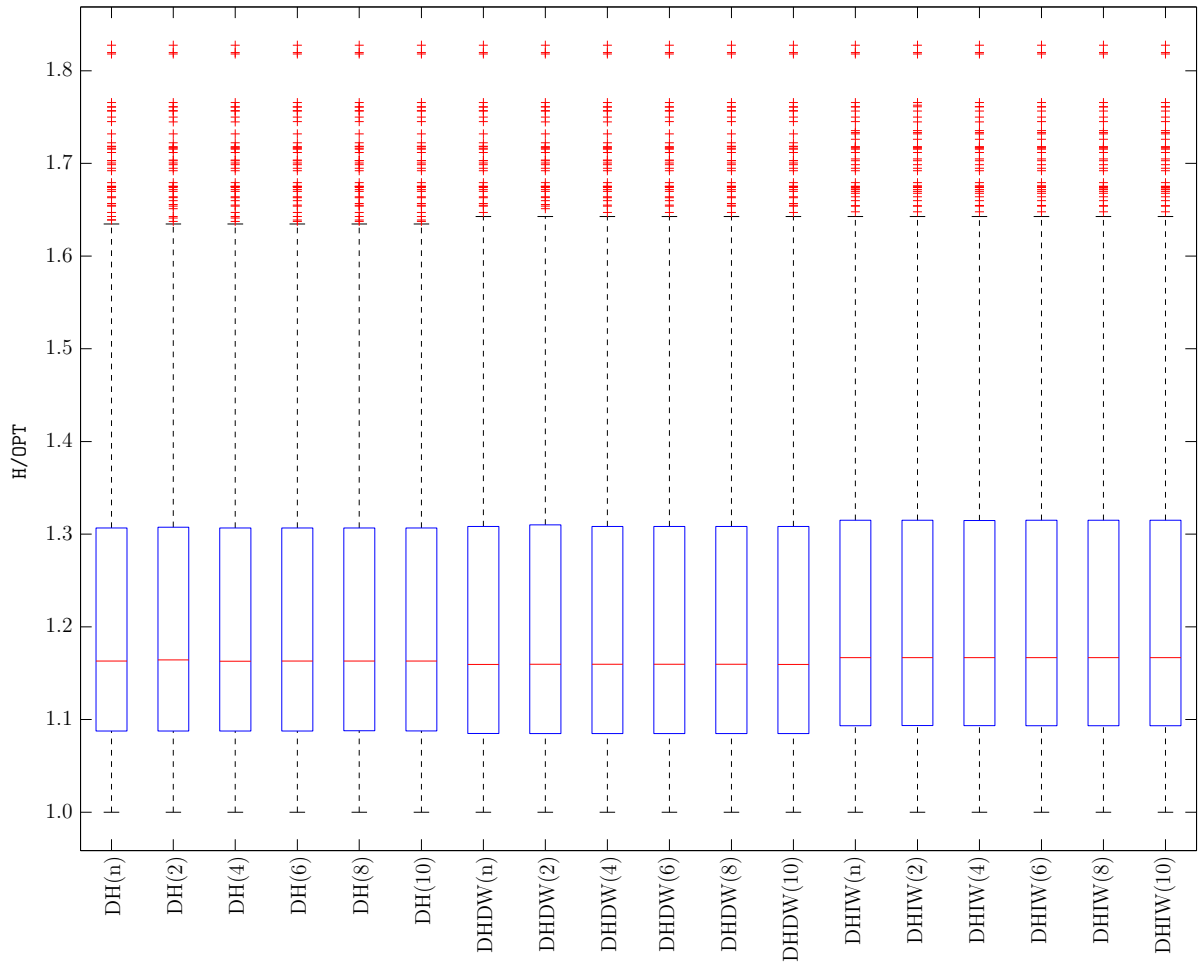


Figure 6.4: *Box plot of the distribution of results for the B2FA algorithms described in §3.3.2 when applied to the 1 170 strip packing problem benchmark instances described in §6.1.*

[4] These two tests did yield one difference; the Bonferroni t test suggests that the $\text{JOIN}_{10}\text{DHDW}$ and $\text{JOIN}_{15}\text{DH}$ algorithms require equivalent times to find solutions for "nice" data, while the Tukey test suggests that these solution times are not equivalent.

	DH(n)	DH(2)	DH(4)	DH(6)	DH(8)	DH(10)
Low. Q. `H/OPT`	108.8%	108.8%	108.8%	108.8%	108.8%	108.8%
Med. `H/OPT`	116.3%	116.4%	116.3%	116.3%	116.3%	116.3%
Up. Q. `H/OPT`	130.6%	130.7%	130.6%	130.6%	130.6%	130.6%
IQR	21.9%	22.0%	21.9%	21.9%	21.8%	21.9%
Max. `H/OPT`	182.8%	182.8%	182.8%	182.8%	182.8%	182.8%
Mean Rank	14.23 (8)	14.94 (12)	14.62 (11)	14.37 (10)	14.37 (9)	14.22 (7)
Nem. Class	JKL	J	JK	JKL	JKL	JKLM
Nice 5 000 t	19.1291^{A}	2.1593^{C}	2.1947^{C}	2.2037^{C}	2.2384^{C}	2.2603^{C}
Path 5 000 t	5.9836^{B}	2.1593^{C}	2.1816^{C}	2.1765^{C}	2.1985^{C}	2.2073^{C}
Bon. Class	A	B	B	B	B	B

	DHDW(n)	DHDW(2)	DHDW(4)	DHDW(6)	DHDW(8)	DHDW(10)
Low. Q. `H/OPT`	108.5%	108.5%	108.5%	108.5%	108.5%	108.5%
Med. `H/OPT`	115.9%	116.0%	116.0%	116.0%	116.0%	115.9%
Up. Q. `H/OPT`	130.8%	131.0%	130.8%	130.8%	130.8%	130.8%
IQR	22.3%	22.5%	22.3%	22.3%	22.3%	22.3%
Max. `H/OPT`	182.8%	182.8%	182.8%	182.8%	182.8%	182.8%
Mean Rank	12.73 (2)	13.39 (6)	13.08 (5)	12.90 (4)	12.84 (3)	12.69 (1)
Nem. Class	LM	JKLM	KLM	LM	LM	M
Nice 5 000 t	19.1329^{A}	2.1623^{C}	2.1979^{C}	2.2070^{C}	2.2423^{C}	2.2636^{C}
Path 5 000 t	5.9757^{B}	2.1616^{C}	2.1845^{C}	2.1792^{C}	2.2018^{C}	2.2107^{C}
Bon. Class	A	B	B	B	B	B

	DHIW(n)	DHIW(2)	DHIW(4)	DHIW(6)	DHIW(8)	DHIW(10)
Low. Q. `H/OPT`	109.3%	109.4%	109.3%	109.3%	109.3%	109.3%
Med. `H/OPT`	116.7%	116.7%	116.7%	116.7%	116.7%	116.7%
Up. Q. `H/OPT`	131.5%	131.5%	131.4%	131.5%	131.5%	131.5%
IQR	22.2%	22.1%	22.1%	22.2%	22.2%	22.2%
Max. `H/OPT`	182.8%	182.8%	182.8%	182.8%	182.8%	182.8%
Mean Rank	17.40 (13)	17.48 (17)	17.55 (18)	17.46 (16)	17.40 (14)	17.43 (15)
Nem. Class	I	I	I	I	I	I
Nice 5 000 t	18.5926^{A}	2.1637^{C}	2.2034^{C}	2.2077^{C}	2.2402^{C}	2.2621^{C}
Path 5 000 t	5.5692^{B}	2.1646^{C}	2.1861^{C}	2.1808^{C}	2.2021^{C}	2.2093^{C}
Bon. Groups	A	B	B	B	B	B

Table 6.5: *A summary of the results for the variations of the B2FA algorithm described in §3.3.2 when applied to the* 1 170 *strip packing benchmark problem instances described in §6.1. The row labelled* 'Median `H/OPT`' *contains the median packing height for all benchmark instances listed in Table 6.1 as a percentage of the optimal packing height, or its lower bound if the optimal is not known. The row labelled* 'Low. Q. `H/OPT`' *contains the value of the lower quartile, the row labelled* 'Up. Q. `H/OPT`' *contains the values of the upper quartile and the interquartile range (in the row labelled* 'IQR'*) is the difference between the two. The row labelled* 'Max. `H/OPT`' *contains the worst result achieved by the algorithms for all benchmark instances. The row labelled* 'Nem. Class' *are results obtained by means of a Nemenyi test. Algorithms in the same group (indicated by letters) do not produce results that are significantly different. The row labelled* 'Bon. Class' *contains results obtained by means of a Bonferroni t test on the average times required to solve all instances of* 5 000 *items. The row labelled* 'Mean Rank' *contains the mean rank achieved by the algorithms in this set (a rank of 1 indicates that the algorithm packed to the lowest height for an instance), with their ranks shown in parentheses. If algorithms yielded the same packing height for an instance, the mean of the ranks that would have been awarded is used. The rows labelled* 'Nice 5 000 t' *and* 'Path 5 000 t' *show the time (in seconds) required for instances of* 5 000 *items (for the "nice" and "pathological" benchmark problem instances [156]), with results from a Bonferroni t test for each set presented as superscripts. Note that this table is not independent of Table 6.6. The algorithm rankings include those of the algorithms in Table 6.6.*

	DH(n)	DH(2)	DH(4)	DH(6)	DH(8)	DH(10)
Low. Q. H/OPT	110.0%	109.6%	110.0%	110.0%	110.0%	110.1%
Med. H/OPT	117.5%	116.7%	117.3%	117.5%	117.6%	117.6%
Up. Q. H/OPT	131.8%	131.6%	131.6%	132.0%	132.0%	131.9%
IQR	21.8%	22.0%	21.6%	22.0%	22.0%	21.9%
Max. H/OPT	188.8%	183.5%	182.8%	188.0%	188.0%	188.0%
Mean Rank	22.93 (31)	19.81 (20)	21.63 (26)	22.00 (28)	21.89 (27)	22.07 (29)
Nem. Class	BCD	GH	DEF	DEF	DEF	DEF
Nice 5 000 t	14.9448^{B}	2.1559^{C}	2.1880^{C}	2.1898^{C}	2.2205^{C}	2.2369^{C}
Path 5 000 t	9.5525^{A}	2.1644^{C}	2.1928^{C}	2.1926^{C}	2.2209^{C}	2.2330^{C}
Bon. Class	A	B	B	B	B	B

	DHDW(n)	DHDW(2)	DHDW(4)	DHDW(6)	DHDW(8)	DHDW(10)
Low. Q. H/OPT	109.8%	109.1%	109.6%	109.6%	109.8%	109.8%
Med. H/OPT	117.2%	116.3%	116.7%	116.9%	117.1%	117.1%
Up. Q. H/OPT	131.8%	131.4%	131.8%	132.0%	132.0%	131.9%
IQR	22.0%	22.3%	22.1%	22.4%	22.2%	22.1%
Max. H/OPT	188.0%	182.8%	182.8%	188.0%	188.0%	188.0%
Mean Rank	21.62 (25)	18.35 (19)	19.94 (21)	20.50 (22)	20.76 (23)	20.81 (24)
Nem. Class	DEF	HI	GH	FG	EFG	EFG
Nice 5 000 t	14.9535^{B}	2.1608^{C}	2.1918^{C}	2.1938^{C}	2.2241^{C}	2.2409^{C}
Path 5 000 t	9.5555^{A}	2.1695^{C}	2.1971^{C}	2.1965^{C}	2.2252^{C}	2.2371^{C}
Bon. Class	A	B	B	B	B	B

	DHIW(n)	DHIW(2)	DHIW(4)	DHIW(6)	DHIW(8)	DHIW(10)
Low. Q. H/OPT	110.6%	110.2%	110.6%	110.8%	110.7%	110.7%
Med. H/OPT	117.7%	117.3%	117.7%	117.7%	117.7%	117.7%
Up. Q. H/OPT	132.2%	131.8%	132.4%	132.6%	132.5%	132.5%
IQR	21.6%	21.6%	21.8%	21.8%	21.8%	21.8%
Max. H/OPT	182.8%	182.8%	182.8%	182.8%	182.8%	182.8%
Mean Rank	25.06 (36)	22.42 (30)	23.96 (32)	24.31 (33)	24.50 (35)	24.35 (34)
Nem. Class	A	CDE	ABC	AB	AB	AB
Nice 5 000 t	14.3837^{B}	2.1610^{C}	2.1924^{C}	2.1918^{C}	2.2210^{C}	2.2320^{C}
Path 5 000 t	9.9444^{A}	2.1706^{C}	2.1990^{C}	2.1982^{C}	2.2244^{C}	2.2396^{C}
Bon. Class	A	B	B	B	B	B

Table 6.6: *A summary of the results for the variations of the B2FW algorithm described in §3.3.2 when applied to the* 1 170 *strip packing benchmark problem instances described in §6.1. The row labelled* 'Median H/OPT' *contains the median packing height for all benchmark instances listed in Table 6.1 as a percentage of the optimal packing height, or its lower bound if the optimal is not known. The row labelled* 'Low. Q. H/OPT' *contains the value of the lower quartile, the row labelled* 'Up. Q. H/OPT' *contains the values of the upper quartile and the interquartile range (in the row labelled* 'IQR') *is the difference between the two. The row labelled* 'Max. H/OPT' *contains the worst result achieved by the algorithms for all benchmark instances. The row labelled* 'Nem. Class' *are results obtained by means of a Nemenyi test. Algorithms in the same group (indicated by letters) do not produce results that are significantly different. The row labelled* 'Bon. Class' *contains results obtained by means of a Bonferroni t test on the average times required to solve all instances of* 5 000 *items. The row labelled* 'Mean Rank' *contains the mean rank achieved by the algorithms in this set (a rank of 1 indicates that the algorithm packed to the lowest height for an instance), with their ranks shown in parentheses. If algorithms yielded the same packing height for an instance, the mean of the ranks that would have been awarded is used. The rows labelled* 'Nice 5 000 t' *and* 'Path 5 000 t' *show the time (in seconds) required for instances of* 5 000 *items (for the "nice" and "pathological" benchmark problem instances [156]), with results from a Bonferroni t test for each set presented as superscripts. Note that this table is not independent of Table 6.5. The algorithm rankings include those of the algorithms in Table 6.5.*

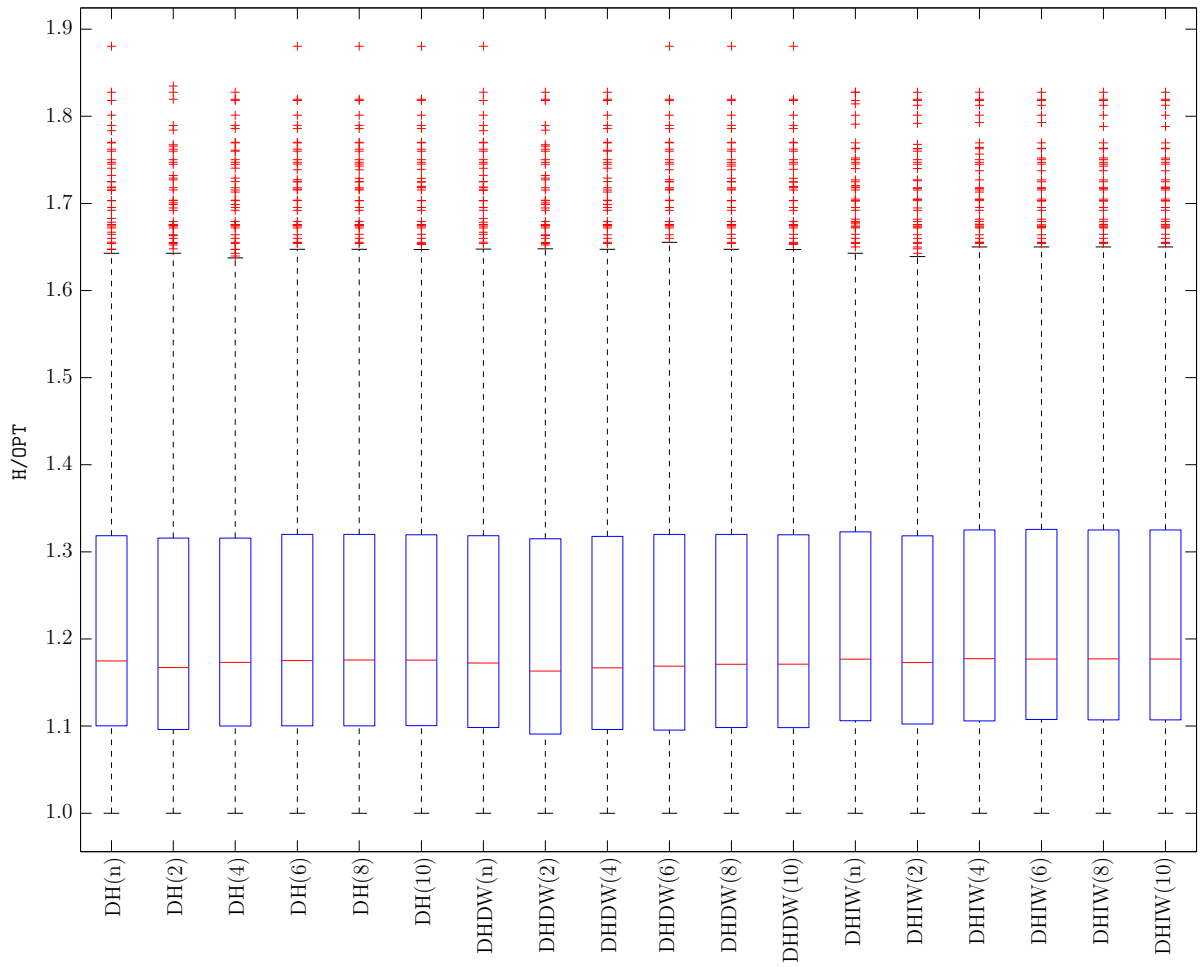


Figure 6.5: *Box plot of the distribution of results for the B2FW algorithms described in §3.3.2 when applied to the 1 170 strip packing problem benchmark instances described in §6.1.*

By grouping together the algorithms using the same sorting method, applying ANOVA to the data yields a P-value of 0.0068, indicating that there is a significant difference between the sorting groups. Performing Bonferroni t tests on the groups suggests that there is a significant difference between the DHIW and DHDW sorting methods, but that there is no significant difference between the algorithms that sort according to the DH method and the other two methods. If one creates two sets, one for the algorithms that replace items according to area and another set for the algorithms that replace items according to their width, and performs an ANOVA on these two groups, a P-value smaller than 0.0001 is found, suggesting that the algorithms which replace items according to their width are, on average, better than the algorithms that replace items according to their width. Combining all algorithms according to the number of items that may be replaced and applying an ANOVA yields a P-value of 0.9905, suggesting that the number of items that may be searched does not result in a significant difference with respect to the mean packing height.

The mean ranks appear to support some of the findings. The maximum mean rank for algorithms which replace items according to area (B2FA algorithms) is 17.55, achieved by the $\text{B2FA}_6\text{DHIW}$ algorithm, while the minimum mean rank for the B2F algorithms which replace items according to their width (B2FW algorithms) is 18.35, achieved by the $\text{B2FW}_2\text{DHDW}$ algorithm. The minimum mean rank may be attributed to the $\text{B2FA}_{10}\text{DHDW}$ algorithm, while

the maximum mean rank of 25.06 may be attributed to the B2FW$_n$DHIW algorithm. The mean ranks may be compared between sorting types. The average of the mean ranks for the B2FADH algorithms is 14.46; for the B2FADHDW algorithms it is 12.94 and for the B2FADHIW algorithms it is 14.95. A similar pattern emerges for the B2FW algorithms. The average for the mean ranks of the B2FWDH algorithms is 21.72; for the B2FWDHDW algorithms it is 20.33 and for the B2FWDHIW algorithms it is 24.10.

The times that the algorithms require to pack 5 000 items are significantly different (applying an ANOVA yields $P < 0.0001$ for only the "nice" item instances, only the "pathological" item instances and for the two sets combined at a confidence level of 95%), but only between those algorithms that allow all items to be searched and the algorithms that restrict the search to a limited number of items. The algorithms that allow all items to be searched do not require significantly different times to find solutions to the large benchmark instances. The algorithms that restrict the search to a limited number of items do not require a significantly different execution time. Interestingly, the B2F$_n$ algorithms require significantly more computational time than the B2FW$_n$ algorithms for "nice" instances, but are significantly faster for "pathological" instances. The results in Tables 6.5 and 6.6 show a large time difference between the algorithms that allow unrestrained searching, and the others. Therefore, the results in Tables 6.5 and 6.6 suggest that the best algorithm (in terms of packing height) will sort items according to decreasing height and decreasing width, compare items according to their areas, and restrict the search of the unpacked items to a limited number. The B2FA$_{10}$DHDW algorithm has the lowest mean rank and is therefore used in further comparisons.

6.2.5 A Comparison of the Level-Packing Algorithms

All the best algorithms of the previous subsections are compared in this subsection in order to identify the best level-packing algorithm. A comparison of the algorithms in the first subsection revealed that the BFDHDW algorithm yielded the best results most often. In the subsection on the KP algorithms the original algorithm was found to be too slow for large data sets, while the KP$_{\text{TR}}$DHIW achieved the second lowest mean rank; therefore it will represent that set of algorithms in the remainder of the chapter. An analysis of the JOIN algorithms led to the conclusion that the JOIN$_0$(DH) and JOIN$_0$(DHDW) algorithms are the best in the set, but only the JOIN$_0$(DHDW) algorithm will represent that set of algorithms in the remainder of the chapter as it yielded the lowest mean rank. A comparison of the B2F algorithms led to the conclusion that the B2FA$_{10}$DHDW algorithm would represent that set. The results are plotted as box plots in Figure 6.6 and are also listed in Table 6.7. A Friedman test yields $P = 0$ and an ANOVA on the results yields $P = 0.0031$, suggesting that the null hypothesis (that all algorithms are equivalent) may be rejected.

The KP$_{\text{TR}}$DHIW algorithm is the worst algorithm in this set of representative algorithms, with the largest IQR, the highest upper quartile and the worst maximum packing height. The Nemenyi test suggests that it is significantly worse than two of the algorithms in this set, but significantly better in terms of packing height in comparison to the JOIN algorithm. It is also close to 40 times slower than the other algorithms for benchmark instances containing 5 000 items. The JOIN algorithm is ranked as the worst algorithm in terms of packing height, achieving the lowest mean packing height and is significantly worse according to the Nemenyi test. However, it is not significantly worse than the BFDHDW and B2FA$_{10}$DWDH algorithms in terms of packing time. The BFDHDW and B2FA$_{10}$DWDH algorithms are neither significantly different in terms of packing height nor solution time for large instances and are the best level-packing algorithms of those tested in this dissertation.

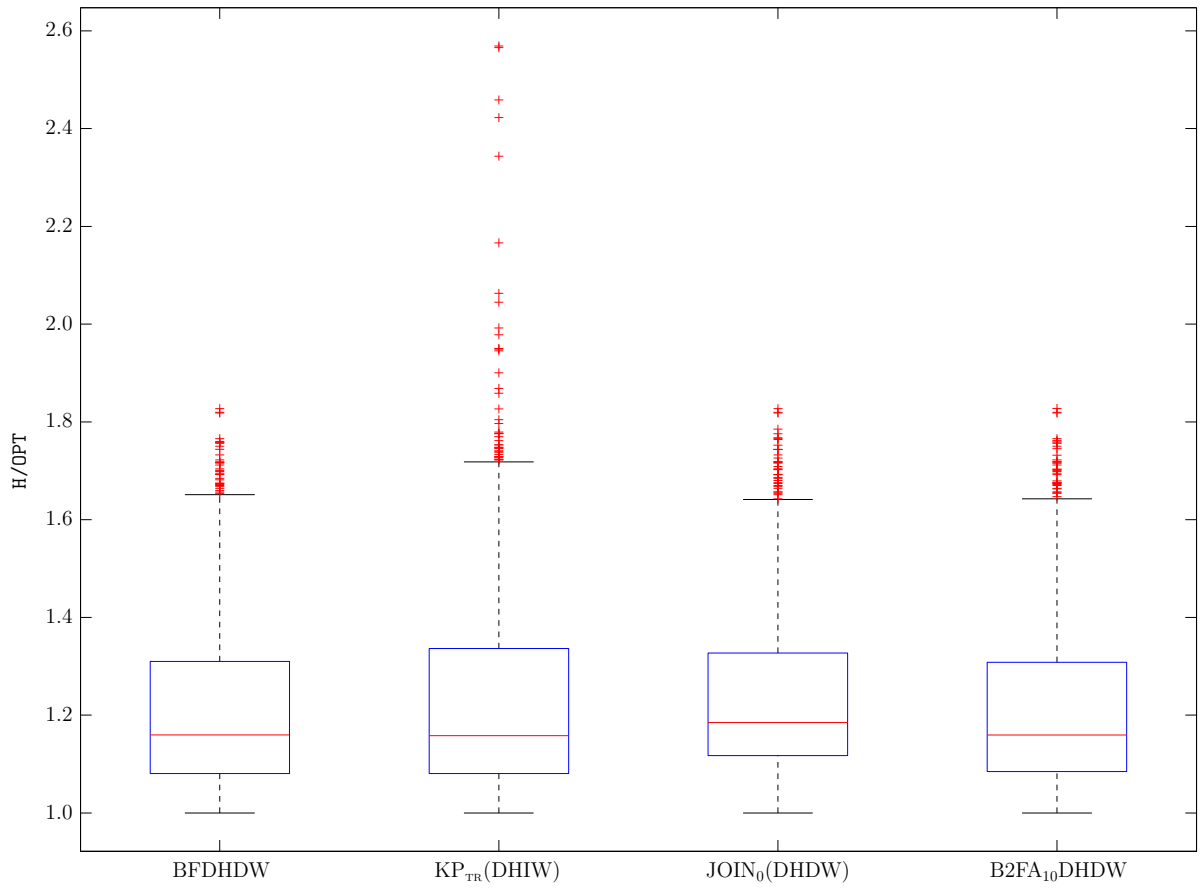


Figure 6.6: *Box plots of the results for the best level-packing algorithms described in §3.2 when applied to the 1 170 strip packing problem benchmark instances described in §6.1.*

6.3 Results Obtained by Pseudolevel-Packing Heuristics

The results obtained by applying the pseudolevel-packing algorithms to the benchmark problem instances in §6.1 are presented in this section. In the first subsection the known BFDH* and FC_{OG} algorithms and their variations, the SAS algorithm and the new algorithms that yield guillotine packings, namely the SASm, BFS and SL algorithms are considered. This is followed by a comparison of the free-packing pseudolevel algorithms; namely the FC_{OF}, SC and SCR algorithms.

6.3.1 Guillotine Pseudolevel Heuristics

In this subsection the FC_{OG} algorithm by Lodi *et al.* [106], as described in §4.2.1, and the BFDH* algorithm by Bortfeldt [18], as described in §4.2.2 are compared to the SAS algorithm by Ntene and Van Vuuren [125, 127] and the new algorithms presented in §4.3. Four variations of the SL algorithm are considered, each for different values of δ which are shown in parenthesis in the first row of Table 6.8. The value of δ determines the maximum height difference allowed between two items for them to be considered joined for stacking purposes. Figure 6.7 contains box plots of the results achieved by these algorithms and further results are tabulated in Table 6.8. A Friedman test on the packing heights results in $P = 0$, suggesting that the algorithms are significantly different in terms of packing height. A Nemenyi test on the results yield a critical distance of 0.53. An ANOVA performed on the results yields $P < 0.0001$.

	BFDHDW	KP_{TR}DHIW	$JOIN_0$(DHDW)	$B2FA_{10}$DHDW
Low. Q. H/OPT	108.1%	108.1%	111.7%	108.5%
Med. H/OPT	116.0%	115.8%	118.5%	115.9%
Up. Q. H/OPT	131.0%	133.6%	132.7%	130.8%
IQR	22.9%	25.5%	20.9%	22.3%
Max. H/OPT	182.7%	256.9%	182.7%	182.8%
Mean Rank	2.02 (1)	2.84 (3)	3.05 (4)	2.09 (2)
Sig. Class	C(B)	B(AB)	A(A)	C(B)
Nice 5 000 t	2.3164^{B}	76.475^{A}	2.3166^{B}	2.2636^{B}
Path 5 000 t	2.2992^{B}	81.418^{A}	2.2797^{B}	2.2107^{B}
Bon. Class	B	A	B	B

Table 6.7: *A summary of the results for the best level algorithms described in Chapter 3 when applied to the* 1 170 *strip packing benchmark problem instances described in §6.1. The row labelled* 'Median H/OPT' *contains the median packing height for all benchmark instances listed in Table 6.1 as a percentage of the optimal packing height, or its lower bound if the optimal is not known. The row labelled* 'Low. Q. H/OPT' *contains the value of the lower quartile, the row labelled* 'Up. Q. H/OPT' *contains the values of the upper quartile and the interquartile range (in the row labelled* 'IQR'*) is the difference between the two. The row labelled* 'Max. H/OPT' *contains the worst result achieved by the algorithms for all benchmark instances. The row labelled* 'Sig. Class' *are results obtained by means of a Nemenyi test, with the results from a Bonferroni t test in parentheses. Algorithms in the same group (indicated by letters) do not produce results that are significantly different. The row labelled* 'Bon. Class' *contains results obtained by means of a Bonferroni t test on the average times required to solve all instances of* 5 000 *items. The row labelled* 'Mean Rank' *contains the mean rank achieved by the algorithms in this set (a rank of 1 indicates that the algorithm packed to the lowest height for an instance), with their ranks shown in parentheses. If algorithms yielded the same packing height for an instance, the mean of the ranks that would have been awarded is used. The rows labelled* 'Nice 5 000 t' *and* 'Path 5 000 t' *show the time (in seconds) required for instances of* 5 000 *items (for the "nice" and "pathological" benchmark problem instances [156]), with results from a Bonferroni t test for each set presented as superscripts.*

The SASm algorithm is a significant improvement on the SAS algorithm, the median packing height is approximately 2% better for the new algorithm than for the original. The box plot in Figure 6.7 suggests that the distribution of the relative packing heights for the modified algorithm is closer to the optimal packing height for the new SAS algorithm than for the original by Ntene [125,127]. The new algorithm finds the minimum packing height between the two more often (a mean ranking of 10.70 versus 9.39), but both variations of the SAS algorithm result in the worst packing height of the algorithms listed in Table 6.8.

The BFDH* algorithms are better in terms of the distribution of their relative packing heights, with the BFDH*(DW) as the best of the three variations. It achieves lower first quartile, median and third quartile values than the other two BFDH* algorithms. This better performance is mirrored in the mean ranking, which is significantly lower for the BFDH*(DW) algorithm, compared to the BFDH* and BFDH*(IW) algorithms. However, these algorithms are worse than the FC_{OG} algorithms, which result in a better distribution of packing heights for the 1 170 benchmark instances used in this dissertation. The results for the FC_{OG} algorithms show lower median, upper quartile and IQR values and this is mirrored by the lower mean ranking values for the FC_{OG} algorithms compared to those of the BFDH* rankings. Once again the DHDW sorting procedure proves to result in the best packing heights, on average.

However, the SAS, BFDH* and FC_{OG} algorithms perform worse in terms of packing density when compared to the stacking algorithms, namely the BFS and SL algorithms. Of these stacking algorithms the SL algorithm with $\delta = 5$ proves to achieve the lowest packing height of these algorithms most consistently (as suggested by the mean rank values). Allowing floor-

	$FC_{OG}DH$	$FC_{OG}DHDW$	$FC_{OG}DHIW$	BFDH*	BFDH*(DW)	BFDH*(IW)	SAS
Low. Q. `H/OPT`	106.6%	106.3%	107.0%	107.5%	107.1%	107.8%	110.9%
Med. `H/OPT`	110.7%	110.4%	111.0%	113.3%	112.8%	113.4%	116.5%
Up. Q. `H/OPT`	118.5%	118.5%	118.6%	123.1%	122.8%	123.5%	125.2%
IQR	12.0%	12.2%	11.7%	15.6%	15.7%	15.7%	14.3%
Max. `H/OPT`	167.8%	167.8%	168.6%	176.6%	176.6%	176.6%	195.1%
Mean Rank	5.68 (5)	5.16 (2)	6.19 (8)	8.73 (10)	8.32 (9)	9.17 (11)	10.70 (13)
Sig. Class	EFG(CD)	GH(CDE)	E(C)	CD(B)	D(B)	BC(B)	A(A)
Nice 5 000 t	4.8808^{B}	4.9004^{B}	4.9284^{B}	5.9963^{A}	5.9884^{A}	6.0012^{A}	1.3351^{D}
Path 5 000 t	5.7305^{A}	5.7298^{A}	5.7347^{A}	5.3191^{B}	5.3378^{B}	5.3077^{B}	1.4400^{F}
Bon. Class	C(B)	BC(B)	ABC(AB)	AB(A)	A	AB(A)	E(D)

	SASm	BFS	SL_0	SL_5	SL_{10}	SL_{15}
Low. Q. `H/OPT`	109.2%	106.2%	106.0%	105.9%	106.2%	106.3%
Med. `H/OPT`	114.2%	110.1%	109.6%	109.4%	109.5%	109.8%
Up. Q. `H/OPT`	122.8%	117.3%	116.5%	116.2%	116.0%	116.4%
IQR	13.7%	11.1%	10.5%	10.3%	9.7%	10.1%
Max. `H/OPT`	195.1%	169.2%	169.2%	169.2%	169.2%	169.2%
Mean Rank	9.39 (12)	6.16 (7)	5.39 (4)	4.96 (1)	5.33 (3)	5.82 (6)
Sig. Class	B(B)	E(DCE)	FGH(DE)	H(E)	FGH(E)	EF(DCE)
Nice 5 000 t	1.0615^{E}	2.3812^{C}	2.3498^{C}	2.3281^{C}	2.3241^{C}	2.3096^{C}
Path 5 000 t	1.1679^{F}	2.8200^{C}	2.6935^{CD}	2.3707^{DE}	2.3741^{DE}	2.3469^{E}
Bon. Class	E(D)	D(C)	D(C)	D(C)	D(C)	D(C)

Table 6.8: *A summary of the results for new and known guillotine pseudolevel algorithms and their variations described in Chapter 4 when applied to the* 1 170 *strip packing benchmark problem instances described in §6.1. The row labelled* 'Median `H/OPT`' *contains the median packing height for all benchmark instances listed in Table 6.1 as a percentage of the optimal packing height, or its lower bound if the optimal is not known. The row labelled* 'Low. Q. `H/OPT`' *contains the value of the lower quartile, the row labelled* 'Up. Q. `H/OPT`' *contains the values of the upper quartile and the interquartile range (in the row labelled* 'IQR') *is the difference between the two. The row labelled* 'Max. `H/OPT`' *contains the worst result achieved by the algorithms for all benchmark instances. The row labelled* 'Sig. Class' *are results obtained by means of a Nemenyi test, with the results from a Bonferroni t test in parentheses. Algorithms in the same group (indicated by letters) do not produce results that are significantly different. The row labelled* 'Bon. Class' *contains results obtained by means of a Bonferroni t test (results obtained by means of a Tukey test are shown in parentheses if they differ) on the average times required to solve all instances of* 5 000 *items. The row labelled* 'Mean Rank' *contains the mean rank achieved by the algorithms in this set (a rank of 1 indicates that the algorithm packed to the lowest height for an instance), with their ranks shown in parentheses. If algorithms yielded the same packing height for an instance, the mean of the ranks that would have been awarded is used. The rows labelled* 'Nice 5 000 t' *and* 'Path 5 000 t' *show the time (in seconds) required for instances of* 5 000 *items (for the "nice" and "pathological" benchmark problem instances [156]), with results from a Bonferroni t test for each set presented as superscripts.*

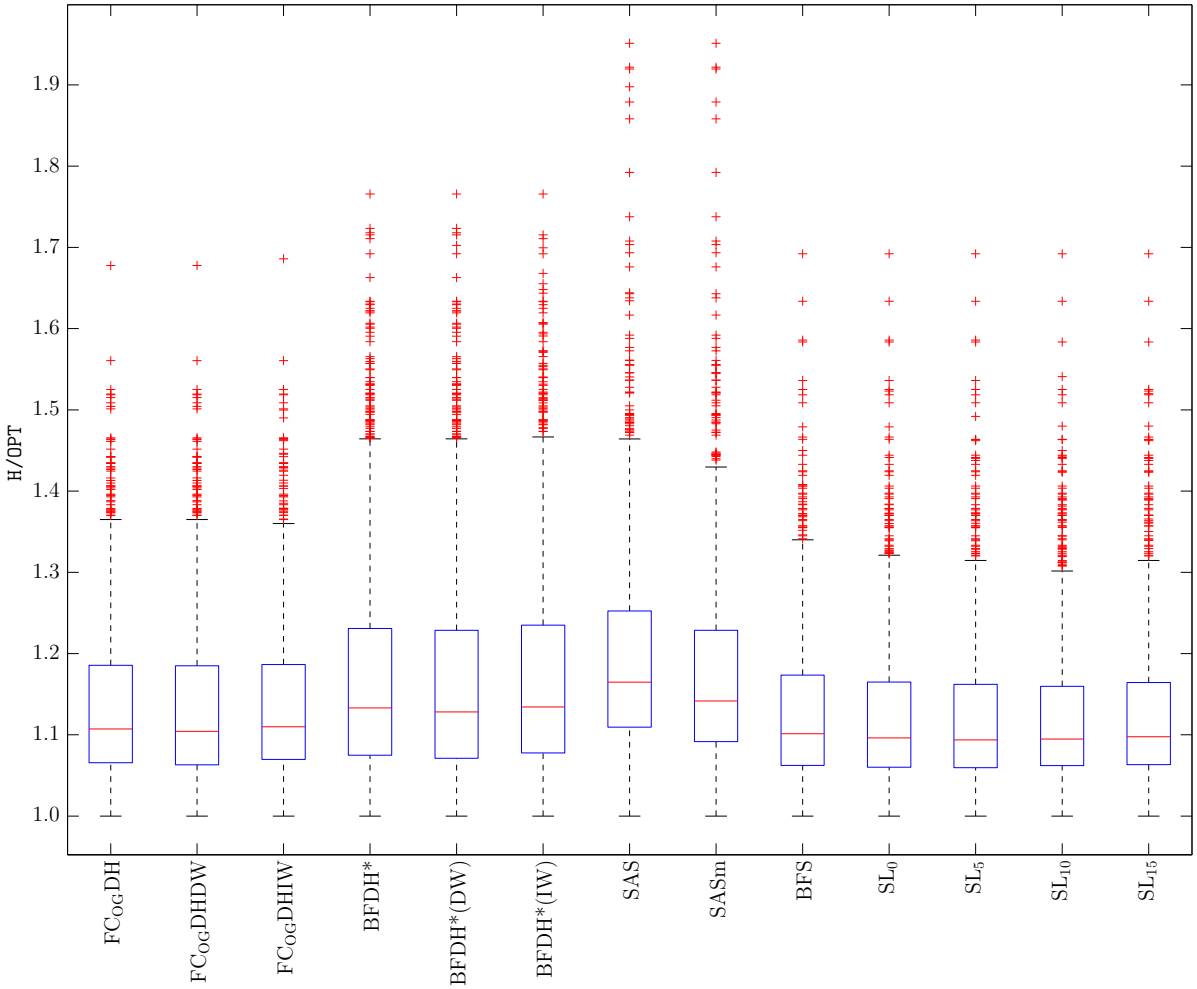


Figure 6.7: *Box plot of the distribution of results for the guillotine pseudolevel algorithms described in §4.2 and §4.3 when applied to the 1 170 strip packing problem benchmark instances described in §6.1.*

packed items to be joined for the purposes of stacking means that the SL algorithms perform better than the BFS algorithm, on average, resulting in a distribution of relative packing heights that is closer to the optimum for the SL algorithms than for the BFS algorithm. The mean ranks suggest that the SL_5 algorithm is not significantly better than the $FC_{OG}DHDW$ algorithm.

Performing an ANOVA (at a 95% confidence level) on the solution times required by the algorithms to solve benchmark instances containing 5 000 items yields $P < 0.0001$ when applied to only "nice" instances, when applied to only "pathological" benchmark instances, and when applied to both sets of benchmark instances, suggesting that the algorithmic times are significantly different.[5] Performing a Bonferroni t test on the algorithmic times yields results that suggest that the BFDH* algorithms are slower than the FC_{OG} algorithms for "pathological" items (the roles are reversed for "nice" items), which are significantly slower than the BFS and SL algorithms. However, the test suggests that the SAS algorithms are significantly faster than the BFS and SL algorithms, sacrificing packing density for speed. The fact that the SL algo-

[5]The results from the Bonferroni t test and the Tukey test when applied to the combined sets of benchmark instances appear very different at first glance, but the only difference is that the Tukey test found a significant difference between the algorithmic times of the $FC_{OG}DHDW$ algorithm and the BFDH* and BFDH*IW algorithms, which the Bonferroni t test did not find.

rithms are significantly faster and yield a better density than the FC_{OG} and BFDH* algorithms (against which the SL_5 algorithm is significantly better), suggests that the SL_5 algorithm would be the best choice of algorithms from this set, except if the speed of the SASm algorithm is preferred over the packing density of the SL_5 algorithm.

6.3.2 Non-Guillotine Pseudolevel Heuristics

In this subsection the single known non-guillotine pseudolevel algorithm, the FC_{OF} algorithm and its variations (see §4.2.1), is compared to the two new non-guillotine algorithms, the SC and SCR algorithms described in §4.3.4. Table 6.9 contains a summary of the results of applying these algorithms to the 1 170 benchmark instances and Figure 6.8 contains box plots of the results obtained by the algorithms. Three variations of the FC_{OF} algorithm are considered; all three sort items by decreasing height, but one variant does not resolve equalities in height (DH), one resolves the equalities by sorting them according to decreasing width and the final variant resolves ties by sorting the items according to increasing width. Performing an ANOVA on these algorithms' results yields $P = 0.1448$, suggesting that the results are not significantly different. However, performing a nonparametric Friedman test yields $P = 0$, suggesting that the results are significantly different. Due to the fact that the parametric ANOVA assumes a normal distribution of results, which does not necessarily apply to the packing height results, suggests that the nonparametric Friedman test is better suited to test the significance of the differences in results. The critical distance for the Nemenyi test is 0.18.

	FC_{OF}DH	FC_{OF}DHDW	FC_{OF}DHIW	SC	SCR
Low. Q. `H/OPT`	106.4%	106.0%	106.8%	105.7%	105.9%
Med. `H/OPT`	110.4%	110.0%	110.6%	109.0%	109.2%
Up. Q. `H/OPT`	117.8%	117.6%	117.9%	114.5%	116.1%
IQR	11.4%	11.6%	11.1%	8.8%	10.2%
Max. `H/OPT`	167.8%	167.8%	168.6%	153.2%	153.2%
Mean Rank	3.10 (4)	2.87 (2)	3.38 (5)	2.56 (1)	3.09 (3)
Nem. Class	B	C	A	D	B
Nice 5 000 t	4.7894^{A}	4.8053^{A}	4.8492^{A}	2.4533^{B}	4.9581^{A}
Path 5 000 t	5.6337^{B}	5.6373^{B}	5.6435^{B}	2.8171^{C}	6.4660^{A}
Bon. Class	B	AB(B)	AB	C	A

Table 6.9: *A summary of the results for the non-guillotine pseudolevel algorithms described in Chapter 4 when applied to the* 1 170 *strip packing benchmark problem instances described in §6.1. The row labelled* 'Median `H/OPT`' *contains the median packing height for all benchmark instances listed in Table 6.1 as a percentage of the optimal packing height, or its lower bound if the optimal is not known. The row labelled* 'Low. Q. `H/OPT`' *contains the value of the lower quartile, the row labelled* 'Up. Q. `H/OPT`' *contains the values of the upper quartile and the interquartile range (in the row labelled* 'IQR'*) is the difference between the two. The row labelled* 'Max. `H/OPT`' *contains the worst result achieved by the algorithms for all benchmark instances. The row labelled* 'Nem. Class' *are results obtained by means of a Nemenyi test. Algorithms in the same group (indicated by letters) do not produce results that are significantly different. The row labelled* 'Bon. Class' *contains results obtained by means of a Bonferroni t test (results obtained by means of a Tukey test are shown in parentheses if they differ) on the average times required to solve all instances of* 5 000 *items. The row labelled* 'Mean Rank' *contains the mean rank achieved by the algorithms in this set (a rank of 1 indicates that the algorithm packed to the lowest height for an instance), with their ranks shown in parentheses. If algorithms yielded the same packing height for an instance, the mean of the ranks that would have been awarded is used. The rows labelled* 'Nice 5 000 t' *and* 'Path 5 000 t' *show the time (in seconds) required for instances of* 5 000 *items (for the "nice" and "pathological" benchmark problem instances [156]), with results from a Bonferroni t test for each set presented as superscripts.*

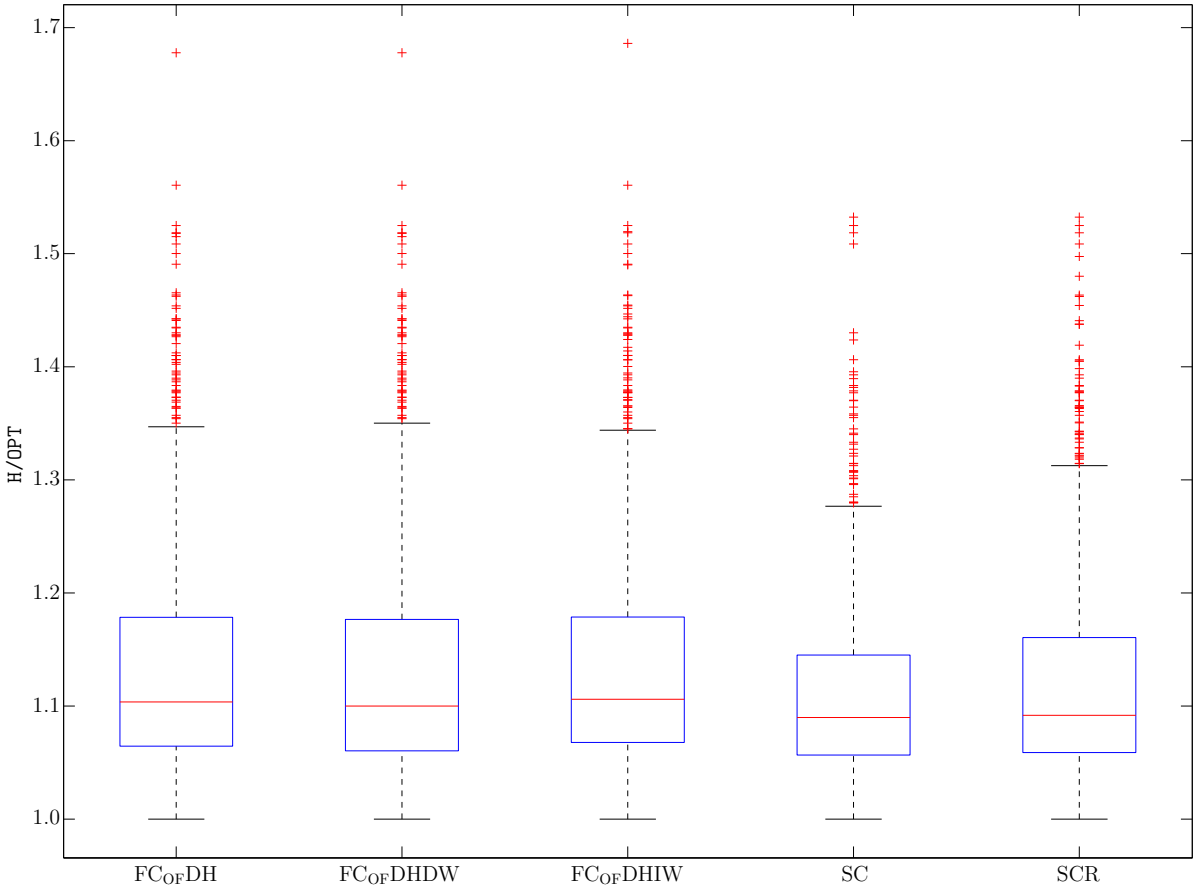


Figure 6.8: *Box plot of the distribution of results for non-guillotine pseudolevel algorithms described in §4.2.1 and §4.3.4 when applied to the 1 170 strip packing problem benchmark instances described in §6.1.*

The comparison of the FC_{OF} algorithms yields familiar results. The lower quartile, median, upper quartile and IQR values are similar, but the mean rank value for the DHDW sorting is lowest, suggesting that it is the best of the three sorting methods for the FC_{OF} algorithm, and the Nemenyi test suggests that the DHDW sorting approach is significantly better than the other two approaches. The SC and SCR algorithms yield better lower quartile, median, upper quartile and IQR values than the FC_{OF} algorithms, suggesting that they are the better algorithms in general. This is reflected in the lower mean rank values for the SC algorithm, but the mean rank for the FC_{OF}DHDW is better than that of the SCR algorithm and the Nemenyi test suggests that the FC_{OF}DHDW algorithm is significantly better than the SCR algorithm. The results suggest that the SC algorithm is the better of the two ceiling-stacking algorithms. The SC algorithm manages to pack three quarters of the instances to within 14.5% if the optimal, while the SCR algorithm packs the same number of benchmark instances to within 16.1% of the optimal packing height.

A Bonferroni t test on the time required by the algorithms to solve problem instances containing 5 000 items suggests that the SCR algorithm is significantly slower than the FC_{OF}DH algorithm (a Tukey test suggests that the FC_{OF}DHDW algorithm is also significantly faster, but the Bonferroni t test was unable to detect this significant difference between the two algorithms). Results from both the Bonferroni t test and the Tukey test suggest that the SCR algorithm is significantly slower for instances containing "pathological" items, but that there was no

significant difference between the algorithms for instances containing "nice" data. However, the SC algorithm is significantly faster than the other algorithms in this comparison set for all instances, combined or separated. The fact that the SC algorithm is both significantly faster, and packs significantly better than the FC_{OF} and SCR algorithms, suggests that it is the best of the non-guillotine pseudolevel algorithms considered in this dissertation for the strip packing problem.

6.4 Results Obtained by Plane-Packing Heuristics

The results obtained by the plane-packing algorithms, when applied to the 1 170 benchmark problem instances of §6.1, are presented in this section. This section is organised in a similar fashion to the previous sections. First, the algorithms are separated into groups of similar procedures and the best algorithm in the group will be selected. The first group comprises the Sleator (see §5.1.1), SF (see §5.1.2), M (see §5.1.5) and UD algorithms (see §5.1.6). The second group comprises the BL algorithm (described in §5.1.3) with its variants, followed by the variants of the SP algorithm (presented in §5.1.4), then the BLF variations (see §5.1.7) and the variations of the GCS algorithm (presented in §5.1.8). Next, each of the three variations of the BF algorithm by Burke *et al.* [22] and their modifications are each compared separately. Finally, the best algorithms from these groups are compared in order to identify the best plane-packing algorithm.

6.4.1 The Free-Packing Sorting-Dependent Algorithms

In this subsection some of the known algorithms (of which one is modified) that do not guarantee guillotine layouts are compared. This includes Sleator's algorithm [148], the modified version of the SP algorithm by Golan *et al.* [62] that does not guarantee a guillotine layout, the M algorithm by Golan [62] and the UD algorithm by Baker *et al.* [5]. There are three versions of Sleator's algorithm and the SPmF algorithm. Items are sorted according to decreasing height for all three versions, but the first does not resolve any equalities, the second resolves equalities by sorting them according to decreasing width, and the third resolves equalities by sorting them according to increasing width. A summary of the results may be found in Table 6.10. A Friedman test performed on the data yields $P = 0$, suggesting that the algorithms pack the benchmark data to significantly different heights. An application of an ANOVA yields $P < 0.0001$, and the critical distance for the Nemenyi test is 0.31.

The results in Table 6.10 suggest that the UD algorithm is the worst in this set of algorithms. The box plot in Figure 6.9 shows that the packing heights are distributed over a larger range for the UD algorithm than for the other algorithms. It may yield lower first quartile and median values than Sleator's algorithm, but its upper quartile is located at a greater relative height and its IQR is more than double that of Sleator's algorithm. The mean rankings in Table 6.10 and the rankings that are obtained when applying the Nemenyi test to the results suggest that Sleator's algorithms are the worst, on average, after the UD algorithm. There appears to be no significant differences within the Sleator set. This observation also holds for the sets of SPmF and M algorithms. The SPmF algorithms yield similar upper quartile and maximum packing height values to the variations of Sleator's algorithm, but the differences between the lower quartile and median values for the SPmF and Sleator's algorithms are larger, suggesting that the majority of the SPmF packing heights are closer to the optimum than for Sleator's algorithm. This observation is corroborated in the results for the mean rank values, where

	S(DH)	S(DHDW)	S(DHIW)	SPmF(DH)	SPmF(DHDW)	SPmF(DHIW)	M	UD
Low. Q. H/OPT	118.8%	119.0%	118.7%	111.1%	110.8%	111.3%	110.1%	110.6%
Med. H/OPT	127.0%	127.0%	127.3%	122.7%	122.4%	123.0%	118.8%	125.8%
Up. Q. H/OPT	134.1%	134.2%	134.0%	133.7%	133.3%	134.2%	136.9%	151.8%
IQR	15.3%	15.2%	15.3%	22.6%	22.5%	22.9%	26.8%	41.2%
Max. H/OPT	197.5%	197.5%	197.5%	197.5%	197.5%	197.5%	234.9%	274.8%
Mean Rank	4.77 (5)	4.81 (7)	4.78 (6)	4.07 (3)	3.97 (1)	4.18 (4)	4.03 (2)	5.40 (8)
Sig. Class	B(B)	B(B)	B(B)	C(B)	C(B)	C(B)	C(B)	A(A)
Nice 5 000 t	2.3014^{B}	2.2902^{B}	2.2902^{B}	198.21^{A}	198.21^{A}	197.03^{A}	4.4478^{B}	2.1288^{B}
Path 5 000 t	2.2873^{B}	2.3031^{B}	2.2940^{B}	878.71^{A}	878.70^{A}	869.67^{A}	4.3817^{B}	2.1242^{B}
Bon. Class	B	B	B	A	A	A	B	B

Table 6.10: *A summary of the results for the SPmF, M, UD and Sleator's algorithms and their variations described in §5.1 when applied to the* 1 170 *strip packing benchmark problem instances described in §6.1. The row labelled* 'Median H/OPT' *contains the median packing height for all benchmark instances listed in Table 6.1 as a percentage of the optimal packing height, or its lower bound if the optimal is not known. The row labelled* 'Low. Q. H/OPT' *contains the value of the lower quartile, the row labelled* 'Up. Q. H/OPT' *contains the values of the upper quartile and the interquartile range (in the row labelled* 'IQR') *is the difference between the two. The row labelled* 'Max. H/OPT' *contains the worst result achieved by the algorithms for all benchmark instances. The row labelled* 'Sig. Class' *are results obtained by means of a Nemenyi test, with the results from a Bonferroni t test in parentheses. Algorithms in the same group (indicated by letters) do not produce results that are significantly different. The row labelled* 'Bon. Class' *contains results obtained by means of a Bonferroni t test on the average times required to solve all instances of* 5 000 *items. The row labelled* 'Mean Rank' *contains the mean rank achieved by the algorithms in this set (a rank of 1 indicates that the algorithm packed to the lowest height for an instance), with their ranks shown in parentheses. If algorithms yielded the same packing height for an instance, the mean of the ranks that would have been awarded is used. The rows labelled* 'Nice 5 000 t' *and* 'Path 5 000 t' *show the time (in seconds) required for instances of* 5 000 *items (for the "nice" and "pathological" benchmark problem instances [156]), with results from a Bonferroni t test for each set presented as superscripts.*

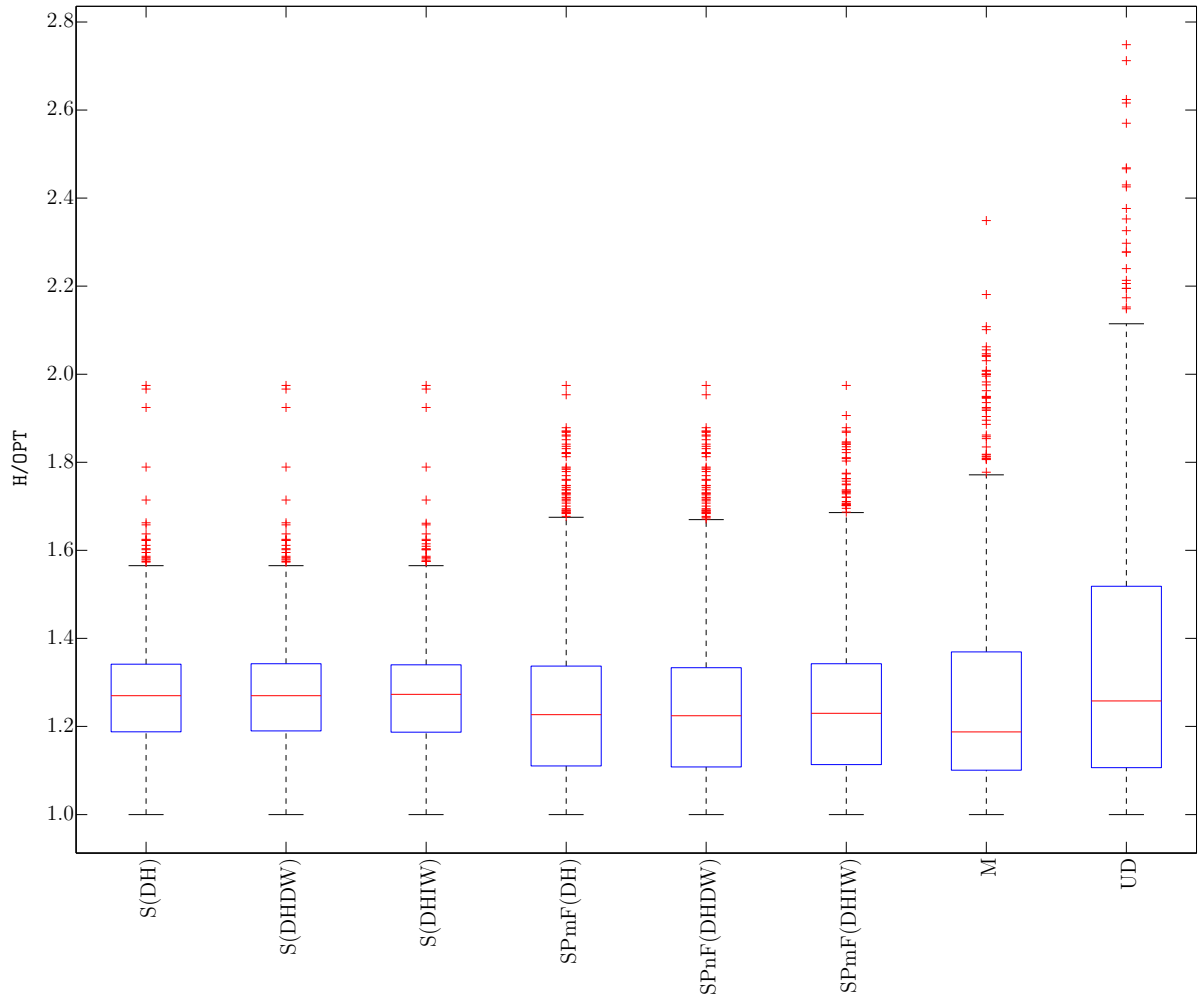


Figure 6.9: *Box plot of the distribution of results for non-guillotine plane-packing algorithms described in §5.1 when applied to the 1 170 strip packing problem benchmark instances described in §6.1.*

the SPmF algorithms yield better mean rankings. The M algorithm proves to yield the lowest median value and the second best mean rank (not significantly different to the algorithm with the worst mean rank).

The SPmF algorithms may be ranked above Sleator's algorithm and the UD algorithm, but it is two orders of magnitude slower, which makes it an inappropriate heuristic to include in further comparisons. A Bonferroni t test on the times required to solve the benchmark instances containing 5 000 items suggest no significant difference between Sleator's algorithms and the UD and M algorithms, suggesting that the M algorithm is the best of the algorithms in this comparison because it performs significantly better than Sleator's algorithms and the UD algorithm in terms of packing height.

6.4.2 The Guillotine-Packing Sorting-Dependent Algorithms

In this subsection some of the known algorithms that yield guillotine layouts (of which one is modified) are compared. This includes the SF algorithm by Coffman *et al.* [32], the SP algorithm by Golan *et al.* [62] and the modified version of the SP algorithm that attempts to find a packing that is more dense. There are three versions of each of the algorithms. Items

are sorted according to decreasing height for the three versions of the SF algorithm, but the first does not resolve any equalities, the second resolves equalities by sorting them according to decreasing width, and the third resolves equalities by sorting them according to increasing width. Similarly, the SP algorithm and the modified version are each given three packing types that all sort the items according to decreasing width; the first does not resolve ties, the second resolves ties by sorting those items according to decreasing height, and the third resolves the ties by sorting them according to increasing height. A summary of the results may be found in Table 6.11 and box plots of the results may be found in Figure 6.10. A Friedman test on the results yield $P = 0$, suggesting that there are significant differences in the results produced by the various algorithms. An application of an ANOVA yields a similar result, with $P < 0.0001$. The critical distance between ranks for the Nemenyi test is 0.35.

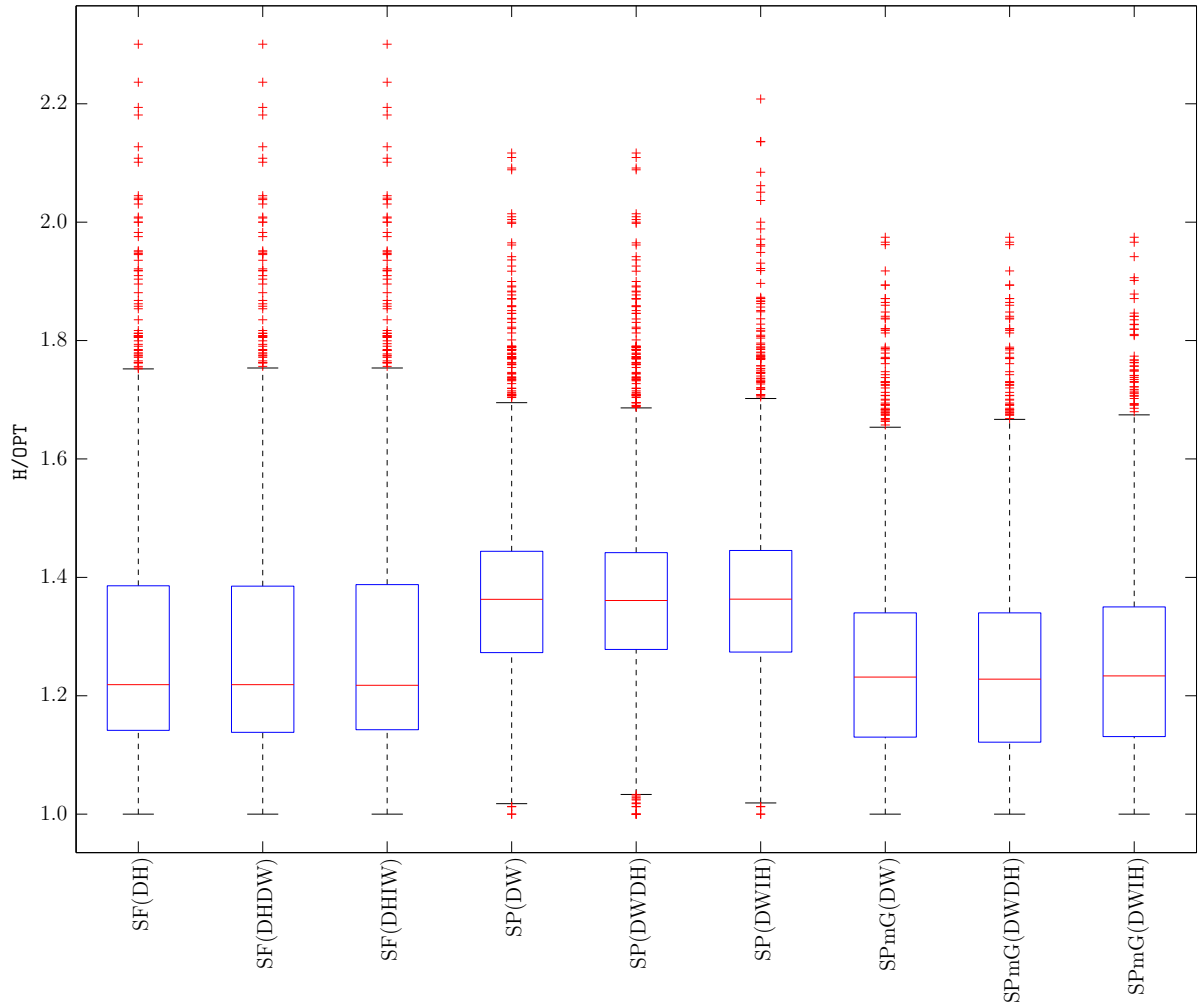


Figure 6.10: *Box plot of the distribution of results for guillotine plane-packing algorithms described in §5.1 when applied to the 1 170 strip packing problem benchmark instances described in §6.1.*

The box plots in Figure 6.10 suggest that the SP algorithms are the worst group of algorithms in this comparison set. Their lower quartile, median and upper quartile values are higher than those of the other six algorithms. The argument is further strengthened by the mean rank values, which are larger for the SP algorithms than for the SF or SPmG algorithms. Both the Nemenyi test and the Bonferroni t test suggest that the SP algorithms are not significantly different from one another, but both tests rank them the worst algorithms in this set. The

	SF(DH)	SF(DHDW)	SF(DHIW)	SP(DW)	SP(DWDH)	SP(DWIH)	SPmG(DW)	SPmG(DWDH)	SPmG(DWIH)
Low. Q. `H/OPT`	114.1%	113.8%	114.2%	127.3%	127.8%	127.4%	113.0%	112.2%	113.1%
Med. `H/OPT`	121.9%	121.9%	121.8%	136.3%	136.1%	136.3%	123.2%	122.8%	123.3%
Up. Q. `H/OPT`	138.6%	138.5%	138.7%	144.4%	144.2%	144.5%	134.0%	134.0%	135.0%
IQR	24.4%	24.7%	24.5%	17.1%	16.3%	17.2%	21.0%	21.8%	21.9%
Max. `H/OPT`	230.1%	230.1%	230.1%	211.7%	211.7%	220.8%	197.5%	197.5%	197.5%
Mean Rank	4.37 (5)	4.28 (4)	4.51 (6)	7.29 (9)	7.25 (8)	7.20 (7)	3.41 (2)	3.20 (1)	3.49 (3)
Sig. Class	B(B)	B(B)	B(B)	A(A)	A(A)	A(A)	C(C)	C(C)	C(C)
Nice 5 000 t	2.5760[B]	2.5891[B]	2.5888[B]	2.3169[C]	2.3332[C]	2.3331[C]	3.1723[A]	3.1580[A]	3.1560[A]
Path 5 000 t	2.5468[B]	2.5471[B]	2.5477[B]	2.3402[C]	2.3406[C]	2.3410[C]	3.4890[A]	3.4899[A]	3.4815[A]
Bon. Class	B	B	B	C	C	C	A	A	A

Table 6.11: *A summary of the results for the SF, SP and SPmG algorithms and their variations described in §5.1 when applied to the* 1 170 *strip packing benchmark problem instances described in §6.1. The row labelled* 'Median `H/OPT`' *contains the median packing height for all benchmark instances listed in Table 6.1 as a percentage of the optimal packing height, or its lower bound if the optimal is not known. The row labelled* 'Low. Q. `H/OPT`' *contains the value of the lower quartile, the row labelled* 'Up. Q. `H/OPT`' *contains the values of the upper quartile and the interquartile range (in the row labelled* 'IQR') *is the difference between the two. The row labelled* 'Max. `H/OPT`' *contains the worst result achieved by the algorithms for all benchmark instances. The row labelled* 'Sig. Class' *are results obtained by means of a Nemenyi test, with the results from a Bonferroni t test in parentheses. Algorithms in the same group (indicated by letters) do not produce results that are significantly different. The row labelled* 'Bon. Class' *contains results obtained by means of a Bonferroni t test on the average times required to solve all instances of* 5 000 *items. The row labelled* 'Mean Rank' *contains the mean rank achieved by the algorithms in this set (a rank of 1 indicates that the algorithm packed to the lowest height for an instance), with their ranks shown in parentheses. If algorithms yielded the same packing height for an instance, the mean of the ranks that would have been awarded is used. The rows labelled* 'Nice 5 000 t' *and* 'Path 5 000 t' *show the time (in seconds) required for instances of* 5 000 *items (for the "nice" and "pathological" benchmark problem instances [156]), with results from a Bonferroni t test for each set presented as superscripts.*

Bonferroni t test and the Nemenyi test find the same results. The SPmG algorithms are the highest-ranked algorithms in this set, with the SPmG(DHDH) algorithm yielding the best mean rank.

However, these good rankings for the SPmG algorithms come at the cost of increased computation time. A Bonferroni t test on the times required to solve large problem instances (containing 5 000 items) suggest significant differences between the groups, with the algorithm group's time ranks inversely proportional to their packing height ranks. This makes for a difficult choice with respect to which algorithm in this set is the best. If one requires a guillotine packing, then these algorithms are all bettered by the SL_5 algorithm, the application of which yielded a median packing height for the 1 170 benchmark instances which is lower than the lower quartile of any algorithm in this set, and an upper quartile value that is lower than any of the median values in this set. The better packing density is not achieved at the cost of increased computation time, with the SL_5 algorithm requiring similar times to the SP algorithms, the fastest algorithms in this set. This observation is reflected in the mean rank values over all algorithms, with the best algorithm in this set achieving a mean rank of 184.05 compared to the SL_5 algorithm's mean rank of 72.62 when taken over all strip packing algorithms in this chapter. Therefore, the algorithms in this set will not be used in any further comparisons.

6.4.3 The BL Algorithm

This subsection is dedicated to the results obtained when applying the 23 variations of the BL algorithm by Baker *et al.* [6], discussed in §5.1.3, to the 1 170 benchmark instances. There are three versions of items sorted according to decreasing height, width and area each, and 7 variations of the xWDWDH and xRDWDH sorting methods. The distribution of the results for each of the sorting methods may be found in the form of box plots in Figure 6.11 and further results may be found in Table 6.12. A Friedman test yields $P = 0.000$ and an ANOVA yields $P < 0.0001$, suggesting that the algorithms are significantly different. The Nemenyi test yields a critical distance of 1.01.

The box plots in Figure 6.11 suggest that the DW, DA and xRDWDH sorting methods typically yield greater packing heights than the DH and xWDWDH sorting methods. The plots suggest that the xRDWDH sorting methods yield better solutions as the number of items sorted according to decreasing width decreases. The algorithms that sort items according to decreasing height appear to yield very good solutions, but the algorithms that sort items according to their relative width yield the best results, on average, with the algorithm that only sorts by width those items that are wider than half the strip width yielding the best results. This may be seen in the mean rank values, which prove to be lower for the worst xWDWDH algorithm than the best of the other sorting methods. In fact, the upper quartile value of the results for the $\frac{1}{2}DHDW$ variation is similar to the lower quartile value of the DWIH variation. The mean ranking suggests that this is the best of the algorithms in the comparison set and performing a Nemenyi test on the results confirms that the $\frac{1}{2}DHDW$ variation is significantly better than 18 of the other algorithms in this set. The parametric Bonferroni t test does not yield as fine a ranking as the nonparametric Nemenyi test, suggesting that all xWDWDH algorithms are not significantly different from one another.

However, the better performance of the xWDWDH algorithms comes at the cost of increased computation time for benchmark instances containing "nice items." Applying an ANOVA at a confidence level of 95% on the results for algorithmic time yields $P < 0.0001$ for only the instances containing "nice" items, only the instances containing "pathological" items and both sets combined. For large benchmark instances consisting of 5 000 items, the xWDWDH

	DH	DHDW	DHIW	DW	DWDH	DWIH	DA	DADH	DADW	$\frac{2}{3}$W	$\frac{3}{5}$W	$\frac{11}{20}$W
Low. Q. H/OPT	111.5%	111.1%	111.9%	118.8%	118.6%	119.2%	118.2%	118.2%	118.2%	110.0%	109.8%	109.6%
Med. H/OPT	116.6%	116.5%	116.9%	124.6%	124.6%	124.4%	124.5%	124.4%	124.5%	114.6%	114.3%	114.3%
Up. Q. H/OPT	123.6%	123.4%	123.6%	136.5%	136.3%	136.4%	135.0%	134.6%	135.0%	120.0%	119.5%	119.5%
IQR	12.1%	12.2%	11.8%	17.8%	17.6%	17.3%	16.8%	16.5%	16.8%	10.0%	9.7%	9.9%
Max. H/OPT	168.3%	168.3%	168.3%	204.8%	204.8%	200.4%	200.0%	200.0%	200.0%	162.1%	162.1%	162.1%
Mean Rank	11.62 (13)	11.26 (12)	12.15 (15)	18.24 (23)	18.16 (22)	17.94 (21)	17.19 (19)	17.15 (18)	17.20 (20)	8.06 (4)	7.73 (3)	7.63 (2)
Sig. Class	EF(I)	EFG(I)	DE(I)	A(A)	AB(AB)	AB(ABC)	B(BC)	B(C)	B(BC)	P(J)	LM(J)	LM(J)
Nice 5 000 t	8.7234^{H}	8.7342^{H}	8.7277^{H}	8.9346^{G}	8.9476^{G}	8.9451^{G}	8.9173^{G}	8.9297^{G}	8.9323^{G}	11.176^{A}	11.170^{A}	11.173^{A}
Path 5 000 t	9.7106^{B}	9.7015^{B}	9.6640^{B}	11.608^{A}	11.598^{A}	11.580^{A}	11.269^{A}	11.273^{A}	11.271^{A}	12.127^{A}	12.131^{A}	12.127^{A}
Bon. Class	C	C	C	BC	BC	BC	BC	BC	BC	A	A	A

	$\frac{1}{2}$W	$\frac{9}{20}$W	$\frac{2}{5}$W	$\frac{1}{3}$W	$\frac{2}{3}$R	$\frac{3}{5}$R	$\frac{11}{20}$R	$\frac{1}{2}$R	$\frac{9}{20}$R	$\frac{2}{5}$R	$\frac{1}{3}$R
Low. Q. H/OPT	109.6%	110.0%	110.0%	109.6%	114.9%	114.2%	113.3%	112.6%	112.1%	111.9%	111.5%
Med. H/OPT	114.2%	115.0%	115.2%	115.1%	120.2%	119.9%	118.8%	118.1%	117.3%	116.5%	116.0%
Up. Q. H/OPT	119.3%	121.1%	120.8%	120.3%	129.7%	128.0%	127.0%	125.2%	123.9%	122.8%	121.8%
IQR	9.8%	11.1%	10.8%	10.6%	14.8%	13.8%	13.6%	12.5%	11.8%	10.9%	10.3%
Max. H/OPT	151.7%	151.7%	151.7%	156.0%	201.2%	200.0%	197.0%	197.0%	197.0%	184.2%	184.2%
Mean Rank	7.45 (1)	8.92 (7)	8.60 (6)	8.14 (5)	13.43 (17)	12.84 (16)	11.97 (14)	11.10 (11)	10.46 (10)	9.65 (9)	9.12 (8)
Sig. Class	M(J)	IJK(J)	JKL(J)	JKLM(J)	C(D)	CD(DE)	DEF(EF)	FG(FG)	GH(GH)	HI(HI)	IJ(I)
Nice 5 000 t	11.163^{A}	11.172^{A}	11.164^{A}	11.168^{A}	9.1251^{F}	9.4065^{E}	9.4711^{D}	9.4826^{D}	9.6665^{C}	9.7140^{C}	9.8571^{B}
Path 5 000 t	12.150^{A}	12.146^{A}	12.147^{A}	12.149^{A}	12.087^{A}	12.067^{A}	12.018^{A}	11.885^{A}	11.837^{A}	11.792^{A}	11.782^{A}
Bon. Class	A	A	A	A	AB	AB	AB	AB	AB	AB	AB

Table 6.12: *A summary of the results for the BL algorithm and its sorting variations described in §5.1.3 when applied to the* 1 170 *strip packing benchmark problem instances described in §6.1. The row labelled* 'Median H/OPT' *contains the median packing height for all benchmark instances listed in Table 6.1 as a percentage of the optimal packing height, or its lower bound if the optimal is not known. The row labelled* 'Low. Q. H/OPT' *contains the value of the lower quartile, the row labelled* 'Up. Q. H/OPT' *contains the values of the upper quartile and the interquartile range (in the row labelled* 'IQR') *is the difference between the two. The row labelled* 'Max. H/OPT' *contains the worst result achieved by the algorithms for all benchmark instances. The row labelled* 'Sig. Class' *are results obtained by means of a Nemenyi test, with the results from a Bonferroni t test in parentheses. Algorithms in the same group (indicated by letters) do not produce results that are significantly different. The row labelled* 'Bon. Class' *contains results obtained by means of a Bonferroni t test on the average times required to solve all instances of* 5 000 *items. The row labelled* 'Mean Rank' *contains the mean rank achieved by the algorithms in this set (a rank of 1 indicates that the algorithm packed to the lowest height for an instance), with their ranks shown in parentheses. If algorithms yielded the same packing height for an instance, the mean of the ranks that would have been awarded is used. The rows labelled* 'Nice 5 000 t' *and* 'Path 5 000 t' *show the time (in seconds) required for instances of* 5 000 *items (for the "nice" and "pathological" benchmark problem instances [156]), with results from a Bonferroni t test for each set presented as superscripts.*

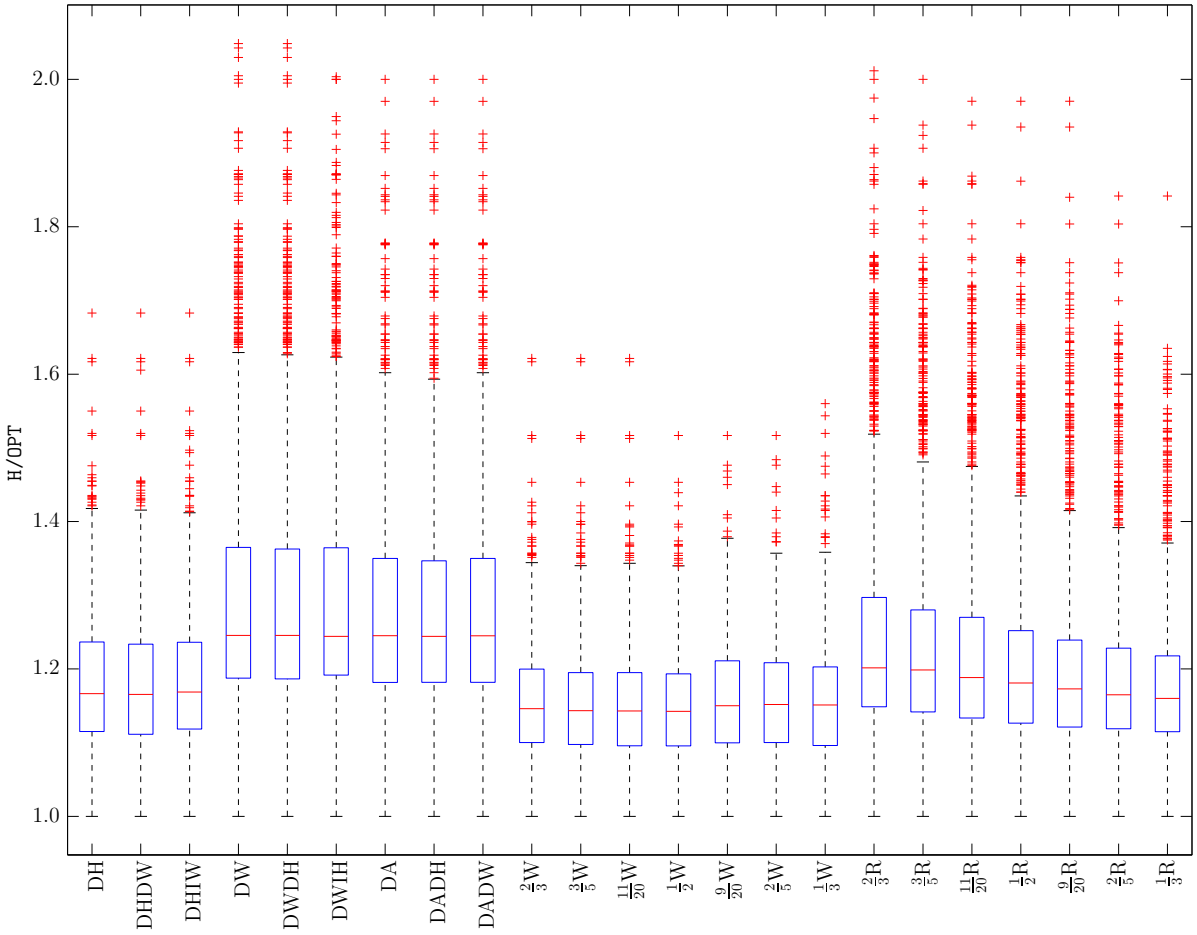


Figure 6.11: *Box plot of the distribution of results for the BL algorithms described in §5.1.3 when applied to the 1 170 strip packing problem benchmark instances described in §6.1.*

algorithms require, on average, more than 2 seconds additional time to find a solution than do the DH packing variations (which are significantly faster than the other algorithms for "nice" and "pathological" items). The DH, DHDW and DHIW algorithms are not significantly different in terms of time; hence the DHDW variation, which yielded a better mean rank than the DH and DHIW variations, would be the better algorithm to use if computation time played a significant role in the algorithm selection. All algorithms that are not significantly different in terms of time were significantly worse in terms of packing height.

6.4.4 The BLF Algorithm

This subsection is dedicated to the results obtained by applying the 23 variations of the BLF algorithm by Chazelle [25], discussed in §5.1.7, to the 1 170 benchmark instances in §6.1. There are three versions in which items sorted according to decreasing height, width and area each, and 7 variations of the xWDWDH and xRDWDH sorting methods. The distribution of the results for each of the sorting methods may be found in the form of box plots in Figure 6.12 and further results may be found in Table 6.13. A Friedman test yields $P = 0.000$ and an ANOVA yields $P < 0.0001$, suggesting that the algorithms are significantly different. The Nemenyi test requires a critical difference of 1.01 between the mean rankings of algorithms for them to be significantly different.

	DH	DHDW	DHIW	DW	DWDH	DWIH	DA	DADH	DADW	$\frac{2}{3}$W	$\frac{3}{5}$W	$\frac{11}{20}$W
Low. Q. H/OPT	106.0%	105.7%	106.4%	107.9%	107.1%	108.1%	106.4%	106.3%	106.4%	104.7%	104.5%	104.1%
Med. H/OPT	109.5%	109.1%	109.9%	116.6%	116.4%	117.1%	112.9%	113.0%	112.9%	108.1%	107.9%	107.8%
Up. Q. H/OPT	114.4%	114.1%	114.9%	128.7%	128.5%	129.5%	119.5%	119.6%	119.5%	113.3%	113.0%	113.1%
IQR	8.4%	8.3%	8.5%	20.8%	21.4%	21.4%	13.1%	13.3%	13.1%	8.7%	8.5%	9.0%
Max. H/OPT	151.7%	151.7%	151.7%	193.5%	193.5%	192.4%	193.2%	193.2%	193.2%	151.7%	151.7%	151.7%
Mean Rank	12.32 (15)	11.50 (12)	13.33 (17)	17.83 (22)	17.10 (21)	18.14 (23)	14.23 (19)	14.20 (18)	14.23 (20)	9.92 (7)	9.38 (6)	8.81 (5)
Sig. Class	DEF(HI)	FG(HI)	CD(GH)	AB(A)	B(A)	A(A)	C(CDE)	C(CDE)	C(CDE)	H(HI)	HI(HI)	IJ(I)
Nice 5 000 t	25.364^{F}	25.374^{F}	25.307^{F}	34.301^{ABC}	34.301^{ABC}	34.382^{ABC}	33.414^{D}	33.407^{D}	33.408^{D}	27.834^{E}	27.846^{E}	27.829^{E}
Path 5 000 t	31.172^{D}	31.174^{D}	30.934^{D}	178.35^{AB}_{A}	178.39^{A}	193.86^{A}	91.056^{CD}_{BCD}	91.075^{CD}_{BCD}	91.083^{CD}_{BCD}	33.611^{D}_{CD}	33.632^{D}_{CD}	33.626^{D}_{CD}
Bon. Class	C	C	C	AB	AB	A	BC	BC	BC	C	C	C

	$\frac{1}{2}$W	$\frac{9}{20}$W	$\frac{2}{5}$W	$\frac{1}{3}$W	$\frac{2}{3}$R	$\frac{3}{5}$R	$\frac{11}{20}$R	$\frac{1}{2}$R	$\frac{9}{20}$R	$\frac{2}{5}$R	$\frac{1}{3}$R
Low. Q. H/OPT	104.0%	103.9%	104.0%	104.2%	105.8%	105.5%	105.2%	105.4%	105.2%	105.5%	105.5%
Med. H/OPT	107.5%	107.6%	107.5%	107.7%	113.3%	112.1%	111.8%	111.1%	110.6%	110.3%	109.9%
Up. Q. H/OPT	113.0%	112.8%	113.0%	113.4%	122.0%	120.8%	119.6%	118.9%	117.8%	116.9%	116.2%
IQR	9.0%	8.9%	9.0%	9.2%	16.2%	15.3%	14.4%	13.4%	12.6%	11.5%	10.7%
Max. H/OPT	151.7%	151.7%	151.7%	156.0%	201.2%	200.0%	197.0%	197.0%	197.0%	184.2%	184.2%
Mean Rank	8.26 (3)	8.18 (2)	8.09 (1)	8.34 (4)	13.27 (16)	12.28 (14)	11.85 (13)	11.40 (11)	11.19 (10)	11.08 (8)	11.08 (9)
Sig. Class	J(I)	J(I)	J(I)	J(HI)	CDE(B)	EF(BC)	FG(BCD)	FG(CDE)	G(DEF)	G(EF)	G(FG)
Nice 5 000 t	27.851^{E}	27.832^{E}	27.821^{E}	27.825^{E}	34.796^{A}	34.488^{AB}	34.484^{AB}	34.417^{ABC}	34.124^{BC}	33.796^{DC}	33.343^{D}
Path 5 000 t	33.798^{D}_{CD}	33.801^{D}_{CD}	33.798^{D}_{CD}	33.794^{D}_{CD}	112.13^{BC}_{B}	104.26^{C}_{B}	95.655^{CD}_{BC}	91.264^{CD}_{BCD}	84.979^{CD}_{BCD}	77.933^{CD}_{BCD}	74.110^{CD}_{BCD}
Bon. Class	C	C	C	C	ABC	ABC	ABC	ABC(BC)	BC	BC(C)	C

Table 6.13: *A summary of the results for the BLF algorithm and its sorting variations described in §5.1.7 when applied to the* 1 170 *strip packing benchmark problem instances described in §6.1. The row labelled* 'Median H/OPT' *contains the median packing height for all benchmark instances listed in Table 6.1 as a percentage of the optimal packing height, or its lower bound if the optimal is not known. The row labelled* 'Low. Q. H/OPT' *contains the value of the lower quartile, the row labelled* 'Up. Q. H/OPT' *contains the values of the upper quartile and the interquartile range (in the row labelled* 'IQR'*) is the difference between the two. The row labelled* 'Max. H/OPT' *contains the worst result achieved by the algorithms for all benchmark instances. The row labelled* 'Sig. Class' *are results obtained by means of a Nemenyi test, with the results from a Bonferroni t test in parentheses. Algorithms in the same group (indicated by letters) do not produce results that are significantly different. The row labelled* 'Bon. Class' *contains results obtained by means of a Bonferroni t test (results obtained by means of a Tukey test are shown in parentheses if the two tests yield different results) on the average times required to solve all instances of* 5 000 *items. The row labelled* 'Mean Rank' *contains the mean rank achieved by the algorithms in this set (a rank of 1 indicates that the algorithm packed to the lowest height for an instance), with their ranks shown in parentheses. If algorithms yielded the same packing height for an instance, the mean of the ranks that would have been awarded is used. The rows labelled* 'Nice 5 000 t' *and* 'Path 5 000 t' *show the time (in seconds) required for instances of* 5 000 *items (for the "nice" and "pathological" benchmark problem instances [156]), with results from a Bonferroni t test for each set presented as superscripts (results from the Tukey test appear as subscripts if they are different).*

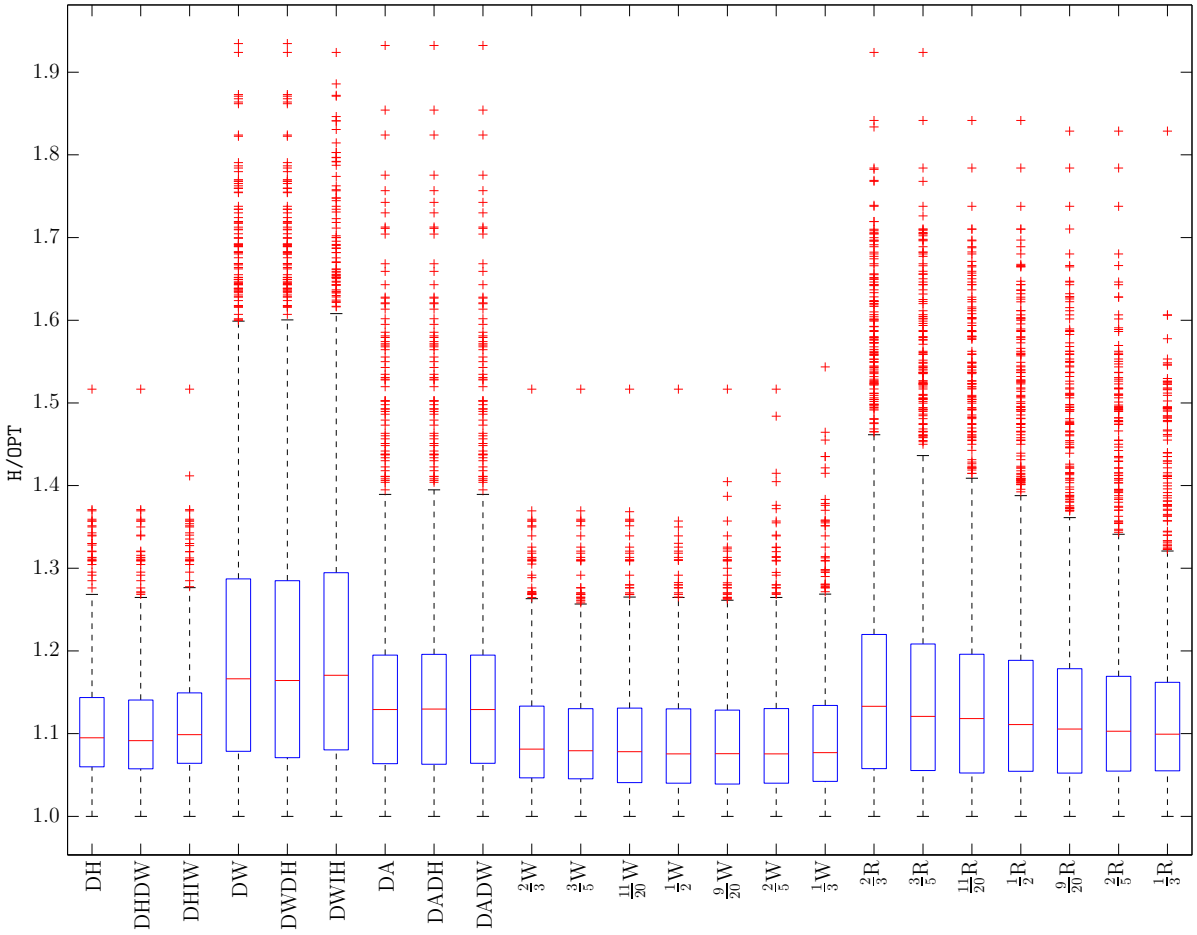


Figure 6.12: *Box plot of the distribution of results for the BLF algorithms described in §5.1.7 when applied to the 1 170 strip packing problem benchmark instances described in §6.1.*

The distributions of the results for the algorithms, as shown in the box plots in Figure 6.12, again suggest a familiar pattern in packing height results for the various sorting methods. The algorithms that sort all of the items according to decreasing width yield the worst results, and the results of the sorting methods that sort a fixed number of items according to decreasing width (and the remainder according to decreasing height) improve as the number of items that are sorted according to decreasing width diminishes. The algorithms that sort items according to decreasing area yielded similar results to the DW algorithms for the BL algorithm. However, for the BLF algorithm the majority the DA algorithms' results are distributed closer to the optimum than for the DW algorithm. Once again the DH algorithms yield a good distribution of results, but the algorithms that sort items according to their relative width yield the best results, as was found for the BL algorithm. However, a Nemenyi test performed on the results suggests that the $\frac{1}{2}$WDWDH, the $\frac{9}{20}$WDWDH and the $\frac{2}{5}$WDWDH algorithms do not yield significant differences in packing height, but these three algorithms are significantly better than the remainder of the algorithms.

The $\frac{1}{2}$WDWDH algorithmic variation may have sacrificed speed in order to pack more densely than the DHDW variation for the BL algorithm, but for the BLF algorithm a Bonferroni t test suggests that the time required for the two algorithms to solve large problems is not significantly different, even though the mean times for the $\frac{1}{2}$WDWDH variation are more than two seconds slower than for the DHDW variation. Sorting the items according to decreasing width yields the

slowest solution times which, when combined with the poor results in terms of packing height distribution relative to the other sorting methods, suggest it is the worst sorting method for the BLF algorithm. There are a few differences in the results generated when a Bonferroni t test and a Tukey test is applied to the solution times. When the instances containing "nice" and "pathological" items are combined, the Tukey test suggests a significant difference between the $\frac{2}{5}$RDWDH variation and the DW and DWDH variations, and between the $\frac{1}{2}$RDWDH and DHIH variations, while the Bonferroni t test did not find significant differences. When considering only the "pathological" items, the Tukey test suggests a significant difference between the DW and $\frac{2}{3}$RDWDH variations, and between the $\frac{11}{20}$RDWDH and DH, DHDW and DHIW variations, which the Bonferroni t test did not.

6.4.5 The GCS Algorithm

This subsection is dedicated to the results obtained by applying the 23 variations of the GCS algorithm by MacLeod *et al.* [109], discussed in §5.1.8. Only 1 150 benchmark instances were used for comparison purposes for the GCS algorithm as it did not find solutions within an hour for the benchmark instances which contain 5 000 items. There are three versions in which items are sorted according to decreasing height, width and area each, and 7 variations of the xWDWDH and xRDWDH sorting methods. The distribution of the results for each of the sorting methods may be found in the form of box plots in Figure 6.13 and further results may be found in Table 6.14. A Friedman test yields $P = 0.000$ and an ANOVA yields $P < 0.0001$, suggesting that the algorithms are significantly different. The critical distance for Nemenyi's test is 1.02 for 23 algorithms and 1 150 benchmark instances.

The variations on the GCS algorithm that sort all items according to decreasing width are, as for the BL and BLF algorithms, the worst performing of the variations in this comparison set in terms of packing height. For the previous two algorithms considered in §6.4.3 and §6.4.4 the variations that sort according to decreasing area were the second worst subset in terms of packing height. However, for the GCS algorithm the decreasing area sorting method is not significantly worse than the DH and DHIW sorting variations. As with the other algorithms, the variations that sort items according to DHDW are bettered only by those algorithms that sort items according to their relative width. The $\frac{1}{2}$WDWDH variation is the highest ranked algorithm (in terms of packing height) in this comparison set, and is significantly better than 17 of the other algorithms.

Applying an ANOVA to the solution times the algorithm requires to pack all benchmark instances of 2 000 items results in P-value of 0.3118, suggesting that the difference in computation times is not significant. However, performing an ANOVA on the "nice" items yields $P = 0.0038$, and when applied to the "pathological" items it yields $P < 0.0001$, suggesting that the hypothesis that all algorithms are equivalent in terms of solution time may be rejected. The Bonferroni t test and the Tukey test are both not powerful enough to distinguish between the algorithms when applied to "nice" benchmark instances, but they did find significant differences between the algorithms when applied to the benchmark instances containing "pathological" items. The Tukey test detected a significant difference between the DHIW variation and the $\frac{3}{5}$RDWDH and $\frac{2}{3}$RDWDH variations which the Bonferroni t test did not.

6.4.6 The BFLM Algorithm

The results obtained by applying the BFLM algorithm by Burke *et al.* [22], and its modifications presented in §5.1.9, to the 1 170 benchmark instances are reported in this section. There are

	DH	DHDW	DHIW	DW	DWDH	DWIH	DA	DADH	DADW	$\frac{2}{3}$W	$\frac{3}{5}$W	$\frac{11}{20}$W
Low. Q. H/OPT	106.9%	106.6%	107.6%	108.7%	107.4%	109.3%	106.7%	106.7%	106.9%	105.8%	105.2%	105.0%
Med. H/OPT	110.6%	110.3%	111.5%	119.6%	119.0%	119.8%	113.9%	114.0%	113.9%	109.9%	109.6%	109.6%
Up. Q. H/OPT	117.4%	117.0%	117.8%	131.2%	131.0%	131.6%	120.9%	121.0%	121.0%	116.9%	116.6%	116.6%
IQR	10.5%	10.4%	10.3%	22.5%	23.6%	22.3%	14.2%	14.3%	14.1%	11.1%	11.3%	11.6%
Max. H/OPT	151.7%	151.7%	151.7%	197.5%	197.5%	197.5%	194.5%	194.5%	194.5%	151.7%	151.7%	151.7%
Mean Rank	11.72 (9)	10.92 (8)	13.06 (19)	16.98 (22)	16.15 (21)	17.54 (23)	12.46 (11)	12.37 (10)	12.50 (13)	9.67 (7)	9.06 (5)	8.75 (3)
Sig. Class	EF(G)	F(G)	CD(G)	AB(A)	B(A)	A(A)	DE(DEF)	DE(DEF)	DE(DEF)	G(G)	GH(G)	GH(G)
Nice 2 000 t	$2\,021.7^{A}$	$2\,021.1^{A}$	$2\,017.2^{A}$	$2\,088.0^{A}$	$2\,088.1^{A}$	$2\,078.1^{A}$	$2\,009.5^{A}$	$2\,009.6^{A}$	$2\,009.6^{A}$	$2\,021.4^{A}$	$2\,021.4^{A}$	$2\,021.4^{A}$
Path 2 000 t	678.13^{E}_{EF}	678.09^{E}_{EF}	664.82^{E}_{F}	789.05^{DE}_{EF}	788.96^{DE}_{EF}	787.60^{E}_{EF}	883.15^{CDE}_{C-F}	882.79^{CDE}_{C-F}	883.05^{CDE}_{DEF}	678.36^{E}_{EF}	678.33^{E}_{EF}	678.65^{E}_{EF}

	$\frac{1}{2}$W	$\frac{9}{20}$W	$\frac{2}{5}$W	$\frac{1}{3}$W	$\frac{2}{3}$R	$\frac{3}{5}$R	$\frac{11}{20}$R	$\frac{1}{2}$R	$\frac{9}{20}$R	$\frac{2}{5}$R	$\frac{1}{3}$R
Low. Q. H/OPT	104.9%	104.9%	105.0%	105.5%	107.0%	106.8%	106.8%	106.9%	106.7%	107.0%	107.0%
Med. H/OPT	109.7%	109.9%	110.0%	110.1%	116.4%	115.9%	115.1%	114.5%	114.2%	113.3%	113.0%
Up. Q. H/OPT	116.5%	116.6%	116.8%	117.5%	124.9%	124.0%	122.7%	122.4%	121.5%	120.8%	120.1%
IQR	11.6%	11.7%	11.8%	12.1%	17.9%	17.1%	15.9%	15.5%	14.8%	13.9%	13.1%
Max. H/OPT	151.7%	155.8%	151.7%	148.4%	192.4%	192.4%	178.4%	178.4%	178.4%	178.4%	170.8%
Mean Rank	8.48 (1)	8.56 (2)	8.78 (4)	9.45 (6)	13.59 (20)	13.06 (18)	12.53 (14)	12.62 (16)	12.47 (12)	12.58 (15)	12.69 (17)
Sig. Class	H(G)	H(G)	GH(G)	GH(G)	C(B)	CD(BC)	DE(BCD)	CDE(CDE)	DE(DEF)	CDE(EF)	CDE(F)
Nice 2 000 t	$2\,021.3^{A}$	$2\,021.4^{A}$	$2\,021.3^{A}$	$2\,021.4^{A}$	$2\,069.1^{A}$	$2\,084.8^{A}$	$2\,068.5^{A}$	$2\,104.5^{A}$	$2\,070.1^{A}$	$2\,070.6^{A}$	$2\,086.1^{A}$
Path 2 000 t	694.25^{E}_{EF}	694.15^{E}_{EF}	698.24^{E}_{EF}	733.44^{E}_{EF}	$1\,006.9^{B-E}$	997.84^{B-E}	$1\,132.0^{BCD}$	$1\,254.1^{B}$	$1\,316.3^{B}$	$1\,215.3^{BC}$	$1\,724.5^{A}$

Table 6.14: *A summary of the results for the GCS algorithm and its sorting variations described in §5.1.8 when applied to the* 1 170 *strip packing benchmark problem instances described in §6.1. The row labelled* 'Median H/OPT' *contains the median packing height for all benchmark instances listed in Table 6.1 as a percentage of the optimal packing height, or its lower bound if the optimal is not known. The row labelled* 'Low. Q. H/OPT' *contains the value of the lower quartile, the row labelled* 'Up. Q. H/OPT' *contains the values of the upper quartile and the interquartile range (in the row labelled* 'IQR') *is the difference between the two. The row labelled* 'Max. H/OPT' *contains the worst result achieved by the algorithms for all benchmark instances. The row labelled* 'Sig. Class' *are results obtained by means of a Nemenyi test, with the results from a Bonferroni t test in parentheses. Algorithms in the same group (indicated by letters) do not produce results that are significantly different. The row labelled* 'Bon. Class' *contains results obtained by means of a Bonferroni t test on the average times required to solve all instances of* 2 000 *items. The row labelled* 'Mean Rank' *contains the mean rank achieved by the algorithms in this set (a rank of 1 indicates that the algorithm packed to the lowest height for an instance), with their ranks shown in parentheses. If algorithms yielded the same packing height for an instance, the mean of the ranks that would have been awarded is used. The rows labelled* 'Nice 2 000 t' *and* 'Path 2 000 t' *show the time (in seconds) required for instances of* 2 000 *items (for the "nice" and "pathological" benchmark problem instances [156]), with results from a Bonferroni t test for each set presented as superscripts (results from the Tukey test appear as subscripts if they are different).*

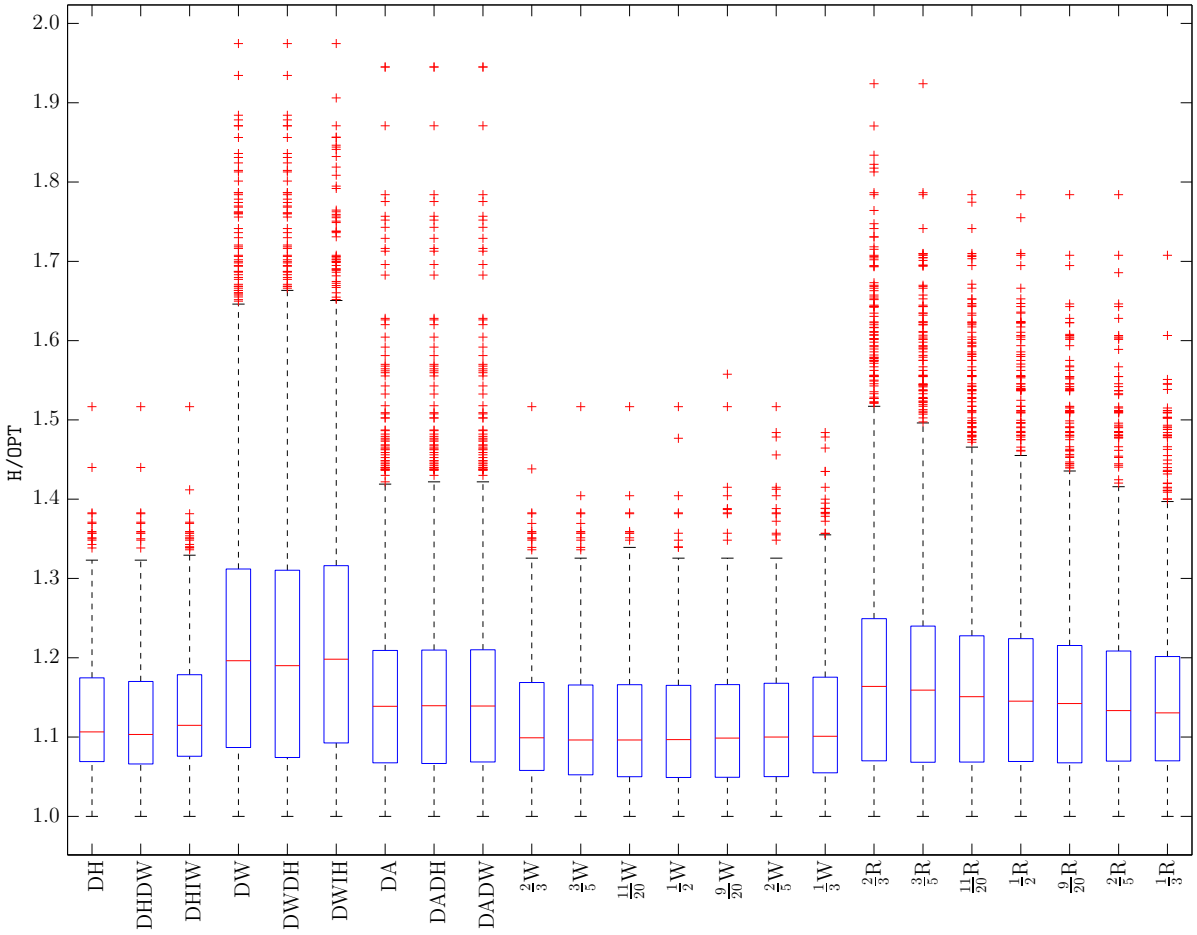


Figure 6.13: *Box plot of the distribution of results for the GCS algorithms described in §5.1.8 when applied to the 1 170 strip packing problem benchmark instances described in §6.1.*

three versions in which items are sorted according to area, and 7 variations of the xWDWDH and xRDWDH sorting methods. The distribution of the results for each of the sorting methods may be found in the form of box plots in Figure 6.14 and further results may be found in Table 6.15. A Friedman test yields $P = 0.000$ and an ANOVA yields $P < 0.0001$ (both tests are performed at a confidence interval of 95%), suggesting that the null hypothesis (that all algorithms yield similar results) may be rejected. The critical distance between algorithm's mean ranks is 1.06 for 24 algorithms, 1 170 benchmark instances and a confidence interval of 95%.

It is clear from the box plots in Figure 6.14 and the results listed in Table 6.15 that the oriented version of the BFLM algorithm typically packs items less densely than does the modified version for other sortings, excluding those sorting according to decreasing width (the BFLM algorithm is equivalent to the BFmLM(DWDH) algorithm). Sorting the items by decreasing area yields a better result for the BFmLM algorithm than it had for the BL, BLF and GCS algorithms. The BFmLM algorithms which sort items according to decreasing area yield the lowest upper quartile values and the lowest IQR. However, some of the xWDWDH algorithms yield better lower quartile values and maximum packing heights than the algorithms sorting according to decreasing area. The xRDWDH sorting method is not as effective as the other sorting methods, also yielding the worst upper quartile values and the maximum packing heights in this comparison set, better only than the oriented version of the original algorithm and the

	BFLM	DH	DHDW	DHIW	DW	DWDH	DWIH	DA	DADH	DADW	$\frac{2}{3}$W	$\frac{3}{5}$W
Low. Q. H/OPT	105.5%	108.2%	107.9%	108.9%	106.5%	105.5%	106.8%	105.8%	105.6%	105.7%	106.3%	106.0%
Med. H/OPT	113.5%	112.3%	112.9%	113.1%	113.8%	113.5%	114.3%	109.4%	109.4%	109.4%	110.6%	110.2%
Up. Q. H/OPT	123.9%	117.4%	117.0%	117.8%	124.1%	123.9%	126.4%	113.8%	113.8%	113.8%	115.6%	115.1%
IQR	18.4%	9.2%	9.1%	8.9%	17.6%	18.4%	19.7%	8.0%	8.2%	8.2%	9.3%	9.1%
Max. H/OPT	191.6%	156.3%	156.3%	156.3%	191.6%	191.6%	192.2%	171.3%	171.3%	171.3%	156.3%	151.7%
Mean Rank	15.01 (20)	14.93 (19)	14.43 (18)	15.81 (22)	16.15 (23)	15.01 (20)	16.68 (24)	10.15 (5)	10.09 (4)	10.08 (3)	12.61 (17)	12.01 (12)
Sig. Class	C(A)	C(DEF)	C(EF)	B(CDE)	AB(A)	C(A)	A(A)	GH(G)	GH(G)	GH(G)	D(FG)	D(G)
Nice 5 000 t	2.3633^{IJ}	2.3797^{I}	2.3682^{I}	2.3740^{I}	2.3790^{I}	2.3741^{I}	2.3743^{I}	2.3272^{K}	2.3427^{JK}	2.3444^{JK}	4.8397^{A}	4.8465^{A}
Path 5 000 t	2.4973^{I}	2.3781^{J}	2.3831^{J}	2.3913^{J}	2.4990^{I}	2.5179^{I}	2.5149^{I}	2.4824^{I}	2.4831^{I}	2.4849^{I}	4.8011^{A}	4.8009^{A}
Bon. Class	IJK	K	K	JK	IJ	I	I	IJK	IJK	IJK	A	A

	$\frac{11}{20}$W	$\frac{1}{2}$W	$\frac{9}{20}$W	$\frac{2}{5}$W	$\frac{1}{3}$W	$\frac{2}{3}$R	$\frac{3}{5}$R	$\frac{11}{20}$R	$\frac{1}{2}$R	$\frac{9}{20}$R	$\frac{2}{5}$R	$\frac{1}{3}$R
Low. Q. H/OPT	105.2%	104.7%	104.6%	104.5%	104.5%	105.6%	105.4%	105.4%	105.4%	105.4%	105.9%	106.4%
Med. H/OPT	110.0%	110.0%	109.8%	109.6%	109.3%	110.7%	110.7%	110.4%	110.3%	110.1%	110.2%	110.5%
Up. Q. H/OPT	114.7%	114.5%	114.1%	114.0%	114.0%	120.2%	119.1%	118.1%	117.7%	117.0%	116.3%	115.5%
IQR	9.5%	9.8%	9.5%	9.5%	9.4%	14.6%	13.7%	12.7%	12.3%	11.6%	10.5%	9.1%
Max. H/OPT	151.7%	151.7%	151.7%	151.7%	152.2%	185.6%	184.4%	183.1%	182.5%	182.9%	181.3%	175.2%
Mean Rank	11.23 (8)	10.75 (7)	10.48 (6)	10.05 (2)	9.70 (1)	12.56 (16)	12.08 (14)	11.92 (11)	11.91 (10)	11.86 (9)	12.01 (13)	12.49 (15)
Sig. Class	EF(G)	FG(G)	GH(G)	GH(G)	H(G)	D(B)	D(CB)	DE(CBD)	DE(CDE)	DE(CDE)	D(DE)	D(DEF)
Nice 5 000 t	4.8410^{A}	4.8406^{A}	4.8402^{A}	4.8468^{A}	4.8403^{A}	2.5249^{H}	2.6421^{G}	2.7315^{F}	2.8501^{E}	2.9732^{D}	3.1258^{C}	3.3109^{B}
Path 5 000 t	4.8002^{A}	4.8016^{A}	4.8006^{A}	4.8018^{A}	4.8032^{A}	2.6435^{H}	2.7412^{G}	2.8292^{F}	2.9373^{E}	3.0519^{D}	3.1665^{C}	3.3763^{B}
Bon. Class	A	A	A	A	A	H	G	F	E	D	C	B

Table 6.15: *A summary of the results for the BFLM algorithm and the modified versions of the algorithm described in §5.1.9 when applied to the* 1 170 *strip packing benchmark problem instances described in §6.1. The row labelled* 'Median H/OPT' *contains the median packing height for all benchmark instances listed in Table 6.1 as a percentage of the optimal packing height, or its lower bound if the optimal is not known. The row labelled* 'Low. Q. H/OPT' *contains the value of the lower quartile, the row labelled* 'Up. Q. H/OPT' *contains the values of the upper quartile and the interquartile range (in the row labelled* 'IQR') *is the difference between the two. The row labelled* 'Max. H/OPT' *contains the worst result achieved by the algorithms for all benchmark instances. The row labelled* 'Sig. Class' *are results obtained by means of a Nemenyi test, with the results from a Bonferroni t test in parentheses. Algorithms in the same group (indicated by letters) do not produce results that are significantly different. The row labelled* 'Bon. Class' *contains results obtained by means of a Bonferroni t test on the average times required to solve all instances of* 5 000 *items. The row labelled* 'Mean Rank' *contains the mean rank achieved by the algorithms in this set (a rank of 1 indicates that the algorithm packed to the lowest height for an instance), with their ranks shown in parentheses. If algorithms yielded the same packing height for an instance, the mean of the ranks that would have been awarded is used. The rows labelled* 'Nice 5 000 t' *and* 'Path 5 000 t' *show the time (in seconds) required for instances of* 5 000 *items (for the "nice" and "pathological" benchmark problem instances [156]), with results from the Bonferroni t test for each set presented as superscripts.*

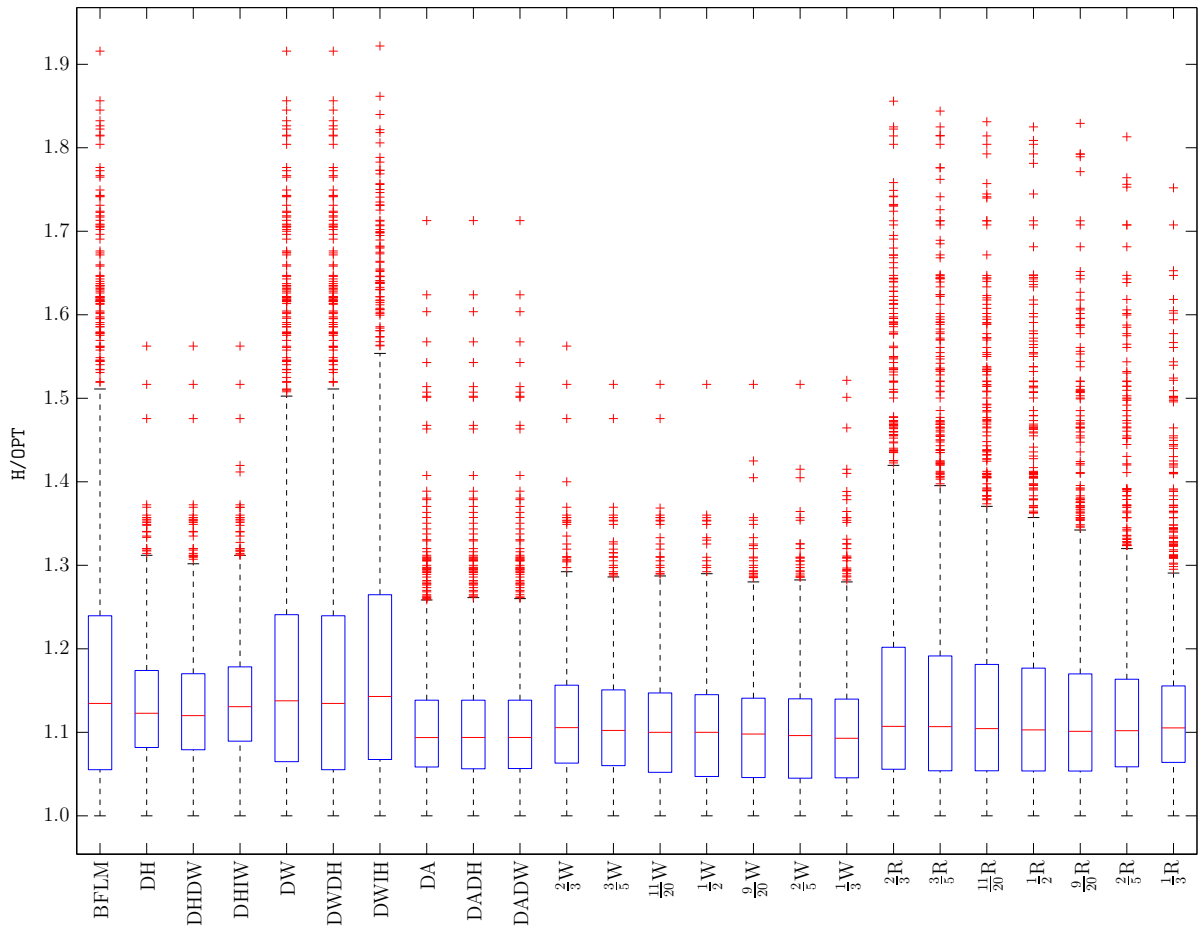


Figure 6.14: *Box plot of the distribution of results for the BFLM algorithms described in §5.1.9 when applied to the 1 170 strip packing problem benchmark instances described in §6.1.*

version that sort according to decreasing width. The variations sorting according to decreasing height, or decreasing width, yield the worst results. The poor mean rank achieved by the BFmLM$\frac{2}{3}$WDWDH algorithm is surprising considering that its distribution of results (see Figure 6.14) appears to be better than that of many of the xRDWDH algorithms.

The xWDWDH algorithms may yield a slightly better packing than the DA algorithms (not significantly better according to the Nemenyi test), but this comes at the expense of increased computation time. Applying an ANOVA at a 95% confidence level to the times results in $P < 0.0001$ for instances containing "nice" items, instances containing "pathological" items, and when the two are combined. The times required to solve benchmark instances containing 5 000 items may not be significantly different for the xWDWDH algorithms when compared to each other by means of a Bonferroni t test at a confidence level of 95%, but they are all significantly slower than the other algorithms in this comparison set. The xRDWDH algorithms all require significantly different solution times when compared to one another, with the solution inversely proportional to the fraction x. The DA, DH and original algorithms are not significantly different from one another (when instances containing "nice" and "pathological" items are combined), but they are significantly faster than the other algorithms. This suggests that the DADH (or DADW) version of the modified algorithm may be the best choice from this list due to the fact that it typically yields a packing height close to that of the $\frac{1}{3}$WDWDH variation, but requiring only half the computation time.

6.4.7 The BFTN Algorithm

This subsection is dedicated to the results obtained by applying the tallest neighbour variation of the BF algorithm by Burke *et al.* [22], and the modifications presented in §5.1.9, to the 1 170 benchmark instances of §6.1. There are three versions in which items are sorted according to area, and 7 variations of the xWDWDH and xRDWDH sorting methods. The distribution of the results for each of the sorting methods may be found in the form of box plots in Figure 6.15 and further results may be found in Table 6.16. A Friedman test yields $P = 0$ and an ANOVA yields $P < 0.0001$, suggesting that the null hypothesis (that all algorithms yield similar results) may be rejected. The critical distance between mean ranks for the Nemenyi test is 1.06.

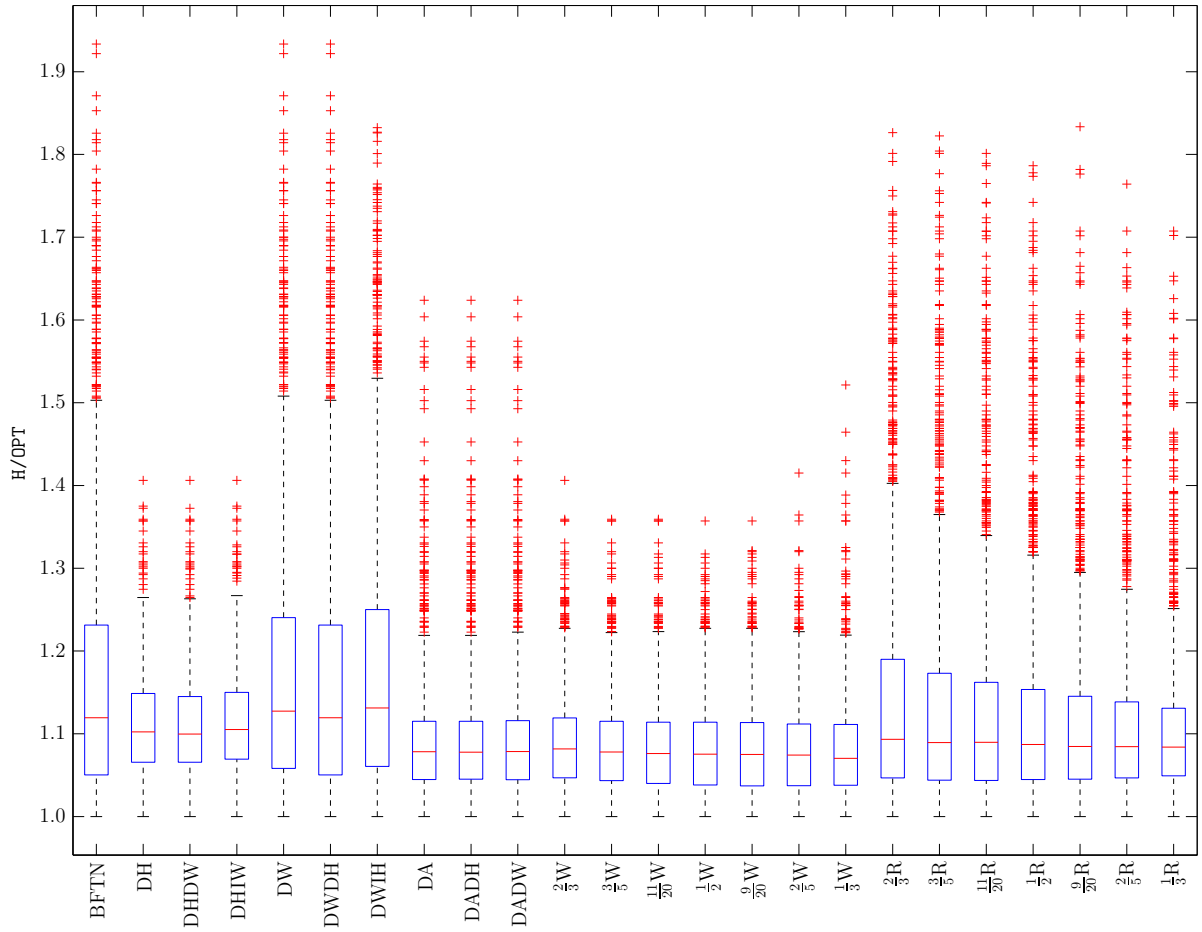


Figure 6.15: *Box plot of the distribution of results for the BFTN algorithms described in §5.1.9 when applied to the 1 170 strip packing problem benchmark instances described in §6.1.*

These results are very similar to those for the BFLM algorithm when taking a broad view. The oriented version of the original algorithm performs the worst in most cases, as do the modified versions where items are sorted according to decreasing width (the BFTN and BFmTN(DWDH) algorithms are equivalent), clearly visible when comparing the distributions of the results in Figure 6.15. Table 6.16 shows that the median for the BFTN algorithm is larger than the upper quartile values of the DA and xWDWDH algorithms. It also achieves the largest IQR of all algorithms in this comparison set and the highest maximum packing height. The xRDWDH algorithms follow the BFTN algorithm in the rankings. The $\frac{2}{3}$RDWDH variation is the worst,

	BFTN	DH	DHDW	DHIW	DW	DWDH	DWIH	DA	DADH	DADW	$\frac{2}{3}$W	$\frac{3}{5}$W
Low. Q. H/OPT	105.0%	106.6%	106.6%	106.9%	105.8%	105.0%	106.1%	104.5%	104.5%	104.5%	104.7%	104.3%
Med. H/OPT	111.9%	110.2%	110.0%	110.5%	112.7%	111.9%	113.1%	107.8%	107.8%	107.8%	108.2%	107.8%
Up. Q. H/OPT	123.1%	114.9%	114.5%	115.0%	124.0%	123.1%	125.0%	111.5%	111.5%	111.6%	111.9%	111.5%
IQR	18.1%	8.3%	7.9%	8.1%	18.2%	18.1%	18.9%	7.0%	7.0%	7.1%	7.2%	7.2%
Max. H/OPT	193.3%	140.6%	140.6%	140.6%	193.3%	193.3%	183.2%	162.4%	162.4%	162.4%	140.6%	135.9%
Mean Rank	16.27 (21)	14.42 (19)	14.10 (18)	14.90 (20)	17.31 (23)	16.27 (21)	17.83 (24)	10.17 (6)	10.23 (7)	10.14 (5)	11.57 (10)	10.91 (9)
Sig. Class	B(A)	CD(FG)	D(FG)	C(EF)	A(A)	B(A)	A(A)	IJ(GH)	IJ(GH)	IJ(GH)	GH(H)	HI(H)
Nice 5 000 t	2.3715^{IJ}	2.3895^{IJ}	2.4243^{HIJ}	2.5061^{HI}	2.4106^{HIJ}	2.3997^{HIJ}	2.3936^{HIJ}_{IJ}	2.3370^{J}	2.3529^{J}	2.3540^{J}	4.8449^{A}	4.8514^{A}
Path 5 000 t	2.5092^{I}	2.3737^{J}	2.3955^{J}	2.4016^{J}	2.4994^{I}	2.5194^{I}	2.5190^{I}	2.5069^{I}	2.5094^{I}	2.5106^{I}	4.8138^{A}	4.8079^{A}
Bon. Class	H	H	H	H	H	H	H	H	H	H	A	A

	$\frac{11}{20}$W	$\frac{1}{2}$W	$\frac{9}{20}$W	$\frac{2}{5}$W	$\frac{1}{3}$W	$\frac{2}{3}$R	$\frac{3}{5}$R	$\frac{11}{20}$R	$\frac{1}{2}$R	$\frac{9}{20}$R	$\frac{2}{5}$R	$\frac{1}{3}$R
Low. Q. H/OPT	104.0%	103.8%	103.7%	103.7%	103.8%	104.7%	104.4%	104.4%	104.5%	104.5%	104.7%	104.9%
Med. H/OPT	107.6%	107.5%	107.5%	107.4%	107.0%	109.3%	108.9%	109.0%	108.7%	108.5%	108.4%	108.4%
Up. Q. H/OPT	111.4%	111.4%	111.3%	111.2%	111.1%	119.0%	117.3%	116.2%	115.3%	114.5%	113.8%	113.1%
IQR	7.4%	7.6%	7.6%	7.5%	7.3%	14.3%	12.9%	11.8%	10.9%	10.0%	9.2%	8.2%
Max. H/OPT	135.9%	135.7%	135.7%	141.5%	152.2%	182.6%	182.2%	180.1%	178.6%	183.3%	176.4%	170.8%
Mean Rank	10.24 (8)	9.91 (4)	9.77 (3)	9.49 (2)	9.11 (1)	13.20 (17)	12.72 (16)	12.50 (15)	12.36 (14)	12.24 (13)	12.11 (11)	12.18 (12)
Sig. Class	IJ(H)	J(H)	JK(H)	JK(H)	K(H)	E(B)	EF(BC)	EF(BCD)	F(CDE)	FG(DEF)	FG(EF)	FG(FG)
Nice 5 000 t	4.8458^{A}	4.8451^{A}	4.8451^{A}	4.8507^{A}	4.8459^{A}	2.5369^{GH}	2.6542^{FG}	2.7440^{EF}	2.8587^{DE}	2.9831^{D}	3.1446^{C}	3.3254^{B}
Path 5 000 t	4.8083^{A}	4.8082^{A}	4.8081^{A}	4.8077^{A}	4.8078^{A}	2.6521^{H}	2.7531^{G}	2.8398^{F}	2.9484^{E}	3.0640^{D}	3.1783^{C}	3.3668^{B}
Bon. Class	A	A	A	A	A	G	F	F	E	D	C	B

Table 6.16: *A summary of the results for the BFTN algorithm and the modified versions of the algorithm described in §5.1.9 when applied to the* 1 170 *strip packing benchmark problem instances described in §6.1. The row labelled* 'Median H/OPT' *contains the median packing height for all benchmark instances listed in Table 6.1 as a percentage of the optimal packing height, or its lower bound if the optimal is not known. The row labelled* 'Low. Q. H/OPT' *contains the value of the lower quartile, the row labelled* 'Up. Q. H/OPT' *contains the values of the upper quartile and the interquartile range (in the row labelled* 'IQR') *is the difference between the two. The row labelled* 'Max. H/OPT' *contains the worst result achieved by the algorithms for all benchmark instances. The row labelled* 'Sig. Class' *are results obtained by means of a Nemenyi test, with the results from a Bonferroni t test in parentheses. Algorithms in the same group (indicated by letters) do not produce results that are significantly different. The row labelled* 'Bon. Class' *contains results obtained by means of a Bonferroni t test on the average times required to solve all instances of* 5 000 *items. The row labelled* 'Mean Rank' *contains the mean rank achieved by the algorithms in this set (a rank of 1 indicates that the algorithm packed to the lowest height for an instance), with their ranks shown in parentheses. If algorithms yielded the same packing height for an instance, the mean of the ranks that would have been awarded is used. The rows labelled* 'Nice 5 000 t' *and* 'Path 5 000 t' *show the time (in seconds) required for instances of* 5 000 *items (for the "nice" and "pathological" benchmark problem instances [156]), with results from the Bonferroni t test for each set presented as superscripts (results from the Tukey test appear as subscripts if they are different).*

with the rankings improving as x decreases (excluding the case where $x = \frac{1}{3}$). The three DA algorithms are significantly better than the xRDWDH algorithms, and are also significantly better than the $\frac{2}{3}$WDWDH algorithm, but they are not significantly different compared to the remaining xWDWDH algorithms, excluding the $\frac{1}{3}$WDWDH variation which achieves the best rank in this comparison set. The DA algorithms yield upper quartile values that are lower than the median of the BFTN algorithm, the lowest IQR and maximum packing heights lower than those of the original algorithm and the xRDWDH algorithms. However, some of the xWDWDH algorithms yield lower first quartile values than the DA algorithms, and lower maximum packing heights, resulting in them achieving the best rankings. The $\frac{1}{3}$WDWDH algorithm yields the best ranking of all algorithms in this comparison set and is significantly better than 14 of the other algorithms in this set.

As was shown in the previous section on the results for the BFLM algorithm and its variations, the improved packing by the xWDWDH algorithm comes at the expense of increased computation time. An ANOVA applied to the computation times yields $P < 0.0001$ for the instances containing only "nice" items, only "pathological" items and when the two sets are combined (suggesting that the algorithms do not all require similar computation times to solve large problem instances). A Bonferroni t test[6] on the times suggests that the xWDWDH algorithms are not significantly different in terms of computation time, but that the xRDWDH algorithms are significantly faster, becoming faster as the value of x increases. The original and nine DA, DH and DW algorithms are not significantly different in terms of the computation time required to solve large problem instances (when "nice" and "pathological" items are combined, the DH algorithms are significantly faster for "pathological" items) and are the ten fastest algorithms in this comparison set, by a significant margin. This makes for a difficult decision in deciding between the DADW or $\frac{1}{3}$WDWDH algorithms as the best in the comparison set because the DADW algorithm yields very good results. Therefore, both algorithms will be compared to the algorithms from other sets in order to determine which is the best free plane-packing heuristic.

6.4.8 The BFSN Algorithm

The results obtained when applying the BFSN algorithm by Burke *et al.* [22] (see §5.1.9) to the benchmark instances described in §6.1 are reported in this subsection. There are three variations of the modified BFSN algorithm that sort items according to decreasing area and seven variations of the xWDWDH and xRDWDH algorithms each. The distribution of the results for each of the sorting methods may be found in the form of box plots in Figure 6.16 and further results may be found in Table 6.17. An ANOVA applied to the results yields $P < 0.0001$ and a Friedman test applied to the data yields $P = 0$, suggesting that the null hypothesis (that all algorithms yield similar packings) may be rejected.

The box plots in Figure 6.16 show a pattern similar to that of the box plots of the results for the BFLM and BFTN algorithms and their variations. The original oriented version of the algorithm yields the worst result, with the xRDWDH algorithms performing slightly better. In this case, the DA algorithms perform better in relation to the xWDWDH algorithms than for the BFLM and BFTN algorithms. In fact, the DA algorithms yield better median, upper quartile and IQR values than the xWDWDH algorithms. A Nemenyi test performed on the algorithms in this comparison set suggests that the DA algorithms are not significantly different from each

[6]There is a single discrepancy between the Bonferroni t test and the Tukey test. The Tukey test found a significant difference between the DWIH and $\frac{2}{3}$RDWDH variations that the Bonferroni t test was not able to find.

	BFSN	DH	DHDW	DHIW	DW	DWDH	DWIH	DA	DADH	DADW	$\frac{2}{3}$W	$\frac{3}{5}$W
Low. Q. `H/OPT`	108.3%	109.9%	109.6%	110.4%	109.7%	108.3%	109.7%	107.8%	107.9%	107.8%	108.4%	108.0%
Med. `H/OPT`	116.6%	114.9%	114.8%	115.5%	117%	116.6%	117.6%	112.2%	112.3%	112.3%	113.6%	113.2%
Up. Q. `H/OPT`	127.5%	120.4%	120.1%	120.8%	127.2%	127.5%	128.6%	116.9%	116.9%	117.0%	118.8%	118.0%
IQR	19.2%	10.4%	10.5%	10.3%	17.5%	19.2%	18.9%	9.1%	9.0%	9.1%	10.4%	10.1%
Max. `H/OPT`	192.2%	156.3%	156.3%	156.3%	192.2%	192.2%	192.2%	171.3%	171.3%	171.3%	156.3%	151.7%
Mean Rank	14.78 (20)	14.67 (19)	14.15 (18)	15.44 (22)	15.62 (23)	14.78 (20)	16.17 (24)	10.20 (3)	10.16 (2)	10.15 (1)	12.34 (13)	11.63 (9)
Sig. Class	BC(A)	BC(DE)	C(DEF)	AB(CD)	A(A)	BC(A)	A(A)	H(G)	H(G)	H(G)	DE(EFG)	EF(FG)
Nice 5 000 t	2.2166^{L}	2.3818^{IJK}	2.3858^{IJ}	2.3867^{IJ}	2.3942^{IJ}	2.3989^{IJ}	2.4029^{I}	2.3408^{K}	2.3586^{JK}	2.3611^{IJK}	4.8569^{A}	4.8591^{A}
Path 5 000 t	2.3454^{L}	2.3813^{K}	2.3952^{K}	2.4007^{K}	2.5121^{IJ}	2.5279^{I}	2.5229^{I}	2.4920^{J}	2.4922^{J}	2.4936^{J}	4.8078^{A}	4.8082^{A}
Bon. Class	K	J	J	J	I	I	I	IJ	IJ	IJ	A	A

	$\frac{11}{20}$W	$\frac{1}{2}$W	$\frac{9}{20}$W	$\frac{2}{5}$W	$\frac{1}{3}$W	$\frac{2}{3}$R	$\frac{3}{5}$R	$\frac{11}{20}$R	$\frac{1}{2}$R	$\frac{9}{20}$R	$\frac{2}{5}$R	$\frac{1}{3}$R
Low. Q. `H/OPT`	107.5%	107.3%	107.1%	107.1%	107.1%	108.3%	108.1%	108.1%	108.0%	108.0%	108.3%	108.5%
Med. `H/OPT`	113.0%	112.9%	112.9%	113.0%	112.8%	114.3%	114.0%	113.8%	114.1%	113.7%	113.6%	113.6%
Up. Q. `H/OPT`	117.8%	117.6%	117.4%	117.8%	117.8%	123.0%	122.5%	121.4%	120.9%	120.1%	119.8%	119.0%
IQR	10.3%	10.4%	10.3%	10.7%	10.6%	14.7%	14.4%	13.3%	12.9%	12.1%	11.5%	10.5%
Max. `H/OPT`	151.7%	151.7%	151.7%	151.7%	184.4%	192.2%	192.2%	192.2%	192.2%	192.2%	192.2%	184.4%
Mean Rank	10.97 (8)	10.46 (6)	10.35 (4)	10.48 (7)	10.45 (5)	12.86 (17)	12.45 (14)	12.24 (10)	12.30 (12)	12.27 (11)	12.50 (15)	12.58 (16)
Sig. Class	FG(G)	GH(G)	GH(G)	GH(G)	GH(G)	D(B)	D(BC)	DE(BC)	DE(BCD)	DE(CD)	D(CD)	D(DE)
Nice 5 000 t	4.8506^{A}	4.8554^{A}	4.8561^{A}	4.8566^{A}	4.8484^{A}	2.5508^{H}	2.6667^{G}	2.7617^{F}	2.8718^{E}	2.9892^{D}	3.1434^{C}	3.3280^{B}
Path 5 000 t	4.8070^{A}	4.8082^{A}	4.8071^{A}	4.8072^{A}	4.8083^{A}	2.6615^{H}	2.7591^{G}	2.8446^{F}	2.9511^{E}	3.0657^{D}	3.1801^{C}	3.3671^{B}
Bon. Class	A	A	A	A	A	H	G	F	E	D	C	B

Table 6.17: *A summary of the results for the BFSN algorithm and the modified versions of the algorithm described in §5.1.9 when applied to the* 1 170 *strip packing benchmark problem instances described in §6.1. The row labelled* 'Median `H/OPT`' *contains the median packing height for all benchmark instances listed in Table 6.1 as a percentage of the optimal packing height, or its lower bound if the optimal is not known. The row labelled* 'Low. Q. `H/OPT`' *contains the value of the lower quartile, the row labelled* 'Up. Q. `H/OPT`' *contains the values of the upper quartile and the interquartile range (in the row labelled* 'IQR'*) is the difference between the two. The row labelled* 'Max. `H/OPT`' *contains the worst result achieved by the algorithms for all benchmark instances. The row labelled* 'Sig. Class' *are results obtained by means of a Nemenyi test, with the results from a Bonferroni t test in parentheses. Algorithms in the same group (indicated by letters) do not produce results that are significantly different. The row labelled* 'Bon. Class' *contains results obtained by means of a Bonferroni t test on the average times required to solve all instances of* 5 000 *items. The row labelled* 'Mean Rank' *contains the mean rank achieved by the algorithms in this set (a rank of 1 indicates that the algorithm packed to the lowest height for an instance), with their ranks shown in parentheses. If algorithms yielded the same packing height for an instance, the mean of the ranks that would have been awarded is used. The rows labelled* 'Nice 5 000 t' *and* 'Path 5 000 t' *show the time (in seconds) required for instances of* 5 000 *items (for the "nice" and "pathological" benchmark problem instances [156]), with results from the Bonferroni t test for each set presented as superscripts.*

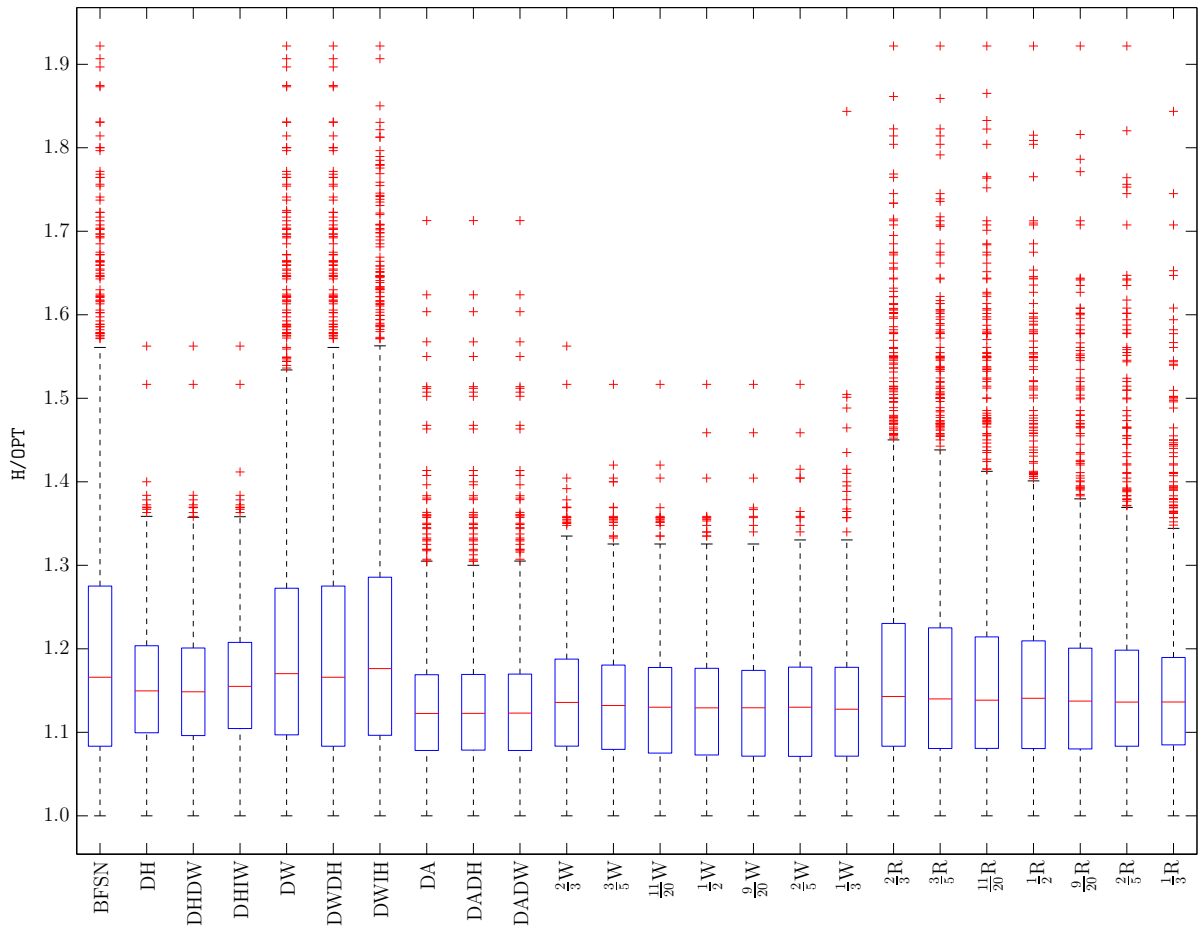


Figure 6.16: *Box plot of the distribution of results for the BFSN algorithms described in §5.1.9 when applied to the 1 170 strip packing problem benchmark instances described in §6.1.*

other, nor from five of the seven xWDWDH variations (they are significantly better than the $\frac{2}{3}$WDWDH and $\frac{3}{5}$WDWDH variations). The Nemenyi test suggests that the BFSN algorithm is significantly worse than the other algorithms (excluding the three related DW algorithms) in this comparison set. It also suggests that the three DA algorithms are significantly better than 10 of the other algorithms in this comparison set.

An ANOVA on the computation time results yields $P < 0.0001$ for the data sets of 5 000 items when "nice" items are separated from "pathological" items and when they are combined, suggesting that one may reject the null hypothesis that all algorithms require similar computation times to find solutions. The original oriented algorithm proves to be significantly faster than the new algorithms. The results are similar to the results of the other modified BF algorithms, with the DA algorithms significantly faster for a similar packing height to the xWDWDH algorithms.

6.4.9 Identification of the Best Plane-Packing Heuristic

In order to identify the best plane-packing algorithm, the best algorithm from each comparison set from the previous subsections are compared to one another. This may include the fastest algorithm and/or the algorithm that packs the most densely. While pseudolevel algorithms

adhere to an additional restriction (the packing of levels), the best of these are included for the sake of interest.

The fastest algorithms from the pseudolevel algorithms that yield guillotine feasible layouts in §6.3.1 are the SAS algorithms (they are not significantly different in speed), of which the SASm algorithm yields significantly better results. The algorithm that typically yields the best packing heights for the benchmark instances is the SL_5 algorithm. Of those pseudolevel algorithms in §6.3.2 that do not guarantee a guillotine feasible layout, the SC algorithm yields the best results in terms of time and packing density.

During a search for the best free-packing plane algorithms in §6.4.1, the M algorithm was shown to yield the best packing results, not significantly better than the SPmF algorithms in terms of packing height, but two orders of magnitude faster in terms of computation time. The plane algorithms that yield a guillotine layout (excluding the GCS algorithm) in §6.4.2 were shown to be worse in terms of both packing quality and computation time when compared to the SL_5 algorithm and may thus be ignored.

The BL$\frac{1}{2}$WDWDH algorithm was shown to yield the best packing results, achieving a significant difference with respect to the second best algorithm, but the BL(DHDW) algorithm does yield good results at a significantly better computation time and is therefore included in this comparison. The BLF algorithm yields five algorithms that showed no significant difference for the position as typically yielding the best packing heights. The BLF$\frac{2}{5}$WDWDH algorithm is selected to represent these three algorithms as its output results in the lowest mean rank. In this comparison set the algorithms typically yielding the best packing height belong to the subset yielding the fastest computation times. Therefore, no other algorithm needs to be selected to represent the fastest of the set. Of the GCS algorithms in §6.4.5, the GCS$\frac{1}{2}$WDWDH algorithm yields the best packing results. The BFmLM$\frac{1}{3}$WDWDH algorithm typically yields the best packing height of the algorithms in §6.4.6, but the BFmLM(DADW) yields good packing results at significantly lower computation times; hence both algorithms are included in this comparison. The results are similar for the BFmTN algorithms in §6.4.7, and hence the BFmTN$\frac{1}{3}$WDWDH and BFmTN(DADW) algorithms are included in the comparison. Finally, the Nemenyi test ranked the fastest algorithms as the best in terms of packing height by their mean rank in §6.4.8 and the BFmSN(DADW) algorithm is selected to represent this set of algorithms.

A Friedman test on the results yields a zero P-value (and so does an ANOVA), suggesting that the null hypothesis that all algorithms yield similar results may be rejected. Two observations are immediately apparent. The results depicted in Figure 6.17 show that the M algorithm does not pack well in comparison with the remaining algorithms, and the GCS algorithm is prohibitively slow (it is the reason no benchmark instances with more than 2 000 items could be used for comparative purposes). Fortunately the M algorithm is not faster than some of the remaining algorithms that also pack more densely; hence the M algorithm may be ignored for further comparison purposes. The Nemenyi test suggests that the SL_5 and GCS$\frac{1}{2}$WDWDH algorithms are not significantly different in terms of packing heights achieved. Thus the GCS algorithm will be ignored in further comparisons, leaving the SASm and SL_5 as the two guillotine algorithms, the former for its speed and the latter for its packing density.

Table 6.19 aids in the elimination of algorithms. For example, the SC algorithm is ranked higher than the M algorithm in terms of both time and mean rank and therefore the M algorithm may be eliminated from further consideration. The BFmTN(DADW) algorithm is both significantly faster and packs significantly better than the M, BFmLM$\frac{1}{3}$WDWDH, BL(DHDW) and BL$\frac{1}{2}$WDWDH algorithms, and is not significantly slower than the BFmLM(DADW) or

	SASm	SL_5	SC	M	BL(DHDW)	BL($\frac{1}{2}$W)	BLF($\frac{2}{5}$W)
Low. Q. H/OPT	109.3%	106.1%	105.8%	110.2%	111.3%	109.6%	103.9%
Med. H/OPT	114.4%	109.5%	109.0%	118.8%	116.7%	114.4%	107.5%
Up. Q. H/OPT	123.1%	116.2%	114.7%	137.0%	123.5%	119.5%	113.2%
IQR	13.77%	10.09%	8.98%	26.81%	12.19%	9.84%	9.28%
Max. H/OPT	195.1%	169.2%	153.2%	234.9%	168.3%	151.7%	151.7%
Mean Rank	9.78 (11)	6.72 (8)	6.03 (6)	10.57 (13)	10.24 (12)	9.11 (10)	4.43 (3)
Sig. Class	B(B)	E(DE)	FG(EF)	A(A)	AB(B)	C(C)	H(GH)
Nice 2 000 t	0.1369^{H}	0.2672^{G}	0.2791^{F}	0.4843^{E}	1.2473^{C}	1.5018^{B}	4.1914^{A}
Path 2 000 t	0.1518^{F}	0.2705^{EF}	0.3479^{DEF}	0.4825^{DE}	1.3395^{C}	1.6000^{B}	4.7251^{A}
Bon. Class	F	EF	E	D	C	B	A

	GCS($\frac{1}{2}$W)	BFmLM(DADW)	BFmLM($\frac{1}{3}$W)	BFmTN(DADW)	BFmTN($\frac{1}{3}$W)	BFmSN(DADW)
Low. Q. H/OPT	104.9%	106.0%	104.6%	104.6%	104.0%	108.1%
Med. H/OPT	109.7%	109.5%	109.4%	107.9%	107.2%	112.5%
Up. Q. H/OPT	116.5%	113.9%	114.0%	111.6%	111.2%	117.1%
IQR	11.59%	7.88%	9.46%	7.03%	7.19%	8.92%
Max. H/OPT	151.7%	171.3%	152.2%	162.4%	152.2%	171.3%
Mean Rank	6.40 (7)	5.99 (5)	5.55 (4)	4.39 (2)	3.71 (1)	8.09 (9)
Sig. Class	EF(EF)	FG(EF)	G(FG)	H(GH)	I(H)	D(CD)
Nice 2 000 t	2 021.3	0.2736^{G}	0.5338^{D}	0.2758^{GF}	0.5343^{D}	0.2767^{GF}
Path 2 000 t	694.25	0.2918^{EF}	0.5396^{D}	0.2933^{EF}	0.5397^{D}	0.2935^{EF}
Bon. Class	n/i	EF	D	EF	D	EF

Table 6.18: *A summary of the results for the best algorithms from the set of pseudolevel and plane packing algorithms when applied to the* 1 170 *strip packing benchmark problem instances described in §6.1. The headings* (xW) *(where x is a fraction) are abbreviations of xWDWDH. The row labelled* 'Median H/OPT' *contains the median packing height for all benchmark instances listed in Table 6.1 as a percentage of the optimal packing height, or its lower bound if the optimal is not known. The row labelled* 'Low. Q. H/OPT' *contains the value of the lower quartile, the row labelled* 'Up. Q. H/OPT' *contains the values of the upper quartile and the interquartile range (in the row labelled* 'IQR') *is the difference between the two. The row labelled* 'Max. H/OPT' *contains the worst result achieved by the algorithms for all benchmark instances. The row labelled* 'Sig. Class' *are results obtained by means of a Nemenyi test, with the results from a Bonferroni t test in parentheses. Algorithms in the same group (indicated by letters) do not produce results that are significantly different. The row labelled* 'Bon. Class' *contains results obtained by means of a Bonferroni t test on the average times required to solve all instances of* 5 000 *items (hence the GCS results are not included). The row labelled* 'Mean Rank' *contains the mean rank achieved by the algorithms in this set (a rank of 1 indicates that the algorithm packed to the lowest height for an instance), with their ranks shown in parentheses. If algorithms yielded the same packing height for an instance, the mean of the ranks that would have been awarded is used. The rows labelled* 'Nice 2 000 t' *and* 'Path 2 000 t' *show the time (in seconds) required for instances of* 2 000 *items (for the "nice" and "pathological" benchmark problem instances [156]).*

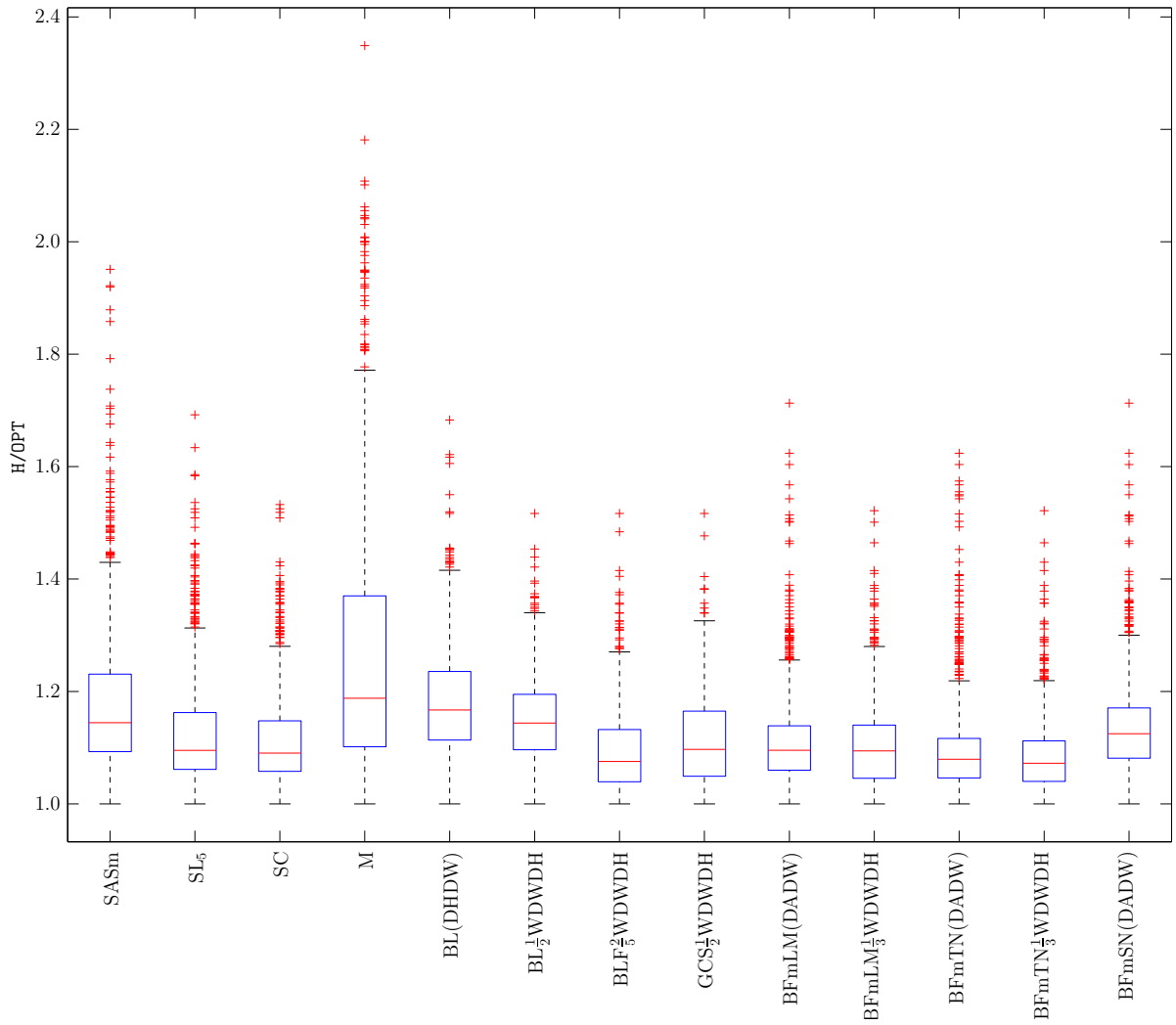


Figure 6.17: *Box plot of the distribution of results for the best algorithms from each subsection when applied to the 1 170 strip packing problem benchmark instances described in §6.1.*

BFmSN(DADW) algorithms, but packs significantly better than them. It is not significantly better than the BLF$\frac{2}{5}$WDWDH algorithm in terms of packing height, but it is significantly faster. Therefore, the BFmTN(DADW) algorithm is better than all other plane-packing algorithms, excluding the BFmTN$\frac{1}{3}$WDWDH algorithm (which typically yields a significantly better packing).

The results in Table 6.20 may be used by a decision maker to come to a similar conclusion. The SASm algorithm is ranked best according to the strip packing efficiency for low values of ℓ (when computation time is important), and is followed by the SL_5 algorithm and the modified BF algorithms that sort items according to decreasing area. For high solution time importance the BL and BLF algorithms have poor ranks and the M and BL algorithms continue to perform poorly as the importance of time decreases. With an increase in computation time importance the rank of the SASm algorithm worsens, while the ranks of the BLF algorithm and that of the modified BF algorithms sorting items according to the strip width improves at the cost of a poorer rank for the pseudolevel algorithms. For very low computation time importance the ranks of the algorithms according to their efficiency approaches their ranks according to their mean packing height.

Algorithm	Mean t	Sig. Set	Rank	Sig. Set	Mean Rank	Algorithm
SASm	1.11	F	1	H	3.43	BFmTN$\frac{1}{3}$WDWDH
SL_5	2.35	EF	2	G	3.94	BFmTN(DADW)
BFmLM(DADW)	2.41	EF	3	G	4.26	BLF$\frac{2}{5}$WDWDH
BFmSN(DADW)	2.43	EF	4	F	5.18	BFmLM$\frac{1}{3}$WDWDH
BFmTN(DADW)	2.43	EF	5	F	5.46	BFmLM(DADW)
SC	2.64	E	6	F	5.47	SC
M	4.41	D	7	E	6.08	SL_5
BFmLM$\frac{1}{3}$WDWDH	4.82	D	8	D	7.44	BFmSN(DADW)
BFmTN$\frac{1}{3}$WDWDH	4.83	D	9	C	8.46	BL$\frac{1}{2}$WDWDH
BL(DHDW)	9.22	C	10	B	9.02	SASm
BL$\frac{1}{2}$WDWDH	11.66	B	11	A	9.51	BL(DHDW)
BLF$\frac{2}{5}$WDWDH	30.81	A	12	A	9.74	M

Table 6.19: *Ranks of the best algorithms for both time (for instances with 5 000 items) and mean rank due to packing height. The mean ranks were calculated for the packing heights over all* 1 170 *benchmark instances. The GCS algorithm is excluded due to its slow solution times. The significance sets were calculated by means of the Bonferroni t test (and confirmed by means of a Tukey test) for the times and the Nemenyi test for the mean ranks. The critical distance for the Nemenyi test for these algorithms is 0.49.*

	SASm	SL_5	SC	M	BL(DHDW)	BL($\frac{1}{2}$W)	BLF($\frac{2}{5}$W)	BFmLM(DADW)	BFmLM($\frac{1}{3}$W)	BFmTN(DADW)	BFmTN($\frac{1}{3}$W)	BFmSN(DADW)
$\bar{t}$ Rank	1	2	6	7	10	11	12	3	8	5	9	4
$\ell = 1$	1	2	6	9	10	11	12	4	8	3	7	5
$\ell = 2$	1	3	6	9	10	11	12	4	8	2	7	5
$\ell = 3$	1	4	6	9	10	11	12	3	8	2	7	5
$\ell = 4$	1	4	6	9	10	11	12	3	8	2	7	5
$\ell = 5$	1	4	5	9	10	11	12	3	8	2	7	6
$\ell = 10$	1	4	5	9	11	10	12	3	8	2	7	6
$\ell = 20$	6	4	3	12	11	10	9	2	8	1	5	7
$\ell = 50$	9	6	4	12	11	10	8	3	5	1	2	7
$\ell = 100$	10	7	5	12	11	9	6	3	4	2	1	8
$\ell = 201$	10	7	6	12	11	9	3	5	4	2	1	8
$\bar{\alpha}$ Rank	10	7	6	12	11	9	2	5	4	3	1	8

Table 6.20: *Ranks of the best algorithms, based on the strip packing efficiency defined in §2.3.2. The row labelled '*$\bar{t}$ Rank*' contains the algorithms' ranks with respect to the mean solution time over the* 1 170 *strip packing benchmark instances. The row labelled '*$\bar{\alpha}$ Rank*' contains the algorithms' ranks with respect to the relative packing heights over the* 1 170 *benchmark instances.*

6.5 Chapter Summary

In this chapter the results of applying the algorithms presented in Chapters 3–5 on the benchmark instances in §6.1 were presented. First a brief description of the benchmarks was given in §6.1, in which the sources were listed, along with the other authors that have used the benchmark instances in order to test their algorithms. This was followed by a comparison of the NFDH, FFDH, BFDH and WFDH algorithms in §6.2.1 by means of box plots and the non-parametric Friedman and Nemenyi tests (as suggested by Demšar [40] for the comparison of

algorithms). These same tests were used to compare the KP algorithms in §6.2.2, the JOIN algorithms in §6.2.3 and the B2F algorithms in §6.2.4. This was followed by a comparison of the best algorithms from each subsection in order to find the best level-packing algorithm. There was no significant difference between the best two algorithms, namely the BFDHDW and B2FA_{10}DWDH algorithms, in terms of packing height (as measured by the mean ranks), nor in terms of time (when measured on benchmark instances with 5 000 items).

The known guillotine pseudolevel algorithms (the FC_{OG}, BFDH* and SAS algorithms) were compared to the new SASm, BFS and SL algorithms in §6.3.1. The SASm is an improvement on the SAS algorithm by Ntene [125, 127] and was found to be the fastest algorithm. The SL_5 algorithm yields the best results in terms of packing height, though not significantly better than the FC_{OG}DHDW algorithm, which is significantly slower. The comparison of free-packing pseudolevel algorithms in §6.3.2 suggested that the new SC algorithm is the best algorithm of this type in terms of both packing height and computation time.

The comparison of plane-packing algorithms began within each algorithm set. The first comparison in this section (see §6.4.1) was made between some free-packing algorithms, of which the M algorithm yields the best packing results within a reasonable time. The new SPmF algorithm is too slow to be considered useful when the M algorithm yields similar packing performance at significantly less time. The results of the guillotine-packing algorithms in §6.4.2 show how the speed and packing density are inversely proportional. However, these algorithms do not yield results that are better than the new SL_5 pseudolevel algorithm. The comparison of the various sorting methods for the BL algorithm in §6.4.3 suggest that the original decreasing width approach is significantly worse than the new xWDHDW sorting methods. The same was found for the BLF algorithm in §6.4.4 and the GCS algorithm in §6.4.5. The GCS algorithm, as programmed by the author, was shown to be prohibitively slow when solving large problem instances. The three oriented versions of the BF algorithm in §6.4.6–6.4.8 were outperformed significantly by modified versions of the algorithms that allowed them to be sorting-independent. In §6.4.9 the BFmTN algorithm was shown to yield the best results of all the plane-packing algorithms in terms of packing density.

CHAPTER 7

The Bin Packing Problem

Contents

In this chapter a brief introduction to the bin packing problem is given, followed by an introduction to the heuristics that have been developed for the two-dimensional *single bin size bin packing problem* (SBSBPP) in §7.1.1. This is followed by an introduction to the *multiple bin size bin packing problem* (MBSBPP) in §7.1.2 and a discussion on the algorithms that have been developed for this problem. Finally, a new heuristic approach towards solving the 2D MBSBPP is presented in §7.2.

7.1 Introduction

The bin packing problem is the problem of packing small items into larger bins, as opposed to the packing of small items into a single large bin with an unlimited height (which formed the basis of the previous four chapters). The aim is to pack the items in such a manner that the smallest bin area is required to accommodate them. Wäscher *et al.* [157, p. 1120] have six names for this problem, which are covered in detail in Table 2.2 in Chapter 2. The literature on packing problems is vast and presented in further detail in surveys such as those by Sweeney and Paternoster [151], Coffman *et al.* [31], Lodi *et al.* [106] and Wäscher *et al.* [157]. Some of the literature on the subject has been presented in §2.2.

In the remainder of this section selected work that has been done on solving these problems by means of heuristics are presented in order to provide the context of the new heuristic which is proposed for the MBSBPP later in this chapter. First, the relevant literature on the heuristics for the SBSBPP are presented and this is followed by selected literature on the MBSBPP.

7.1.1 Single Bin Size Bin Packing

In 1982, Chung *et al.* [28] published their work on the 2D bin packing problem (now called the SBSBPP since the paper by Wäscher *et al.* [157]) as a problem related to the "well-studied packing problems" [28, p. 67], namely the 1D bin packing problem and the 2D strip packing problem. They combined the two problems to solve the 2D SBSBPP by first packing all items into a strip of width equal to the width of the bins. They then treated the levels created by the FFDH algorithm (see 3.2.2) as 1D items that required packing into 1D bins (the width could now be ignored as they were all equal). This problem was solved via the first-fit decreasing algorithm by Johnson *et al.* [85] who called this two-phase algorithm the *hybrid first fit* (HFF) algorithm.

In the same year Bengtsson [14] published his packing algorithm for the case where items were allowed to be rotated and free packing was allowed. The algorithm begins by generating a list of items of size $2n$ consisting of the n items to be packed and their rotated versions. The algorithm then packs the first item in the list that is unpacked on the boundary of the bin, forming sections of items, where the items that follow are the tallest available items that, when packed and moved as low as possible, do not extend past the height of the first item in the section. This process continues until no further items fit into the bin. Then a new bin is opened and the process continues until all items have been packed. The bin with the most waste is selected and its items are returned to the unpacked pool of items. The remaining bin with the most waste is selected and an iterative packing procedure takes place that attempts to pack the unpacked items into the bin. If the result is no better than before, the bin is ignored for further repackings. If the waste decreases, but the bin is still the one with the most wasted space, it is ignored for further repackings. If a new packing is found that is better and results in the bin no longer containing the most wasted space, then the search procedure stops and the bin with the largest wasted space is selected for further packing.

In 1987 two papers were published on this problem, one that presented algorithms for packing items directly into bins and another using the idea of the hybrid approach to solve 2D SBSBP problems. Frenk and Galambos [52] adapted the NFD algorithm to pack items directly into bins and named the algorithm the *hybrid next-fit* (HNF) algorithm. Instead of first packing the items into a strip and then packing the strip levels into the bins, Frenk and Galambos packed the items directly into the last level to have been started, or started a new level in the bin if the item did not fit between the last packed item and the right-hand boundary of the bin, or started a new level in an empty bin if the item could not be packed into the current bin due to height restrictions. They did not use the two-phase packing strategy of Chung *et al.* [28].

The second paper took a more quantitative approach to the evaluation of a number of new algorithms for the SBSBPP. Berkey and Wang [16] designed the *finite next-fit* (FNF) algorithm, which also packs items directly into bin levels in a next-fit manner. They also designed the *finite first-fit* (FFF) algorithm which packs items directly into the bins in such a way that the item is placed into the first level of the first bin that has enough horizontal space to accommodate it. The *finite best-strip* (FBS) algorithm[1] is another hybrid approach which packs items into a strip by means of what Coffman and Shor [34] would later call the BFDH algorithm. The resulting levels are then packed into the bins in a BFD manner. Finally, Berkey and Wang suggested the *finite bottom-left* (FBL) algorithm which adapted the BLF algorithm by Chazelle [25] in order to pack the items into bins. They treated each bin as a hole, creating a new hole by allowing a new bin to accept items for packing when an item did not fit into existing holes or subholes.

[1] As Monaci [119, p. 37] and Lodi *et al.* [102, pp. 246] note, this should be called the *hybrid best-fit* (HBF) algorithm for the sake of uniformity.

However, due to the computational limitations of the time, they were unable to solve problem instances with more than 175 items. In order to overcome this problem, they proposed the *next bottom-left* (NBL) heuristic in which a new bin was considered when an item could not be packed into the current bin, and all previous bins were ignored for further packing. This next-fit approach reduced the number of subholes that their algorithm had to track, thereby reducing the memory required. They found that the FBS algorithm yielded the best results for their benchmark instances.

In 1999 Lodi *et al.* [103, 105] applied the two-phase approach to their *floor-ceiling* (FC) algorithms to effectively design the *hybrid floor-ceiling* (HFC) algorithm (although they did not call it such). These hybrid algorithms pack items into a strip using a level algorithm, and then repack the levels into bins using one of the NFD, FFD or BFD algorithms for 1D bin packing. They also approach the problem as a knapsack problem (previously mentioned in §3.2.4). The items are sorted according to decreasing height and the tallest item initialises a level. The remaining width of the strip is then filled according to an algorithm for the knapsack problem, where the aim is to maximise the area of items packed into the remaining space in the level, with the widths of the items used as the restricting factor (the total width of items may not exceed the width of the space remaining in the level). The 1D bin packing problem of packing the levels into bins is then solved to find the final packing. A third phase may be adopted to pack the items when some have been rotated. The *alternate directions* algorithm by Lodi *et al.* was designed to solve the SBSBPP when free packing is allowed. The algorithm packs items into bands as low as possible, alternating the direction of packing from "left to right" to "right to left" and back until the bin is either full or no further items remain. If a bin is full and items remain unpacked, then the same procedure is used to fill a new bin.

In 2003 Boschetti and Mingozzi [19] put forward an algorithm they called the HBP algorithm for solving the SBSBPP for items that are mixed in terms of being oriented or allowing rotation. It is an algorithm that iterates until a maximum number of iterations has been reached, or the solution requires a number of bins that is equal to a lower bound and is therefore optimal. The items are initially assigned a price according to their area, width, height or perimeter. During the iterations these prices are increased or decreased, according to the bins in which they were packed. These prices determine the initial order of items and the algorithm packs each bin until no further items may be added, before a new bin is considered. Once the items have been packed, the prices of those items in the second half of items are increased, while the prices of those in the first half are decreased by either a fixed (10% in their experiments) or random percentage. While Boschetti and Mingozzi call the HBP algorithm a heuristic [19, p. 138] its iterative nature combined with the changing of the prices of items is closer to that of a metaheuristic.

El Hayek *et al.* [67] published a heuristic called IMA in 2008. The algorithm determines all maximal areas in a bin, where maximal areas are rectangular empty regions in the bin that are not completely included in any other empty rectangular regions. These regions have the following two properties: each of the four edges of the maximal region coincides with either the boundary of the bin, or at least one edge of a packed item, and if the bottom-left corners of two maximal areas have the same location, then one area will have a greater width than the other, which in turn will have a greater height. The algorithm packs the items into the maximal areas in a best-fit manner, where the criterion for best fit is not only the width or area, but a weighted combination of four factors (utilising four weights q_1, q_2, q_3 and q_4 satisfying the constraint $q_1 + q_2 + q_3 + q_4 = 1$), including the ratio of the area of the item to the area of the

maximal area, the ratio of the horizontal length[2] of the item to the width of the maximal area, the ratio of the vertical length of the item to the height of the maximal area and the ratio of the sum of the squares of the horizontal and vertical lengths of the item to the sum of the squares of the width and height of the maximal area. The item/maximal area (that can accommodate the item) couple that yield the best score are then adopted for packing. A new bin is opened when no further items may be packed into the current bin. This process is completed for various values for the q_1, q_2, q_3 and q_4 weights and the best results are stored.

7.1.2 Multiple Bin Size Bin Packing

Friesen and Langston [54] introduced the repacking strategy called *first fit decreasing using largest bins, at end repack to smallest possible bins* (FFDLR) for 1D bin packing in 1986. In an attempt to minimise the wasted space in bins occupied by items, the strategy is to pack all items into the largest bins first and then to attempt repacking the items in these bins into smaller bins in the same order in which the bins were filled. They claim that this algorithm has a time complexity of $\mathcal{O}(n \log n + f \log b)$, where f is the number of bins that are filled after the initial packing, where b is the total number of bins and where n denotes the number of items packed, and established the asymptotic bound

$$\texttt{FFDLR}\,(\mathcal{L}) \leq \frac{3}{2}\,\texttt{OPT}\,(\mathcal{L}) + 1,$$

where $\texttt{FFDLR}(\mathcal{L})$ is the *bin consumption*[3] for the normalised item list $\mathcal{L}$ (where the maximum bin size and perhaps item size is 1) after the FFDLR procedure and $\texttt{OPT}(\mathcal{L})$ is the optimal bin consumption. They also introduced the *first fit decreasing using largest bins, but shifting as necessary* (FFDLS) strategy. Here, all items in a bin are shifted to the smallest bin that will hold them when items are packed into a bin containing another item larger than $\frac{1}{3}$ of the largest bin in size, such that the total size of the items is greater than or equal to $3/4$ of the size of the smaller bin. This shifting procedure is followed by the repacking strategy of the FFDLR algorithm once all items have been packed. They claim that the FFDLS strategy has an $\mathcal{O}(n \log n + n \log b)$ time complexity and established the asymptotic bound

$$\texttt{FFDLS}\,(\mathcal{L}) \leq \frac{4}{3}\,\texttt{OPT}\,(\mathcal{L}) + 3$$

for the strategy, where $\texttt{FFDLS}(\mathcal{L})$ is the sum of the sizes of the bins containing items from the list $\mathcal{L}$ after the FFDLS procedure and $\texttt{OPT}(\mathcal{L})$ denotes the optimal bin consumption.

In 2001 Chu and La [27] found worst-case performance ratios for four approximation algorithms for the 1D MBSBPP. The packing strategies are called the *largest object first with least absolute waste* (LFLAW), the *largest object first with least relative waste* (LFLRW), the *least absolute waste* (LAW) and the *least relative waste* (LRW) algorithms. The two *largest object first* (LF) algorithms allow only the bins of largest size to be packed before smaller bins are considered, while the other two algorithms consider any size of bin during the packing phase. The algorithms that pack according to the absolute waste choose the appropriate bin that would leave the least waste after packing an item, while the algorithms that pack according to relative waste pack into the bin for which the ratio of the waste to the bin size is smallest. Chu and La [27, p. 2072]

[2]The term *horizontal length* is used here instead of *width*, because if the item is rotated, then the width becomes the height. In this way confusion between the widths of rotated and oriented items may be avoided.

[3]This is the term coined by Chu and La [27, p. 2070] for the sum of the sizes (length, area, volume, *etc.*) of the bins that contain items after the packing is completed.

claim that the time complexity of the algorithm is $\mathcal{O}(n^2M)$, where M is the number of sizes of bins and n denotes the number of items packed.

Kang and Park [87] combined the FFDLR strategy with the FFD and BFD algorithms to design the *iterative first-fit decreasing* (IFFD) and *iterative best-fit decreasing* (IBFD) algorithms in 2003. The algorithms first pack the largest bins according to the FFD (BFD, respectively) algorithm and then attempt to repack the items in each bin into smaller bins. These algorithms achieve an optimal packing when the sizes of items and bins are exactly divisible.

Similar 2D problems have been solved via non-heuristic methods. Hopper [75] used the BLF algorithm in combination with a genetic algorithm to solve the MBSBPP, Yanasse *et al.* [159] used a pattern-generation algorithm to solve the similar multiple stock size stock cutting problem, and Pisinger and Sigurd [137] have used a branch-and-price algorithm to find exact solutions to the 2D MBSBPP with variable bin costs. However, there appear to be no simple heuristics to solve this problem. Therefore, the concept of packing large bins first and then repacking them into smaller bins is combined here with the hybrid packing approach to 2D bin packing in order to design a heuristic to solve this problem in a short time.

7.2 A New Heuristic for the MBSBPP

It is clear that the level and pseudolevel algorithms of §3 and §4 may be combined with any algorithm for the 1D SBSBPP to design hybrid algorithms for the 2D SBSBPP. However, the bin width may not be the same for all bins in the 2D MBSBPP. Hence the first fit procedure is used to pack the levels resulting from most level and pseudolevel strip packing heuristics into bins. Almost all algorithms yield levels that are decreasing in height as the height of the strip increases. The only algorithm that may not result in levels packed in order of decreasing height is the SAS algorithm[4] by Ntene and Van Vuuren [125, 127]. Therefore the levels are typically packed into bins in a FFD manner, except for the SAS algorithm where the levels are packed in a first-fit manner. The two-phase approach combined with the FFDLR strategy allows for the packing of multiple-size bins with the aim of minimising the bin consumption. This strategy consists of two stages.

During the first stage of the algorithm the bins are sorted by decreasing area, with equalities resolved by sorting by increasing perimeter. A strip packing is performed with the strip width taken as the width of the first bin in the list. The levels of the strip are then packed into the bin from the bottom upwards until no further levels fit. If the next bin in the list has the same bin width, the remaining strip levels may be packed into that bin. If there are unpacked levels and the next empty bin has a different width, another strip packing is performed with the unpacked items, taking the strip width as the width of the new bin. This process of packing one bin at a time continues until all items have been packed into bins.

During the second stage the bin with the lowest bin consumption is selected for repacking. This is different from the FFDLR algorithm which attempts to repack the bins in the same order in which they were packed. The reason for this change is the desire to allow the possibly smaller bins that are likely to contain a lower area of items to be emptied in order that these bins may be filled by the items of larger bins. If one were to repack the bins according to the order in which they were packed, an opportunity may be lost to repack the set of items in a large bin

[4]Consider the following situation. A new level is to be initialised and two wide items remain. One item is shorter, but wider than the other. The SAS algorithm would initialise the level with the shorter item, necessitating the creation of a new level of greater height for the taller item.

into a smaller bin, which may be emptied at a later stage. Therefore, when the bin with the lowest area of items is found, the list of bins is searched for the smallest empty bin whose area is at least the area of the items in the packed bin. A strip packing is performed for these items, setting the strip width equal to the width of the empty bin. If the strip height is not greater than the bin height, the items may be packed into the empty bin. The previously packed bin is now empty. However, if the strip height is greater than the bin height, the suitability of the previous bin in the bin list is investigated. Once an attempt has been made to repack the items, the process is repeated with the bin corresponding to the next largest item area. This process continues until all bins have been investigated for repacking. A pseudocode listing of this novel two-stage algorithm for the MBSBPP (2SMBSBP) may be found in Algorithm 7.1.

Algorithm 7.1 Two-stage algorithm for the MBSBPP (2SMBSBP)

Input: The list of items to be packed $\mathcal{I}$, the dimensions of the items $\langle w(\mathcal{I}_i), h(\mathcal{I}_i)\rangle$, the list of bins $\mathcal{B}$ and the dimensions of the bins $\langle w(\mathcal{B}_i), h(\mathcal{B}_i)\rangle$.

Output: A feasible packing of the items into the bins with the aim to minimise unutilised space in bins containing items.

1: sort $\mathcal{B}$ by decreasing area and increasing perimeter
2: **call** STAGEONE $(\mathcal{I}, \mathcal{B})$
3: **call** STAGETWO $(\mathcal{I}, \mathcal{B})$

Procedure 7.1.1 STAGEONE $(\mathcal{I}, \mathcal{B})$

1: $i \leftarrow 1$
2: **while** $\mathcal{I} \neq \emptyset$ **do**
3: **if** $w(\mathcal{B}_i) \neq \mathtt{W}$ **then**
4: $\mathtt{W} \leftarrow w(\mathcal{B}_i)$
5: perform a strip-packing with a strip packing algorithm from §3 or §4
6: let $\mathcal{L}$ be the list of unpacked levels and $\mathcal{PL} \leftarrow \emptyset$
7: **end if**
8: $j \leftarrow 1$
9: **while** $h(\mathcal{B}_i) - \sum_{k=1}^{|\mathcal{PL}|} h(\mathcal{PL}_k) >= h\left(\mathcal{L}_{|\mathcal{L}|}\right)$ **and** $\mathcal{L} \neq \emptyset$ **do**
10: **if** $h(\mathcal{B}_i) - \sum_{k=1}^{|\mathcal{PL}|} h(\mathcal{PL}_k) >= h(\mathcal{L}_j)$ **then**
11: $\mathcal{P} \leftarrow \mathcal{P} \cup \mathcal{I}_k \ \forall\ \mathcal{I}_k \in \mathcal{L}_j$, $\mathcal{I} \leftarrow \mathcal{I} \setminus \mathcal{I}_k \ \forall\ \mathcal{I}_k \in \mathcal{L}_j$
12: $\mathcal{PL} \leftarrow \mathcal{PL} \cup \mathcal{L}_j$, $\mathcal{L} \leftarrow \mathcal{L} \setminus \mathcal{L}_j$
13: **else**
14: $j \leftarrow j + 1$
15: **end if**
16: **end while**
17: $i \leftarrow i + 1$
18: **end while**

7.2.1 Worked Example

In order to illustrate the 2SMBSBP algorithm, the SAS algorithm is used to pack the items in $\mathcal{I}$ (Table 7.1) into the bin set $\mathcal{B}$, shown in Table 7.2. The bins are sorted by decreasing area. Hence a strip packing of the items in $\mathcal{I}$ is performed first with $\mathtt{W} = w(\mathcal{B}_1) = 15$. The bottom-most level has height $h(\mathcal{I}_1) = 15$. Therefore the items in the first level may be packed into $\mathcal{B}_1$. As none of the other levels may be packed into the bin, a strip packing is performed for

Procedure 7.1.2 StageTwo $(\mathcal{I}, \mathcal{B})$

```
let F be the set of bins containing items for which repacking has not been attempted
let E be the set of empty bins, i.e. E ← B \ F
let R be the set of new, previously empty bins whose items have been repacked
while F ≠ ∅ do
    i ← 1, Found ← false
    let F_s be the bin in F containing the smallest area of items
    while i ≤ |E| and not Found do
        if Area_b(E_i) ≥ Area_i(F_s) then
            W ← w(E_i)
            perform a strip-packing with an algorithm from §3 or §4
            if H ≤ h(E_i) then
                Found ← true
                R ← R ∪ E_i, E ← E ∪ F_s
                F ← F \ F_s, E ← E \ E_i
                i ← |E|
            else
                i ← i + 1
            end if
        end if
    end while
    if i > |E| and not Found then
        R ← R ∪ F_s, F ← F \ F_s
    end if
end while
```

the remaining items with $\mathtt{W} = w(\mathcal{B}_2) = 10$. Items $\mathcal{I}_3$ and $\mathcal{I}_4$ fit into the first level and $\mathcal{I}_{10}$ fits into the second. The bottom level fits into $\mathcal{B}_2$, hence the items are packed into that bin and, because $w(\mathcal{B}_3) = w(\mathcal{B}_2) = 10$, no further strip packing is required and the second level ($\mathcal{I}_{10}$) may be packed into $\mathcal{B}_3$. This intermediate packing is shown in Figure 7.1(a).

Item, $\mathcal{I}_i$	1	2	3	4	5	6	7	8	9	10
Height, $h(\mathcal{I}_i)$	15	9	8	7	1	6	6	1	3	4
Width, $w(\mathcal{I}_i)$	4	4	4	4	6	6	4	4	2	3

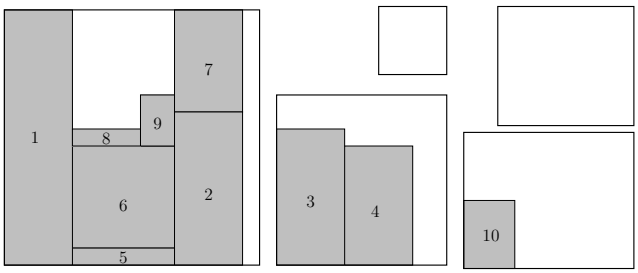
Table 7.1: *Dimensions of the items in $\mathcal{I}$.*

Bin, $\mathcal{B}_i$	1	2	3	4	5
Height, $h(\mathcal{B}_i)$	15	10	8	7	4
Width, $w(\mathcal{B}_i)$	15	10	10	8	4

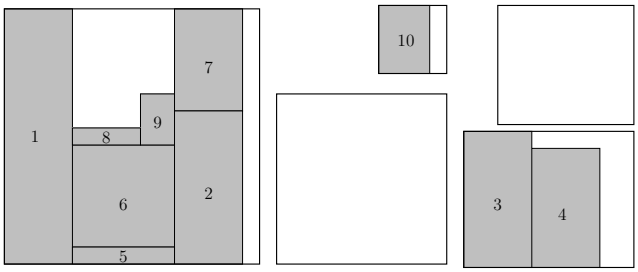
Table 7.2: *Dimensions of the bins in the bin set $\mathcal{B}$.*

Bin $\mathcal{B}_3$ has the smallest area of items, $A(\mathcal{I}(\mathcal{B}_3)) = 12$, which is less than the area of the smallest bin $A(\mathcal{B}_5) = 16$. Thus, a strip packing is performed for the items in $\mathcal{B}_3$ (*i.e.* $\mathcal{I}_{10}$) with $\mathtt{W} = w(\mathcal{B}_5) = 4$. Item $\mathcal{I}_{10}$ fits into $\mathcal{B}_5$ and therefore remains in it. Bin $\mathcal{B}_3$ is now empty and an attempt is made to pack the contents of $\mathcal{B}_2$ ($A(\mathcal{I}(\mathcal{B}_2)) = 60$) into a smaller bin. Bin $\mathcal{B}_4$ is the smallest empty bin, but has an area smaller than that of the items in $\mathcal{B}_2$, thus the strip width

is set equal to the width of $\mathcal{B}_3$. The resulting strip height is less than $h(\mathcal{B}_3) = 8$. Therefore the items remain in $\mathcal{B}_3$, and $\mathcal{B}_2$ is empty. The area of the items in $\mathcal{B}_1$ is $A(\mathcal{I}(\mathcal{B}_1)) = 172$, which is larger than $A(\mathcal{B}_2) = 100$. Hence no further repacking takes place. The final packing is shown in Figure 7.1(b).



(a) Intermediate Packing



(b) Final Packing

Figure 7.1: *Results obtained by the two-stage algorithm for the MBSBPP (2SMBSBP-SAS), using the SAS algorithm for strip packing.*

7.2.2 Worst-Case Time Complexity

The algorithm begins by sorting the bins according to decreasing area in line 1, resolving ties by sorting the bins according to increasing perimeter. If the sorting is performed by means of the merge sort algorithm, the time complexity of this step is $\mathcal{O}(b \log b)$, where b is the number of bins. The next step is to call Procedure 7.1.1, which performs the initial packing of items into bins. In the worst case every bin has a different size and only one item may be packed per bin. In that case the algorithm will attempt to pack a strip n times (the *while*-loop spanning lines 2–18), once for each new bin. Let the time complexity of the strip packing algorithm be $\mathcal{O}(s)$. The *while*-loop spanning lines 9–16 will attempt to pack every level into the bin, but in the worst case only one will fit. Therefore, this loop will have a worst-case time complexity of $\mathcal{O}(n)$, which is dominated by the time complexity of the strip packing algorithm. Hence, the time complexity of Procedure 7.1.1 is $\mathcal{O}(n\,s)$.

The second stage (Procedure 7.1.2), will execute the *while*-loop spanning lines 4–24 $\mathcal{O}(n)$ times. Finding the bin with the lowest area of items in line 6 is an $\mathcal{O}(n)$ step. This is followed by the search for the smallest bin that has the same or larger area than the items that are to be repacked. A *while*-loop spanning lines 7–20 is entered that attempts to pack the items into smaller bins. This loop may be executed $\mathcal{O}(b)$ times and contains a call to a strip packing algorithm (in line 10) which has a worst-case time complexity of $\mathcal{O}(s)$. If the packing allows the items to be packed into a new bin, then the items are moved into the new bin via a step that

has a time complexity of $\mathcal{O}(n)$. Therefore, Procedure 7.1.2 has a worst-case time complexity of $\mathcal{O}(n\,(n + b\,s + n)) = \mathcal{O}(n \max\{n, b\,s\})$. The time complexity of $\mathcal{O}(s)$ is no less than $\mathcal{O}(n)$ (sorted items may be packed in $\mathcal{O}(n)$ time by the NFDH algorithm), resulting in a worst case time complexity of $\mathcal{O}(b\,n\,s)$ for the second stage of the algorithm. Thus, the worst-case time complexity of the 2SMBSBP heuristic is

$$\mathcal{O}(b \log b + n\,s + b\,n\,s) = \mathcal{O}(b \max\{\log b, n\,s\})\,.$$

7.2.3 Adapting the 2SMBSBP Algorithm for Plane Algorithms

In §6.4.9, it was found that certain sorted lists of items yielded the best results for the 1 170 benchmark instances when packed by means of the new BFmTN algorithm of §5. It is desirable to adapt the 2SMBSBP algorithm for use by the sorting-independent BFmTN algorithm in an attempt to find better packings than level and pseudolevel algorithms would. Due to the level-packing nature of the algorithms in §3 and §4 it is possible to design a procedure that calls the existing strip packing algorithms with no modifications, because the packing of levels takes the bin ceiling into account. However, the plane-packing algorithms do not pack items into levels and this requires modifications to be made to the BFmTN algorithm in order for it to pack items into the bins of multiple sizes. The new algorithm must take into account the fact that there is a height limit to which items may be packed.

This algorithm has a structure very similar to that of Algorithm 7.1, but instead of calling the BFmTN algorithm, a new, but very similar, algorithm is called. This algorithm is modified to pack items only if their top edges are at the same height as or lower than the height of the bin ceiling. A small modification packs the item into a specific bin (items are no longer all packed into the same bin as in the strip packing problem) and the appropriate item is found by searching for the first item that fits onto the skyline segment, and is short enough to fit between the skyline segment and the top edge of the bin. If no item is found that is both narrow and short enough, the skyline segment is raised to the height of the lowest neighbour and a new search is performed in the hope that a wider (too wide to be packed onto the previous skyline segment), but short enough unpacked item exists. The fact that the BFmTN algorithm does not necessarily pack items in the same order in which they appear in the sorted list of items means that the items have to be re-sorted for every repacking. This is also necessary due to the bin width possibly changing, thereby changing the order in which the xWDWDH algorithm would sort items. Algorithms such as the BL, BLF and GCS algorithms pack items in the order in which they appear in the item list, which means that a list of items packed into a bin would remain in the order in which they were originally sorted, thereby eliminating the need for re-sorting during the repacking phase, except for the xWDWDH variations.

In an attempt to make the algorithm faster, the items are doubly linked when they are packed. The data structure describing the bin stores the index values of the first and last items to be packed into the bin. Therefore, when a temporary list of items is created to be repacked into the smaller bin, the algorithm may find all relevant items by means of a procedure that has a time complexity which is linear in terms of the number of items packed in the bin, and does not need to search all items for items that are packed in the current bin (which would have a time complexity $\mathcal{O}(n)$). This makes a significant difference, because the use of linked lists allows the items in all bins to be copied into individual lists for each bin in $\mathcal{O}(n)$ time, while the lack of linked-lists would result in a procedure of $\mathcal{O}(n\,p)$, where p denotes the number of bins that contain items.

7.3 Chapter Summary

In this chapter a brief overview of the literature on heuristic solutions to the SBSBPP was given in §7.1.1 in fulfilment of Dissertation Objective IX(a), followed by a review of the literature on heuristics for the MBSBPP in §7.1.2 in fulfilment of Dissertation Objective IX(b). A new algorithm for the 2D MBSBPP was presented in §7.2 in fulfilment of Dissertation Objective X, together with a worked example in §7.2.1 and an analysis of its worst-case time complexity in §7.2.2. Finally, the modifications required for the BFmTN algorithm in order for it to pack multiple bin sizes were presented in §7.2.3.

CHAPTER 8

An Appraisal of the Bin Packing Algorithms

Contents

In this chapter the results from a combination of the strip packing algorithms of §3–§5 and the best algorithm from §6.4 with the new 2SMBSBP heuristic will be presented. First, the known benchmarks that are used to compare the algorithms are described in §8.1.1, and this is followed by a discussion of the new benchmark instances that have been created for the MBSBPP in §8.1.2. The results of the various 2SMBSBP algorithms applied to these instances follow in §8.2–§8.4.

8.1 Benchmarks for the MBSBPP

In this section the benchmark instances used to compare the bin packing algorithms later in the chapter are described briefly. The data sets by Hopper and Turton [75, 79] are described first and this is followed by a description of the instances by Pisinger and Sigurd [137]. The benchmark instances by Berkey and Wang [16] and by Martello and Vigo [112] for the SBSBPP are added for interest sake. Finally, a new set of benchmark instances is generated for the MBSBPP.

8.1.1 Benchmark Instances from the Literature

Although there are many benchmarks available for strip packing or SBSBP problems, there are not many available for the MBSBPP. Some industry data are available (such as those supplied

by Wang [155, p. 585]), but the benchmark sets by Hopper and Turton [75, 79] and Pisinger and Sigurd [137] appear to be the only algorithm-generated benchmark instances available for this problem.

2000 Hopper and Turton

Hopper and Turton [75, 79] created three classes of benchmark instances (named M1, M2 and M3), each containing five problem instances, with one bin set per group. The sets M1 and M2 each contains 100 items and M3 contains 150 items, with six different sizes of bins for each group, a total of 16 bins in M1, 18 bins in M2 and 20 bins in M3. The item sizes were generated randomly, where the larger dimension ranges between 10% and 100% of the width of the bins. The set M1 was generated by means of an algorithm that placed a point randomly within a randomly-selected rectangle and split the rectangle into four parts by means of a horizontal and vertical cut through the point [75, p. 95–96]. The second set was generated by selecting an existing rectangle, randomly selecting an edge on the rectangle, randomly placing a point on that edge, mirroring that point on the opposite edge and cutting the rectangle along the straight line that joins the two points [75, pp. 96–97]. The final set, M3, was generated by means of an algorithm that randomly selects an existing rectangle and randomly places two points in the rectangle. These two points are then used to generate a small rectangle, the two points forming diagonally opposing corners. Further cuts are then made from the edges of this small rectangle to generate four further rectangles [75, pp. 98–99]. The items were finally scaled by a factor (> 1) in an attempt to render the total item area in each group similar. Optimal solutions to these benchmark instances are as yet unknown.

2005 Pisinger and Sigurd

Pisinger and Sigurd [137] based their benchmark instances on those by Berkey and Wang [16], and those by Martello and Vigo[1] [112]. These have been presented in some detail in §6.1. Pisinger and Sigurd follow the same process in generating their benchmark instances and each class is assigned a set of five different sizes of bins, the dimensions of which were selected uniformly from the ranges $[W/2, W] \times [H/2, H]$, where W and H are the width and height, respectively, of the original bins from the SBSBPP benchmark instances. Each bin is given a cost (which is ignored for the purposes of this dissertation) and 10 instances of 20, 40, 60, 80 and 100 items exist for each problem class, resulting in a total of 500 instances. The 500 benchmark instances by Berkey and Wang [16] and Martello and Vigo [112] are included in order to test the effectiveness of the algorithms for the special case of the MBSBPP when it reduces to the SBSBPP.

8.1.2 New Benchmark Instances for the MBSBPP

Additional benchmark instances were also generated. The benchmarks were generated in a manner similar to that of Wang and Valenzuela [156] with the constraint that the dimensions are integer values, and detailed pseudocode of the benchmark generation procedure may be found in Algorithm 8.1. The algorithm begins with a square rectangle of dimensions $1\,000 \times 1\,000$. An optimal solution is therefore known in each case. It can generate "pathological" benchmarks, or

[1] Pisinger and Sigurd incorrectly reference these benchmark sets as originating from the paper by Lodi *et al.* [105], which references the paper by Martello and Vigo [112] as the source of the benchmark sets.

"nice" benchmarks for which the area ratio constraint is $\gamma = 7$ and the aspect ratio constraint is $\rho = 4$, the same values used by Wang and Valenzuela for the two constraints. The item dimensions are restricted such that no item height is larger than the smallest bin height, and no item width is larger than the smallest bin width.

Algorithm 8.1 MBSBPP Benchmark Generator

Input: A vector $\underline{\mathtt{B}}$ containing the number of bins to be created, a vector $\underline{\mathtt{I}}$ containing the number of items to be created, the required number of sets of each $\underline{\mathtt{B}}/\underline{\mathtt{I}}$ pair $\mathtt{R}$, an area ratio constraint γ and an aspect ratio constraint parameter ρ.

Output: A set benchmarks containing dimensions of bins and items with an optimal packing area of 1 000 000 units.

for $i = 1$ **to** $|\underline{\mathtt{I}}|$ **do**
  for $j = 1$ **to** $|\underline{\mathtt{B}}|$ **do**
    $k \leftarrow 1$
    while $k \leq \mathtt{R}$ **do**
      $h(\mathcal{B}_1) \leftarrow 1\,000$, $w(\mathcal{B}_1) \leftarrow 1\,000$
      for $\ell = 2$ **to** $\underline{\mathtt{B}}(j)$ **do**
        determine the largest bin by area — $\mathcal{B}_L$
        if pathological bins are required **then**
          split $\mathcal{B}_L$ randomly through its largest dimension
        else if nice bins are required **then**
          find a random suitable bin $\mathcal{B}_s$ for which $A(\mathcal{B}_s) \geq 2A(\mathcal{B}_L)/\gamma$
          call SplitRec $(\mathcal{B}, s, \ell)$
        end if
      end for
      call CreateItems $(\mathcal{B}, \underline{\mathtt{I}}(i), \mathcal{I})$
      if the item set is valid **then**
        for $\ell = 1$ **to** $\underline{\mathtt{B}}(j)$ **do**
          assign random numbers of copies to $\underline{\mathtt{B}}(j)$
        end for
        $k \leftarrow k + 1$
      end if
    end while
  end for
end for

A rectangle is split by choosing a random integer point within the rectangle, and is then randomly cut horizontally or vertically through the chosen point. In an attempt to find valid benchmark instances faster, the cut direction was not always decided randomly. Instead, the dimension that exceeds the minimum bin dimension by the greatest margin was split in two. If a "nice" data set was required, the point was assigned randomly within limitations discussed in detail by Wang and Valenzuela [156].

Initially, cuts were made into the original rectangle to determine the bin sizes. These bins were subsequently used as initial items, which were further split until the required number of items resulted. A random number of copies of each bin size was created using a uniform distribution in the range $[2, 5]$. Once the required number of items had been cut, the validity of the result was determined. The item and bin sets were only valid if the total area of bins was greater than three times the area of the items, the longest item height was less than or equal to the shortest bin height and the longest item width was less than or equal to the shortest bin width.

Procedure 8.1.1 SPLITREC $(\mathcal{R}, \texttt{InRec}, \texttt{OutRec})$

1: {*When splitting a rectangle, save one resulting rectangle in* $\mathcal{R}_{\texttt{InRec}}$*, and save the other in* $\mathcal{R}_{\texttt{OutRec}}$*.*}
2: $h_i \leftarrow h(\mathcal{R}_{\texttt{InRec}})$, $w_i \leftarrow w(\mathcal{R}_{\texttt{InRec}})$, $m \leftarrow Area(\mathcal{R}_L)$
3: **if** $2h_i/\rho \leq w_i \leq \rho h_i/2$ **then**
4: **if** `RandomNumber` > 0.5 **then**
5: split vertically at random x, if possible, such that:
6: $\lceil \max(h_i/\rho, w_i - \rho h_i, m/\gamma h_i) \rceil \leq x \leq \lfloor \min(h_i\rho, w_i - h_i/\rho, w_i/2) \rfloor$
7: **else**
8: split horizontally at random y, if possible, such that:
9: $\lceil \max(w_i/\rho, h_i - \rho w_i, m/\gamma w_i) \rceil \leq y \leq \lfloor \min(w_i\rho, h_i - w_i/\rho, h_i/2) \rfloor$
10: **end if**
11: **else if** $2h_i/\rho \leq w_i \leq 2\rho h_i$ **then**
12: split vertically at random x, if possible, such that:
13: $\lceil \max(h_i/\rho, w_i - \rho h_i, m/\gamma h_i) \rceil \leq x \leq \lfloor \min(h_i\rho, w_i - h_i/\rho, w_i/2) \rfloor$
14: **else if** $2w_i/\rho \leq h_i \leq 2\rho w_i$ **then**
15: split horizontally at random y, if possible, such that:
16: $\lceil \max(w_i/\rho, h_i - \rho w_i, m/\gamma w_i) \rceil \leq y \leq \lfloor \min(w_i\rho, h_i - w_i/\rho, h_i/2) \rfloor$
17: **end if**

Five copies of each of a combination of 2, 3, 4, 5 and 6 bins, and 25, 50, 100, 200, 300, 400 and 500 items were created (not for the 25 item and 6 bin combination), resulting in a total of 340 benchmark instances for the MBSBPP. These benchmarks may be found online [154]. This results in a total of 1 357 benchmark instances.

8.2 Results of Level-Packing MBSBP Heuristics

In this section the best level-packing algorithms in each set identified in §6.2, *i.e.* those listed in §6.2.5, are be compared. The BFDHDW algorithm for the MBSBPP applied to the 857 benchmark instances yields the best mean rank of the NF, FF,BF and WF algorithms. Of the KP_{TR} algorithms, the KP_{TR}DHDW algorithm yields the best mean rank for bin utilisation (it was not significantly different from the KP_{TR}DHIW algorithm for the strip packing problem, see Table 6.3) and is used for comparison purposes in this section. It was found that the JOIN algorithms that joined items vertically would yield super-items that are taller than some of the instances' bins, yielding infeasible packings. Combined with these algorithms' poor packing densities for the strip packing problem (see §6.2.3), it was decided to remove them from consideration for the MBSBPP. It was also found that changes to δ, where $\delta \in \{0, 5, 10, 15\}$ did not affect the result. All JOIN_δDH algorithms yield an overall mean rank of 40.19, while all JOIN_δDHDW algorithms yielded a mean rank of 39.74 and the JOIN_δDHIW algorithms yield a mean rank of 41.60. Of the B2FA algorithms, the B2FA_{10}DHDW yields the best mean rank and of the B2FW algorithms, the B2FW_2DHDW algorithm yields the best mean rank. A Friedman test [40] performed in MATLAB [113] on the utilisation and fitness scores for the 857 MBSBPP benchmark instances, and the number of bins (for the 500 SBSBPP benchmark instances) each yielded $P = 0$, suggesting that the null hypothesis that all algorithms are equivalent may be rejected. Table 8.1 contains an overview of the results and Figure 8.1 contains box plots of the utilisation results. Table 8.2 contains results for the various classes of benchmark instances for the MBSBPP. The Nemenyi critical distance for 5 items and 857 benchmark instances

Procedure 8.1.2 CREATEITEMS $(\mathcal{B}, r, \mathcal{I})$

```
find MinH and MinW, the minimum height and width of the existing bins, respectively
set the first |B| items' dimensions equal to those of the bins'
i ← |B|
while i < r do
  determine the largest item by area — I_L
  i ← i + 1
  if pathological items are required then
    {When splitting a rectangle, save one rectangle in I_L, and the other in I_i.}
    if h(I_L) > MinH and w(I_L) > MinW then
      if h(I_L) − MinH > w(I_L) − MinW then
        split horizontally randomly
      else if h(I_L) − MinH < w(I_L) − MinW then
        split vertically randomly
      else
        randomly split the item randomly horizontally or randomly vertically
      end if
    else if h(I_L) > MinH then
      split horizontally randomly
    else if w(I_L) > MinW then
      split vertically randomly
    else
      randomly split the item randomly horizontally or randomly vertically
    end if
  else if nice items are required then
    find a random suitable item I_s for which A(I_s) ≥ 2A(I_L)/γ
    call SPLITREC(I, s, i)
  end if
end while
```

at a confidence level of 95% with respect to the utilisation μ and the fitness ν is 0.27. The critical distance between the mean ranks for significance with respect to 5 algorithms and 500 benchmark instances is 0.21.

The box plots in Figure 8.1 show that the distribution of utilisations achieved by the JOIN_0-DHDW algorithm combined with the 2SMBSBP heuristic is lower than the utilisations for the four other algorithms. This is reflected in the summary in Table 8.1, where the upper and lower quartiles, the median and maximum utilisations are lower than the four remaining algorithms. The total number of bins used is larger than for the remaining algorithms and the number of bins that were not repacked is largest. This results in its mean rank being the worst of all algorithms and the Nemenyi test suggests that it is significantly worse than the remaining algorithms. This result is expected when considering the results that the JOIN algorithm yielded for the strip packing problem in §6.2.5.

The distributions of the four other algorithms are too similar to compare by means of the box plots in Figure 8.1 or the quartile values in Table 8.1. Instead, the mean ranks aid in the distinction of one of the algorithms from the others, because the mean rank of 2.99 for the B2FW_2DHDW algorithm suggests it is significantly different from the three other algorithms according to the Nemenyi test. It is also the algorithm that packed items into the second

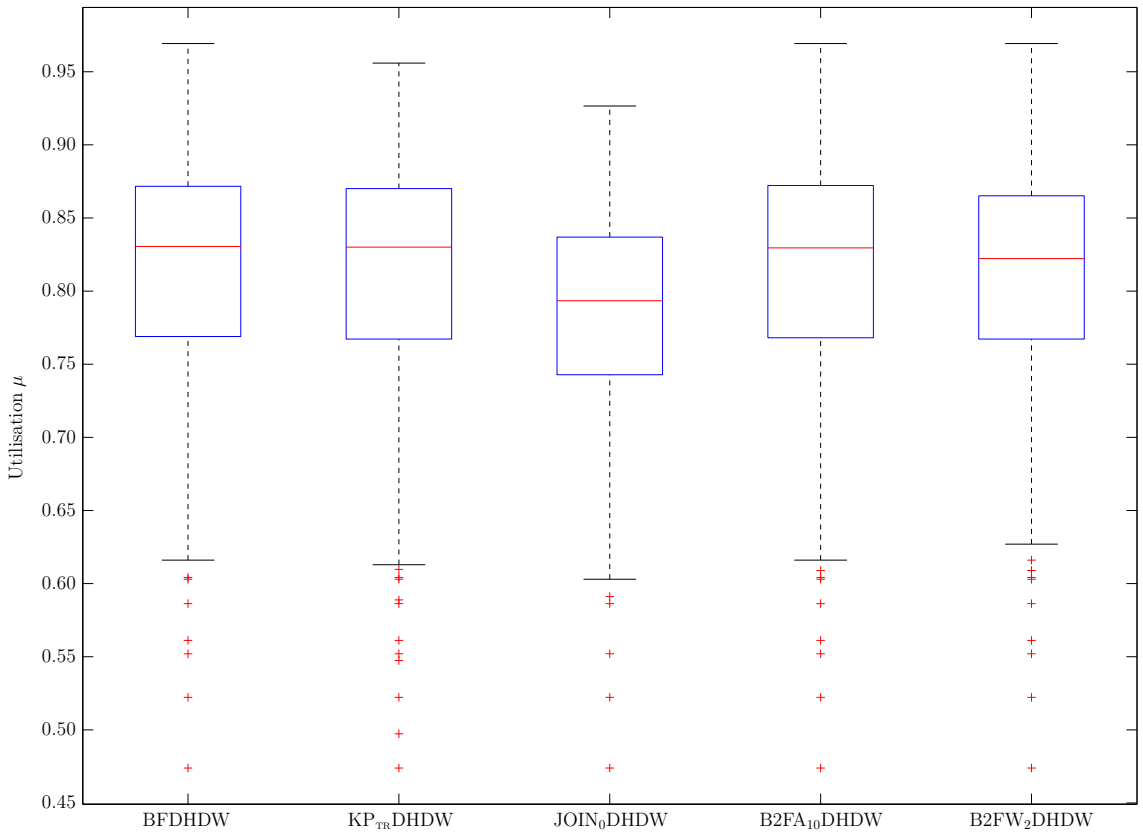


Figure 8.1: *Box plot of the results for the best of the level-packing heuristics from each set in §6.2 for the MBSBPP for all MBSBPP benchmark instances described in §8.1.*

largest number of bins, after the JOIN$_0$DHDW algorithm. However, the Nemenyi test was unable to distinguish between the remaining three algorithms. The B2FA$_{10}$DHDW algorithm yields the highest mean rank, a difference of only 0.04 to the BFDHDW algorithm which yields the best mean rank (the critical distance is 0.21 for five algorithms and 857 instances). The BFDHDW algorithm was able to utilise the smallest number of bins in total (2 fewer than the KP$_{TR}$DHDW algorithm, 28 fewer than the B2FA$_{10}$DHDW algorithm) and repack 14 more bins than the KP$_{TR}$DHDW algorithm, while the B2FA$_{10}$DHDW algorithm was able to repack 68 more, thereby achieving the smallest number of bins that were not repacked.

In an attempt to further differentiate between the algorithms, the fitness ν as proposed by Hopper [75] may be used as a measure of packing density (see §2.3.2). The mean ranks yield the same order as for μ, but subjecting the algorithms' fitness results to the Nemenyi test yields classes suggesting that the BFDHDW algorithm is significantly better than the other algorithms. However, the test is not powerful enough to distinguish between the KP$_{TR}$DHDW and B2FA$_{10}$DHDW algorithms. However, the weakness of the KP$_{TR}$DHDW algorithm is the time required to find solutions to large problems, requiring two orders of magnitude more time to find a solution to the 2 900 item problem by Wang [155]. Therefore, the results suggest that the BFDHDW algorithm is the best level-packing algorithm to use in conjunction with the 2SMBSBP algorithm in order to solve the MBSBPP.

The mean ranks in Table 8.2 confirm the JOIN$_0$DHDW algorithm's poor performance — the algorithm consistently yields the worst rank. For 5 algorithms and 170 benchmark instances the Nemenyi CD increases to 0.47. This means that the Nemenyi test does not find a significant

	BFDHDW	KP_{TR}DHDW	$JOIN_0$DHDW	$B2FA_{10}$DHDW	$B2FW_2$DHDW
Min. μ	47.4%	47.4%	47.4%	47.4%	47.4%
Low. Q. μ	76.9%	76.7%	74.3%	76.8%	76.7%
Med. μ	83.1%	83.0%	79.3%	83.0%	82.2%
Up. Q. μ	87.2%	87.0%	83.7%	87.2%	86.5%
Max. μ	96.9%	95.6%	92.7%	96.9%	96.9%
IQR	10.3%	10.3%	9.4%	10.4%	9.8%
Wang P1 t (s)	5.9768	916.58	6.1197	7.0358	5.8858
Total Bins	9306	9308	9789	9334	9397
Repacked Bins	1503	1491	1772	1559	1581
% Repacked	16.2%	16.0%	18.1%	16.7%	16.8%
Stationary Bins	7 803	7 817	8 017	7 775	7 816
Mean μ Rank	2.68 (1)	2.72 (2)	3.86 (5)	2.74 (3)	2.99 (4)
Nem. μ Class	C	C	A	C	B
Mean ν Rank	2.46 (1)	2.69 (2)	4.11 (5)	2.74 (3)	3.00 (4)
Nem. ν Class	D	C	A	C	B

Table 8.1: *Overview of the level-packing algorithmic results for the MBSBPP. The row labelled* 'Min. μ' *contains the minimum bin utilisation over the 857 MBSBP benchmark instances, while the rows labelled* 'Low. Q. μ', 'Med. μ', 'Up. Q. μ', 'Max. μ' *and* 'IQR' *contain the lower quartile, median, upper quartile, maximum and interquartile range of the results for the instances, respectively. The row labelled* 'Wang P1 t (s)' *contains the time taken (in seconds) for the algorithms to complete the packing of the first problem by Wang [155], the largest benchmark instance. The row labelled* 'Total Bins' *is the total number bins used over all MBSBP benchmark instances, the row labelled* 'Repacked Bins' *documents the total number of bins that were repacked during the repacking phase of the 2SMBSBP algorithm, the row labelled* '% Repacked' *indicates what percentage of the total number of bins this repack value is, and the row labelled* 'Stationary Bins' *lists the number of bins that were not repacked. The row labelled* 'Mean μ Rank' *documents the mean ranks of the algorithms when applied to the utilisation (the ranks are given in parentheses), while the row labelled* 'Nem. μ Class' *shows which algorithms are not significantly different according to the Nemenyi test [40] by assigning them the same letter. The same tests are performed for the fitness ν in the two rows that follow.*

	BFDHDW	KP_{TR}DHDW	$JOIN_0$DHDW	$B2FA_{10}$DHDW	$B2FW_2$DHDW
Wang P1 μ	84.4%	84.4%	73.4%	85.3%	85.3%
Wang P2 μ	88.2%	88.2%	79.9%	88.2%	84.1%
Hopper M1	2.20	3.00	5.00	2.20	2.60
Hopper M2	2.20	3.00	5.00	2.20	2.60
Hopper M3	2.20	2.00	4.90	2.50	3.40
PS 1	2.25	2.78	4.64	2.60	2.73
PS 2	2.87	2.77	3.72	2.82	2.82
PS 3	2.26	2.63	4.47	2.44	3.20
PS 4	2.78	2.86	3.59	2.78	2.99
PS 5	2.28	2.27	4.31	2.83	3.31
PS 6	2.82	2.82	3.58	2.82	2.96
PS 7	2.76	2.63	3.42	3.08	3.11
PS 8	2.37	2.32	4.74	2.57	3.00
PS 9	3.12	2.36	3.75	2.77	3.00
PS 10	2.41	2.31	4.20	2.62	3.46
Nice	2.95	2.76	3.61	2.72	2.95
Path	2.74	3.13	3.47	2.81	2.86

Table 8.2: *Level-packing algorithmic results for the MBSBPP for various sets of benchmark instances. The utilisation achieved for the two problems by Wang are followed by the mean ranks of the algorithms for the Hopper and Turton (labelled* M*), Pisinger and Sigurd (labelled* PS*) and new benchmark instances (split into* Nice *and* Path *instances).*

difference between the best four algorithms for the nice and path data sets, but the mean ranks suggest that the KP_{TR}DHDW and $B2FA_{10}$DHDW algorithms may be better than the BFDHDW algorithm for nice data. However, the positions may be reversed for pathological input, particularly between the BFDHDW and KP_{TR}DHDW algorithms.

	BFDHDW	KP_{TR}DHDW	$JOIN_0$DHDW	$B2FA_{10}$DHDW	$B2FW_2$DHDW
Mean p Rank	2.67 (2)	2.65 (1)	3.81 (5)	2.84 (3)	3.04 (4)
Nem. p Class	C	C	A	BC	B
100 t (ms)	1.8920	108.73	1.8169	1.9134	1.7623
BW 1	20.62	20.64	22.22	20.66	20.72
BW 2	2.64	2.62	2.72	2.64	2.64
BW 3	14.72	14.84	16.14	14.88	15.14
BW 4	2.60	2.64	2.70	2.60	2.64
BW 5	18.70	18.70	19.74	18.92	19.06
BW 6	2.36	2.38	2.40	2.36	2.36
MV 7	17.18	17.10	17.36	17.26	17.28
MV 8	17.52	17.32	19.14	17.54	17.72
MV 9	42.78	42.76	43.16	42.92	42.98
MV 10	10.74	10.72	11.42	10.82	11.00
Total Bins	7 493	7 486	7 850	7 530	7 577

Table 8.3: *Level-packing algorithmic results for the SBSBPP for various sets of benchmark instances. The row labelled* 'Mean p Rank' *shows the mean rank over the 500 benchmark instances in terms of the number of bins packed, while the row* 'Nem. p Class' *shows which algorithms are not significantly different by placing them in the same class, indicated by a letter. Finally, the row labelled* '100 t (ms)' *shows the mean time (in milliseconds) that the algorithms required to solve the SBSBP benchmark instances with 100 items. The results below these rows are the mean numbers of bins for each problem class.*

The same algorithms, combined with the 2SMBSBP algorithm, were applied to a 500 instance benchmark set of a special case of the MBSBPP, namely the SBSBPP, and the results are shown in Table 8.3. The $JOIN_0$DHDW algorithm was the worst again, and the $B2FW_2$DHDW algorithm yielded the second worst set of results for this problem. The $B2FA_{10}$DHDW algorithm yields the third best mean rank, but the Nemenyi test was not powerful enough to distinguish between it and the other algorithms, excluding the $JOIN_0$DHDW algorithm. However, the Nemenyi test suggests that the BFDHDW and KP_{TR}DHDW algorithms are significantly better than those with the worst two ranks. The KP_{TR}DHDW algorithm may yield the best mean rank (by 0.02) in this set and the lowest number of bins in total, but it is slow, requiring a mean time of 108.7 milliseconds to find a solution to the instances of 100 items, while the BFDHDW algorithm required a mean of 1.892 milliseconds to solve the same problems.

8.3 Results of Pseudolevel-Packing MBSBP Heuristics

This section consists of two subsections: one in which the results for the pseudolevel algorithms that yield guillotine results are reported and one in which those algorithms that cannot guarantee a guillotine layout are considered. In §8.3.1 the best of the FC_{OG} algorithms (the FC_{OG}DHDW algorithm) are compared with the best BFDH* algorithm (the BFDH*(DW) algorithm), the best of the SAS algorithms (the new, modified SASm algorithm), the new BFS algorithm and the best of the SL algorithms for the strip packing problem, SL_5 (it is interesting to note that the SL_δ, $\delta \in \{0, 5, 10, 15\}$ algorithms yield the same results for all 1 357 benchmark instances). The results obtained by these algorithms, when applied to the strip packing

problem, may be found in §6.3.1. In §8.3.2 certain non-guillotine pseudolevel algorithms are compared, including the FC_{OF}DHDW algorithm (found to be the best of the FC_{OF} algorithms in 6.4.1), the SC algorithm and the SCR algorithm.

8.3.1 Results for the Guillotine Heuristics

This subsection is dedicated to the results of the pseudolevel-packing algorithms that are guaranteed to yield guillotine layouts. A box plot of the distribution of results for the utilisation measure may be found in Figure 8.2, while Table 8.4 contains a summary of the various algorithmic results for the MBSBPP. Further details may be found in Table 8.5, in which mean results for the various sets of benchmark instances are listed. A Friedman test performed on the utilisation and fitness scores yields $P = 0$, suggesting that the null hypothesis that all algorithms are equivalent may be rejected. The Nemenyi CD for five algorithms and 857 benchmark instances at a 95% confidence level is 0.21.

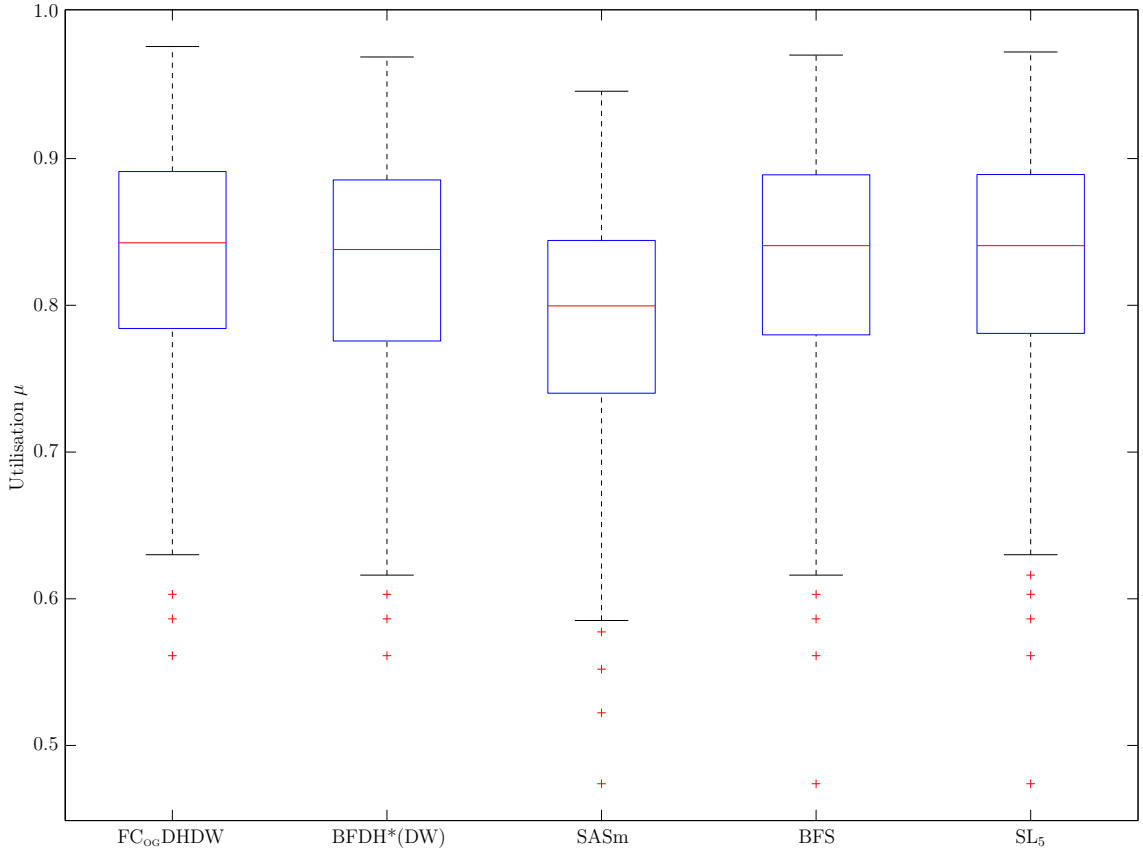


Figure 8.2: *Box plot of the results for the best of the guillotine pseudolevel-packing heuristics in §6.3.1 for the MBSBPP for all MBSBPP benchmark instances described in §8.1.*

The box plots in Figure 8.2 show how the application of the SASm algorithm, in combination with the 2SMBSBP algorithm, yields a distribution of results that are shifted lower than those of the other algorithms, suggesting that it would not be the best choice of algorithm for finding a dense packing. There is further evidence of this in Table 8.4 and Table 8.5. The SASm algorithm yields lower quartile, median and upper quartile values that are approximately 4% worse than the best values. The best utilisation it achieves is 2% lower than that of the next best algorithm. Its poor performance is reflected in the number of bins required to pack all

	$FC_{OG}DHDW$	BFDH*(DW)	SASm	BFS	SL_5
Min. μ	56.1%	56.1%	47.4%	47.4%	47.4%
Low. Q. μ	78.4%	77.6%	74.0%	78.0%	78.1%
Med. μ	84.3%	83.8%	80.0%	84.1%	84.1%
Up. Q. μ	89.1%	88.5%	84.4%	88.9%	88.9%
Max. μ	97.6%	96.9%	94.6%	97.1%	97.3%
IQR	10.7%	11.0%	10.4%	10.9%	10.8%
Wang P1 t (s)	16.082	6.6837	5.4528	5.9540	5.9386
Total Bins	9 248	9 268	9 642	9 254	9 259
Repacked Bins	1 538	1 536	1 840	1 549	1 561
% Repacked	16.6%	16.6%	19.1%	16.7%	16.9%
Stationary Bins	7 710	7 732	7 802	7 705	7 698
Mean μ Rank	2.70 (1)	2.86 (4)	3.98 (5)	2.74 (3)	2.71 (2)
Nem. μ Class	B	B	A	B	B
Mean ν Rank	2.56 (1)	2.93 (4)	4.19 (5)	2.67 (3)	2.65 (2)
Nem. ν Class	C	B	A	C	C

Table 8.4: *Overview of the guillotine pseudolevel-packing algorithmic results for the MBSBPP. The row labelled* 'Min. μ' *contains the minimum bin utilisation over the 857 MBSBP benchmark instances, while the rows labelled* 'Low. Q. μ', 'Med. μ', 'Up. Q. μ', 'Max. μ' *and* 'IQR' *contain the lower quartile, median, upper quartile, maximum and interquartile range of the results for the instances, respectively. The row labelled* 'Wang P1 t (s)' *contains the time taken (in seconds) for the algorithms to complete the packing of the first problem by Wang [155], the largest benchmark instance. The row labelled* 'Total Bins' *is the total number bins used over all MBSBP benchmark instances, the row labelled* 'Repacked Bins' *documents the total number of bins that were repacked during the repacking phase of the 2SMBSBP algorithm, the row labelled* '% Repacked' *indicates what percentage of the total number of bins this repack value is, and the row labelled* 'Stationary Bins' *lists the number of bins that were not repacked. The row labelled* 'Mean μ Rank' *documents the mean ranks of the algorithms when applied to the utilisation (the ranks are given in parentheses), while the row labelled* 'Nem. μ Class' *shows which algorithms are not significantly different according to the Nemenyi test [40] by assigning them the same letter. The same tests are performed for the fitness ν in the two rows that follow.*

	$FC_{OG}DHDW$	BFDH*(DW)	SASm	BFS	SL_5
Wang P1 μ	84.4%	84.4%	77.7%	84.4%	84.4%
Wang P2 μ	90.5%	90.5%	90.5%	90.5%	90.5%
Hopper M1	2.60	2.60	4.60	2.60	2.60
Hopper M2	2.50	2.90	4.20	2.90	2.50
Hopper M3	2.70	2.70	5.00	2.30	2.30
PS 1	2.74	2.70	4.14	2.74	2.68
PS 2	2.85	2.85	3.60	2.85	2.85
PS 3	2.54	2.53	4.81	2.50	2.62
PS 4	2.71	2.81	3.86	2.81	2.81
PS 5	2.58	2.88	4.24	2.59	2.71
PS 6	2.76	2.81	3.75	2.86	2.82
PS 7	2.84	3.03	3.18	2.98	2.97
PS 8	2.81	2.80	4.24	2.64	2.51
PS 9	3.43	2.87	3.10	2.80	2.80
PS 10	2.38	2.89	4.68	2.47	2.58
Nice	2.70	2.84	3.83	2.88	2.75
Path	2.53	3.04	4.13	2.66	2.64

Table 8.5: *Guillotine pseudolevel-packing algorithmic results for the MBSBPP for various sets of benchmark instances. The utilisation achieved for the two problems by Wang are followed by the mean ranks of the algorithms for the Hopper and Turton (labelled* M*), Pisinger and Sigurd (labelled* PS*) and new benchmark instances (split into* Nice *and* Path *instances).*

items in all the benchmark instances. It requires 9 642 bins compared to the 9 268 bins required by the next best algorithm, the BFDH*(DW) algorithm, which requires only 20 bins more than the FC_{OG}DHDW algorithm, the best of the set in this regard. However, the SASm algorithm does yield the fastest solution for the largest problem instance.

It is difficult to distinguish between the remaining four algorithms in this set; the distribution details in Table 8.4 are useless in this regard. The difference between mean ranks for the four algorithms is approximately 0.16, a value lower than the CD that would suggest a significant difference between the algorithms. Using the mean ranks for the fitness test allows for differentiation of the BFDH*(DW) algorithm from the four other algorithms, a result which would have been expected from the results of the strip packing problem (see §6.3.1). This is evident for pathological data sets, for which the BFDH*(DW) algorithm is significantly worse than the FC_{OG}DHDW algorithm (the CD is 0.47 and the mean rank gap between the two is 0.50). For nice data sets they are not significantly different according to the Nemenyi test. The packing performance of the FC_{OG}DHDW, BFS and SL_5 algorithms are not significantly different according to the Nemenyi test, but the FC_{OG}DHDW algorithm does require more than double the time required by the BFS and SL_5 algorithms to find solutions to the largest problem instance, suggesting that one of the two new algorithms may be the best algorithm in this set for the MBSBPP.

	FC_{OG}DHDW	BFDH*(DW)	SASm	BFS	SL_5
Mean p Rank	2.79 (1)	2.82 (3)	3.78 (5)	2.80 (2)	2.82 (4)
Nem. p Class	B	B	A	B	B
100 t (ms)	4.2915	2.5469	1.6787	2.0792	2.0243
BW 1	20.60	20.60	21.24	20.60	20.62
BW 2	2.60	2.60	2.70	2.60	2.60
BW 3	14.72	14.72	16.26	14.72	14.80
BW 4	2.56	2.56	2.70	2.58	2.56
BW 5	18.70	18.70	19.66	18.70	18.70
BW 6	2.36	2.36	2.44	2.36	2.36
MV 7	17.10	17.14	17.22	17.12	17.12
MV 8	17.50	17.52	18.08	17.50	17.54
MV 9	42.78	42.78	43.00	42.78	42.78
MV 10	10.52	10.60	11.42	10.52	10.54
Total Bins	7 472	7 479	7 736	7 474	7 481

Table 8.6: *Guillotine pseudolevel-packing algorithmic results for the SBSBPP for various sets of benchmark instances. The row labelled* 'Mean p Rank' *shows the mean rank over the 500 benchmark instances in terms of the number of bins packed, while the row* 'Nem. p Class' *shows which algorithms are not significantly different by placing them in the same class, indicated by a letter. Finally, the row labelled* '100 t (ms)' *shows the mean time (in milliseconds) that the algorithms required to solve the SBSBP benchmark instances with 100 items. The results below these rows are the mean numbers of bins for each problem class.*

The same algorithms, combined with the 2SMBSBP algorithm, were applied to the 500 benchmark instances by Berkey and Wang [16] and Martello and Vigo [112], and the results are listed in Table 8.9. The SASm algorithm was once again the worst of the set, consistently requiring the largest number of bins to accommodate all items. However, the four other algorithms are very similar, yielding a range of mean ranks of 0.04 for a CD of 0.27, as was the case for the MBSBPP. The FC_{OG}DHDW algorithm requires approximately double the time required by the BFS and SL_5 algorithms to find a similar packing for the benchmark instances where 100 items were packed into bins.

8.3.2 Results for the Free-Packing Heuristics

This subsection is dedicated to the results of the pseudolevel-packing algorithms that are not guaranteed to yield guillotine layouts. A box plot of the distribution of results for the utilisation measure may be found in Figure 8.2, while Table 8.4 contains a summary of the algorithmic results for the MBSBPP. Further details may be found in Table 8.5, in which mean results for the various sets of benchmark instances are shown. A Friedman test performed on the utilisation yields $P = 0.0044$, while the same test applied to the fitness scores yields $P = 0.0011$, suggesting that the null hypothesis that all algorithms are equivalent may be rejected. The Nemenyi CD for three algorithms and 857 benchmark instances at a 95% confidence level is 0.11.

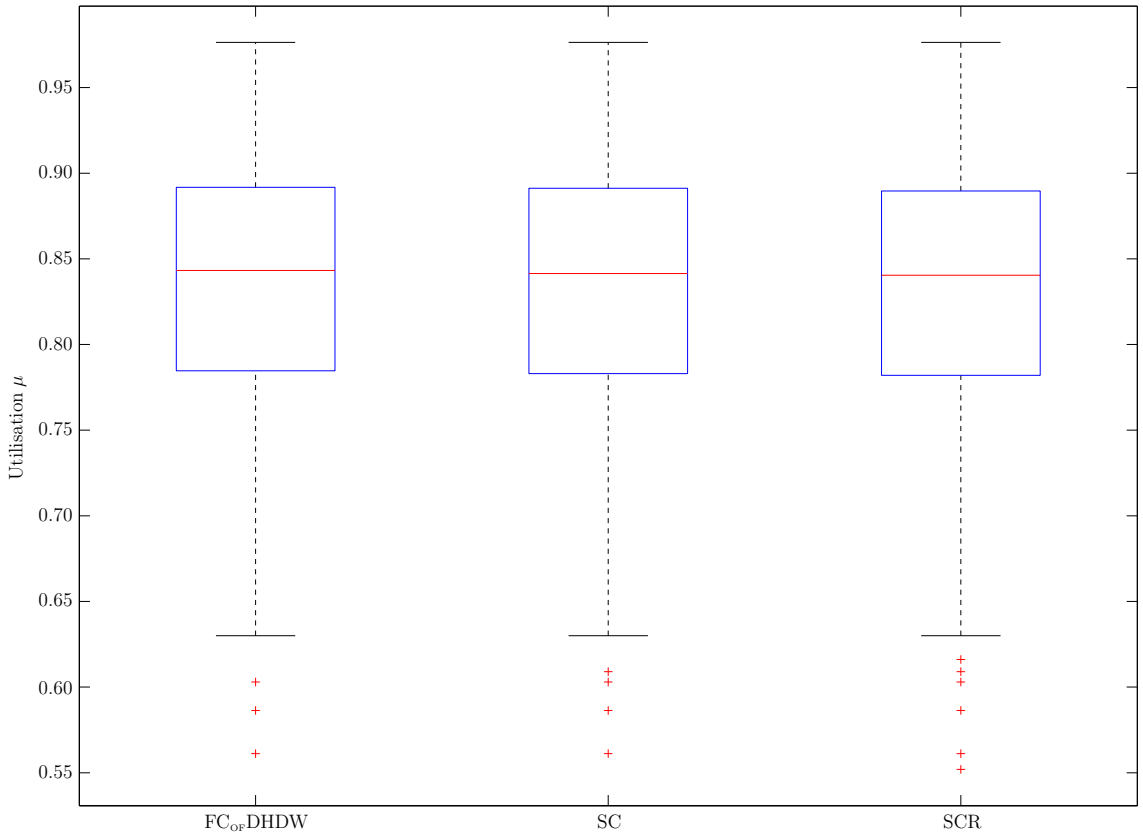


Figure 8.3: *Box plot of the results for the best of the non-guillotine pseudolevel-packing heuristics in §6.3.2 for the MBSBPP for all MBSBPP benchmark instances described in §8.1.*

The box plots in Figure 8.3 do not suggest a large difference between the algorithms, and the distribution figures in Table 8.7 confirm this, with the difference between the best and worst algorithms being approximately 0.3% (absolute) for the lower quartile, the median and the upper quartile values. The FC_{OF}DHDW algorithm utilises fewer bins in total than the SC and SCR algorithms, but the SC and SCR algorithms are able to repack a larger number of bins than the FC_{OF}DHDW algorithm, yielding smaller numbers of bins that were not repacked for the SC and SCR algorithms than for the FC_{OF}DHDW algorithm. The Nemenyi test (at a 95% confidence level) is not powerful enough to distinguish between the three algorithms based on utilisations, but finds a significant difference between the FC_{OF}DHDW and SCR algorithms when the fitness measure is used to compare the algorithms. The better packing performance (though not significantly different from the SC algorithm) of the FC_{OF}DHDW algorithm is balanced by the time it requires to solve the largest benchmark instance — it requires more

	FC_{OF}DHDW	SC	SCR
Min. μ	56.1%	56.1%	55.2%
Low. Q. μ	78.5%	78.3%	78.2%
Med. μ	84.3%	84.1%	84.0%
Up. Q. μ	89.2%	89.1%	88.9%
Max. μ	97.6%	97.6%	97.6%
IQR	10.7%	10.8%	10.7%
Wang P1 t (s)	16.328	5.8691	6.5102
Total Bins	9 246	9 280	9 286
Repacked Bins	1 535	1 610	1 610
% Repacked	16.6%	17.3%	17.3%
Stationary Bins	7 711	7 670	7 676
Mean μ Rank	1.96 (1)	2.01 (2)	2.03 (3)
Nem. μ Class	A	A	A
Mean ν Rank	1.92 (1)	2.00 (2)	2.08 (3)
Nem. ν Class	B	AB	A

Table 8.7: *Overview of the non-guillotine pseudolevel-packing algorithmic results for the MBSBPP. The row labelled* 'Min. μ' *contains the minimum bin utilisation over the 857 MBSBP benchmark instances, while the rows labelled* 'Low. Q. μ', 'Med. μ', 'Up. Q. μ', 'Max. μ' *and* 'IQR' *contain the lower quartile, median, upper quartile, maximum and interquartile range of the results for the instances, respectively. The row labelled* 'Wang P1 t (s)' *contains the time taken (in seconds) for the algorithms to complete the packing of the first problem by Wang [155], the largest benchmark instance. The row labelled* 'Total Bins' *is the total number bins used over all MBSBP benchmark instances, the row labelled* 'Repacked Bins' *documents the total number of bins that were repacked during the repacking phase of the 2SMBSBP algorithm, the row labelled* '% Repacked' *indicates what percentage of the total number of bins this repack value is, and the row labelled* 'Stationary Bins' *lists the number of bins that were not repacked. The row labelled* 'Mean μ Rank' *documents the mean ranks of the algorithms when applied to the utilisation (the ranks are given in parentheses), while the row labelled* 'Nem. μ Class' *shows which algorithms are not significantly different according to the Nemenyi test [40] by assigning them the same letter. The same tests are performed for the fitness ν in the two rows that follow.*

	FC_{OF}DHDW	SC	SCR
Wang P1 μ	84.4%	84.4%	84.4%
Wang P2 μ	90.5%	90.5%	90.5%
Hopper M1	2.00	2.00	2.00
Hopper M2	2.17	1.92	1.92
Hopper M3	1.69	2.15	2.15
PS 1	1.82	2.06	2.12
PS 2	2.02	1.99	1.99
PS 3	1.87	2.08	2.05
PS 4	1.96	1.96	2.08
PS 5	1.81	2.08	2.11
PS 6	2.00	2.00	2.00
PS 7	1.87	2.08	2.05
PS 8	2.02	1.99	1.99
PS 9	2.33	1.85	1.82
PS 10	1.91	1.99	2.10
Nice	1.97	1.99	2.04
Path	1.96	2.01	2.03

Table 8.8: *Non-guillotine pseudolevel-packing algorithmic results for the MBSBPP for various sets of benchmark instances. The utilisation achieved for the two problems by Wang are followed by the mean ranks of the algorithms for the Hopper and Turton (labelled* M*), Pisinger and Sigurd (labelled* PS*) and new benchmark instances (split into* Nice *and* Path *instances).*

than double the time of the SC algorithm to find a solution. The results in Table 8.8 show that there does not appear to be a large difference between the three algorithms for nice or pathological data.

	FC_{OF}DHDW	SC	SCR
Mean p Rank	1.93 (1)	2.03 (2)	2.04 (3)
Nem. p Class	A	A	A
100 t (ms)	3.9288	2.1285	2.5725
BW 1	20.60	20.62	20.64
BW 2	2.60	2.60	2.60
BW 3	14.72	14.86	14.86
BW 4	2.54	2.54	2.56
BW 5	18.70	18.88	18.88
BW 6	2.36	2.36	2.36
MV 7	17.10	17.18	17.18
MV 8	17.50	17.52	17.52
MV 9	42.78	42.90	42.90
MV 10	10.52	10.58	10.62
Total Bins	7 471	7 502	7 506

Table 8.9: *Non-guillotine pseudolevel-packing algorithmic results for the SBSBPP for various sets of benchmark instances. The row labelled* 'Mean p Rank' *shows the mean rank over the 500 benchmark instances in terms of the number of bins packed, while the row* 'Nem. p Class' *shows which algorithms are not significantly different by placing them in the same class, indicated by a letter. Finally, the row labelled* '100 t (ms)' *shows the mean time (in milliseconds) that the algorithms required to solve the SBSBP benchmark instances with 100 items. The results below these rows are the mean numbers of bins for each problem class.*

A Friedman test, at a 95% confidence level, on the number of bins packed for the SBSBPP yields $P = 1.74 \times 10^{-10}$, suggesting that the null hypothesis that all algorithms are equivalent may be rejected. However, the Nemenyi test is again not powerful enough (at a confidence level of 95%) to distinguish between the algorithms in this case. The FC_{OF}DHDW algorithm does appear to yield either an equal mean number of bins per class, or less than that achieved by the SC or SCR algorithms. Comparing the total number of bins utilised suggests that the FC_{OF}DHDW algorithm is the best of this set, but the time is takes to solve problems is its weakness, requiring almost double the time required by the SC algorithm.

8.4 Results of the BFmTN Heuristic for the MBSBPP

This section is dedicated to the presentation of the three variations on the modified version of the BFTN algorithm when combined with the 2SMBSBP algorithm. A box plot of the results may be found in Figure 8.4 and an overview of the algorithmic results may be found in Table 8.10. Further detail of the algorithms' performances for the various benchmark sets may be found in Table 8.11. A Friedman test applied to the utilisation and fitness results for the 857 MBSBPP benchmark instances yielded $P = 0$ for both cases, suggesting that the null hypothesis that all algorithms are equivalent may be rejected.

The box plot for the BFmTN algorithm with items sorted according to decreasing area appears to indicate a distribution that is shifted further toward full utilisation than do the sorting methods that sort according to a user-defined fraction. These observations are shown to be accurate by the summary of the results in Table 8.10, with the utilisation values consistently decreasing from left to right. The mean rank of the BFmTN(DA) algorithm is the lowest for

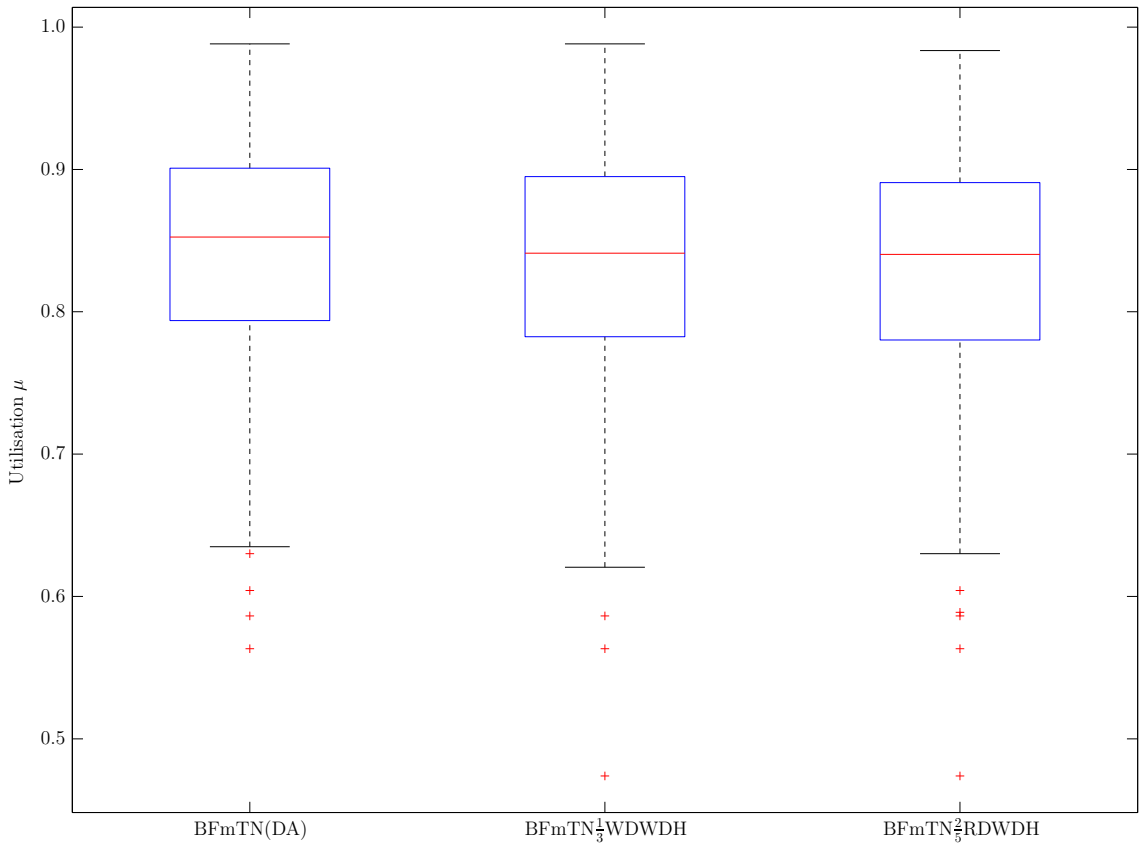


Figure 8.4: *Box plot of the results for the best of the plane-packing heuristics in §6.4.7 for the MBSBPP for all MBSBPP benchmark instances described in §8.1.*

both utilisation and fitness scores and the algorithm is significantly better than the remaining algorithms according to the Nemenyi test which yields a CD of 0.11 for three algorithms and 857 benchmark instances. The better performance of the BFmTN(DA) algorithm is evident in the total number of bins packed. It only requires 9 158 bins in total to pack all the items in the benchmark instances for the MBSBPP, while the BFmTN$\frac{1}{3}$WDWDH algorithm requires 123 more and the BFmTN$\frac{2}{5}$RDWDH algorithm requires 150 more bins to pack all of the items. However, the BFmTN$\frac{2}{5}$RDWDH algorithm is able to repack more bins than the other two algorithms, resulting in it having the lowest number of stationary bins. A significant difference between the two algorithms is the time required to solve the largest of the problem instances. The DA variation requires almost 5 seconds to solve the problem, while the $\frac{1}{3}$WDWDH variation requires over 140 seconds and the $\frac{2}{5}$RDWDH algorithm requires approximately 130 seconds. This is likely to be due to the many re-sorting steps for each bin packing as either the bin width (and therefore the items that should by sorted by width and not height), or the fraction of items sorted by width, may change often. However, the two slower algorithms yield better utilisations for the problem (see Table 8.11).

Table 8.11 shows the mean ranks for the various sets of benchmark instances. The results suggest that the M1 and M3 sets of instances are best solved by means of the BFmTN$\frac{1}{3}$WDWDH algorithm, which achieves mean utilisations of 96.3% and 94.7% for the two problem sets, respectively, compared with the 94.9% and 93.3% by the BFmTN(DA) algorithm and the 95.5% and 93.9% by the BFmTN$\frac{2}{5}$RDWDH algorithm. The results for the M2 set is closer, with a mean of 87.5% for the BFmTN(DA) algorithm, and 87.1% for the other two algorithms. The

	BFmTN(DA)	BFmTN($\frac{1}{3}$W)	BFmTN($\frac{2}{5}$R)
Min. μ	56.3%	47.4%	47.4%
Low. Q. μ	79.4%	78.2%	78.0%
Med. μ	85.3%	84.1%	84.0%
Up. Q. μ	90.1%	89.5%	89.1%
Max. μ	98.8%	98.8%	98.4%
IQR	10.7%	11.2%	11.1%
Wang P1 t (s)	4.8691	142.50	129.26
Total Bins	9 158	9 281	9 308
Repacked Bins	1 541	1 662	1 709
% Repacked	16.8%	17.9%	18.4%
Stationary Bins	7 617	7 619	7 599
Mean μ Rank	1.83 (1)	2.08 (2)	2.09 (3)
Nem. μ Class	B	A	A
Mean ν Rank	1.68 (1)	2.12 (2)	2.19 (3)
Nem. ν Class	B	A	A

Table 8.10: *Overview of the plane-packing algorithmic results for the MBSBPP. The row labelled* 'Min. μ' *contains the minimum bin utilisation over the 857 MBSBP benchmark instances, while the rows labelled* 'Low. Q. μ', 'Med. μ', 'Up. Q. μ', 'Max. μ' *and* 'IQR' *contain the lower quartile, median, upper quartile, maximum and interquartile range of the results for the instances, respectively. The row labelled* 'Wang P1 t (s)' *contains the time taken (in seconds) for the algorithms to complete the packing of the first problem by Wang [155], the largest benchmark instance. The row labelled* 'Total Bins' *is the total number bins used over all MBSBP benchmark instances, the row labelled* 'Repacked Bins' *documents the total number of bins that were repacked during the repacking phase of the 2SMBSBP algorithm, the row labelled* '% Repacked' *indicates what percentage of the total number of bins this repack value is, and the row labelled* 'Stationary Bins' *lists the number of bins that were not repacked. The row labelled* 'Mean μ Rank' *documents the mean ranks of the algorithms when applied to the utilisation (the ranks are given in parentheses), while the row labelled* 'Nem. μ Class' *shows which algorithms are not significantly different according to the Nemenyi test [40] by assigning them the same letter. The same tests are performed for the fitness ν in the two rows that follow.*

	BFmTN(DA)	BFmTN($\frac{1}{3}$W)	BFmTN($\frac{2}{5}$R)
Wang P1 μ	83.4%	86.3%	86.8%
Wang P2 μ	90.5%	90.5%	90.5%
Hopper M1	2.30	1.70	2.00
Hopper M2	1.80	2.40	2.40
Hopper M3	2.40	1.50	2.10
PS 1	1.55	2.40	2.05
PS 2	2.01	1.86	2.13
PS 3	1.52	2.34	2.14
PS 4	1.92	2.04	2.04
PS 5	1.38	2.43	2.19
PS 6	1.85	2.06	2.09
PS 7	2.05	2.06	1.89
PS 8	2.07	1.86	2.07
PS 9	1.39	2.17	2.44
PS 10	1.73	2.26	2.01
Nice	1.91	2.00	2.09
Path	1.95	1.99	2.06

Table 8.11: *Plane-packing algorithmic results for the MBSBPP for various sets of benchmark instances. The utilisation achieved for the two problems by Wang are followed by the mean ranks of the algorithms for the Hopper and Turton (labelled* M*), Pisinger and Sigurd (labelled* PS*) and new benchmark instances (split into* Nice *and* Path *instances).*

mean ranks for the algorithms suggest that the BFmTN(DA) algorithm performs very well for the PS 1, PS 3, PS 5 and PS 9 sets. These differences are, in fact, significant according to the Nemenyi test which yields a CD of 0.47 for 3 algorithms and 50 benchmark instances at a confidence level of 95%. The common theme to these benchmark instances is that the items are large relative to the bins. Instances PS 1 and PS 5 are created by setting item widths and heights equal to uniformly random values in the range $[1, D]$, while bin dimensions are selected uniformly in the range $[D/2, D]$, where $D = 10$ for PS 1 and $D = 100$ for PS 5. For PS 3 the item dimensions are selected in the range $[1, 35]$ and the bin dimensions in the range $[20, 40]$. The set PS 9 belongs to this group of instances containing large items relative to their bins, because the dimensions of 70% of the items are selected within the range $[50, 100]$, which is the same range used to select the dimensions for the bins. The BFmTN(DA) algorithm yields significantly better results than the BFmTN$\frac{1}{3}$WDWDH algorithm for the benchmark instance set PS 10 (where the dimensions of 70% of the items are selected uniformly within the range $[1, 50]$), but it is not significantly better than the BFmTN$\frac{2}{5}$RDWDH algorithm for these instances.

	BFmTN(DA)	BFmTN($\frac{1}{3}$W)	BFmTN($\frac{2}{5}$R)
Mean p Rank	1.77 (1)	2.15 (3)	2.08 (2)
Nem. p Class	B	A	A
100 t (ms)	1.6618	7.1294	8.1977
BW 1	20.20	20.58	20.44
BW 2	2.56	2.52	2.58
BW 3	14.54	15.00	14.94
BW 4	2.52	2.54	2.56
BW 5	18.34	19.12	19.02
BW 6	2.32	2.32	2.32
MV 7	16.88	17.48	17.48
MV 8	17.22	17.36	17.38
MV 9	42.74	42.90	42.76
MV 10	10.42	10.68	10.56
Total Bins	7 387	7 525	7 502

Table 8.12: *Plane-packing algorithmic results for the SBSBPP for various sets of benchmark instances. The row labelled* 'Mean p Rank' *shows the mean rank over the 500 benchmark instances in terms of the number of bins packed, while the row* 'Nem. p Class' *shows which algorithms are not significantly different by placing them in the same class, indicated by a letter. Finally, the row labelled* '100 t (ms)' *shows the mean time (in milliseconds) that the algorithms required to solve the SBSBP benchmark instances with 100 items. The results below these rows are the mean numbers of bins for each problem class.*

The results for the SBSBPP in Table 8.12 suggest similar results for the algorithms when compared to the results for the MBSBPP. The BFmTN(DA) algorithm, with a mean rank of 1.77, proves to be significantly better than the BFmTN$\frac{1}{3}$WDWDH and BFmTN$\frac{2}{5}$RDWDH algorithms when considering all 500 instances (the CD is 0.15 for three algorithms and 500 benchmark instances at a confidence level of 95%). However, the Nemenyi test suggests that there is no significant difference between the BFmTN$\frac{1}{3}$WDWDH and BFmTN$\frac{2}{5}$RDWDH algorithms. The result of the ranking is expected if one compares the number of bins required by the algorithms to pack the items. The BFmTN(DA) algorithm requires only 7 387 bins while the BFmTN$\frac{1}{3}$WDWDH and BFmTN$\frac{2}{5}$RDWDH algorithms require 138 and 115 more bins, respectively. The Nemenyi test suggests that the BFmTN(DA) algorithm is significantly better than the BFmTN$\frac{1}{3}$WDWDH algorithm for the BW 1 set of instances and significantly better than both other algorithms for the BW 3, BW 5 and MV 7 instances, corroborating the evidence from the results for the MBSBPP that the BFmTN(DA) algorithm is significantly better than the other algorithms for problem instances where the items are large relative to the bin sizes.

There does not appear to be a significant difference between the three algorithms for the other benchmark instances. The BFmTN(DA) algorithm was able to solve the problems significantly faster than the other algorithms — in approximately a quarter of the time for instances of 100 items.

8.5 Comparison of the Best Heuristics from each Class

In this section the best algorithms from each class are compared with one another. Of the level-packing algorithms the BFDHDW algorithm was shown to yield the best results, or to be part of the set of algorithms that were not significantly different from one another but yielded the best results, for the MBSBPP and the SBSBPP. It was also the second fastest algorithm for the largest problem instance (1.5% slower than the fastest algorithm), but third fastest for the SBSBPP (7.4% slower than the B2FW$_2$DHDW algorithm). Deciding on a single representative from the set of guillotine pseudolevel algorithms is difficult. The FC$_{\text{OG}}$DHDW, BFS and SL$_5$ algorithms cannot be separated in terms of performance by the Nemenyi test. The FC$_{\text{OG}}$DHDW algorithm consistently yields the lowest mean rank for the utilisation and fitness scores for the MBSBPP, and results in the smallest number of total bins packed for both problems. However, this performance comes at a time cost — the FC$_{\text{OG}}$DHDW requires more than double the time required by the other two algorithms to find solutions to the same problems. Therefore, the FC$_{\text{OG}}$DHDW, SL$_5$ (the best in §6.3.1) and SASm (for its speed) algorithms are included in the comparisons of this section. The free-packing pseudolevel algorithms yield very similar results (a significant difference between the FC$_{\text{OF}}$DHDW and SCR algorithms could only be found by comparing their fitness scores), with the FC$_{\text{OF}}$DHDW consistently yielding the lowest mean rank and the smallest number of bins packed. However, this comes at a cost to the solution time — the SC algorithm requires less than half the time required by the FC$_{\text{OF}}$DHDW algorithm to find a solution to the largest MBSBPP instance. The SCR algorithm is slower than the SC algorithm and has a lower mean rank with respect to all solution measures and may be eliminated from consideration. Finally, the BFmTN(DA) algorithm finds significantly better solutions than the other algorithms in the set and does so in the least time. Performing a Friedman test on the utilisation and fitness scores yields $P = 0$, suggesting that the null hypothesis that the algorithms are all equivalent may be rejected. Figure 8.5 contains box plots of the utilisation results for the selected algorithms with respect to the 857 benchmark instances, and Tables 8.13 and 8.14 contain further details regarding the results.

The box plot in Figure 8.5 shows that the performance of the pseudolevel-packing SASm algorithm typically yields results that are worse than those of the level-packing BFDHDW algorithm, a result that is unexpected when taking into consideration that the SASm algorithm may pack items anywhere within a level and not only on the floor as the BFDHDW algorithm is restricted to doing. The strip packing results suggest that the SASm algorithm would have been better yielding lower median and upper quartile values (see Tables 6.7 and 6.8). However, the BFDHDW algorithm yields a distribution of results that is lower than the distribution of the other pseudolevel-packing algorithms in this set, which is an expected result. The FC$_{\text{OG}}$DHDW, SL$_5$, FC$_{\text{OF}}$DHDW and SC algorithms yield very similar distributions, while the BFmTN(DA) algorithm yields a distribution of utilisation scores that is shifted closer to the 100% mark.

These observations are supported by the results in Table 8.13. The SASm algorithm yields the lowest lower quartile, median, upper quartile and maximum utilisation values. It requires the largest number of bins to pack all the items and the result is the worst ranking, the Nemenyi test (with a CD of 0.31 for seven algorithms and 857 instances for a 95% confidence interval)

	BFDHDW	FC_{OG}DHDW	SASm	SL_5	FC_{OF}DHDW	SC	BFmTN(DA)
Min. μ	47.4%	56.1%	47.4%	47.4%	56.1%	56.1%	56.3%
Low. Q. μ	76.9%	78.4%	74.0%	78.1%	78.5%	78.3%	79.4%
Med. μ	83.1%	84.3%	80.0%	84.1%	84.3%	84.1%	85.3%
Up. Q. μ	87.2%	89.1%	84.4%	88.9%	89.2%	89.1%	90.1%
Max. μ	96.9%	97.6%	94.6%	97.3%	97.6%	97.6%	98.8%
IQR	10.3%	10.7%	10.4%	10.8%	10.7%	10.8%	10.7%
Wang P1 t (s)	5.9768	16.082	5.4528	5.9386	16.328	5.8691	4.8691
Total Bins	9 306	9 248	9 642	9 259	9 246	9 280	9 158
Repacked Bins	1 503	1 538	1 840	1 561	1 535	1 610	1 541
% Repacked	16.2%	16.6%	19.1%	16.9%	16.6%	17.3%	16.8%
Stationary Bins	7 803	7 710	7 802	7 698	7 711	7 670	7 617
Mean μ Rank	4.45 (6)	3.71 (3)	5.49 (7)	3.74 (4)	3.69 (2)	3.77 (5)	3.15 (1)
Nem. μ Class	B	C	A	C	C	C	D
Mean ν Rank	4.86 (6)	3.65 (3)	5.76 (7)	3.75 (5)	3.53 (2)	3.73 (4)	2.71 (1)
Nem. ν Class	B	C	A	C	C	C	D

Table 8.13: *Overview of the algorithmic results for a selected set of algorithms for the MBSBPP. The row labelled* 'Min. μ' *contains the minimum bin utilisation over the 857 MBSBP benchmark instances, while the rows labelled* 'Low. Q. μ', 'Med. μ', 'Up. Q. μ', 'Max. μ' *and* 'IQR' *contain the lower quartile, median, upper quartile, maximum and interquartile range of the results for the instances, respectively. The row labelled* 'Wang P1 t (s)' *contains the time taken (in seconds) for the algorithms to complete the packing of the first problem by Wang [155], the largest benchmark instance. The row labelled* 'Total Bins' *is the total number bins used over all MBSBP benchmark instances, the row labelled* 'Repacked Bins' *documents the total number of bins that were repacked during the repacking phase of the 2SMBSBP algorithm, the row labelled* '% Repacked' *indicates what percentage of the total number of bins this repack value is, and the row labelled* 'Stationary Bins' *lists the number of bins that were not repacked. The row labelled* 'Mean μ Rank' *documents the mean ranks of the algorithms when applied to the utilisation (the ranks are given in parentheses), while the row labelled* 'Nem. μ Class' *shows which algorithms are not significantly different according to the Nemenyi test [40] by assigning them the same letter. The same tests are performed for the fitness ν in the two rows that follow.*

	BFDHDW	FC_{OG}DHDW	SASm	SL_5	FC_{OF}DHDW	SC	BFmTN(DA)
Wang P1 μ	84.4% (3.00)	84.4% (3.00)	77.7% (7.00)	84.4% (3.00)	84.4% (3.00)	84.4% (3.00)	83.4% (6.00)
Wang P2 μ	88.2% (7.00)	90.5% (3.50)	90.5% (3.50)	90.5% (3.50)	90.5% (3.50)	90.5% (3.50)	90.5% (3.50)
Hopper M1	95.5% (3.50)	95.5% (3.50)	91.6% (6.30)	95.5% (3.50)	95.5% (3.50)	95.5% (3.50)	94.9% (4.20)
Hopper M2	89.6% (4.70)	90.4% (3.20)	88.0% (5.50)	90.4% (3.20)	90.4% (3.20)	90.8% (2.60)	87.5% (5.60)
Hopper M3	93.9% (4.20)	94.6% (3.00)	91.8% (6.30)	94.9% (2.70)	94.6% (3.00)	93.9% (4.20)	93.3% (4.60)
PS 1	87.9% (4.55)	88.6% (3.78)	86.3% (5.73)	88.7% (3.71)	88.6% (3.78)	88.4% (4.22)	90.1% (2.23)
PS 2	83.6% (4.36)	85.1% (3.90)	82.2% (4.92)	85.1% (3.90)	85.1% (3.90)	85.4% (3.84)	86.9% (3.18)
PS 3	82.1% (3.79)	82.2% (3.62)	75.7% (6.72)	82.0% (3.73)	82.2% (3.62)	81.7% (4.04)	83.6% (2.48)
PS 4	80.4% (4.29)	82.7% (3.73)	78.0% (5.30)	82.0% (3.87)	83.0% (3.66)	83.0% (3.66)	83.6% (3.49)
PS 5	81.2% (4.26)	81.4% (3.65)	78.3% (5.92)	81.3% (3.85)	81.4% (3.65)	81.1% (4.23)	82.9% (2.44)
PS 6	79.1% (4.16)	80.5% (3.83)	77.1% (5.20)	79.6% (3.90)	80.5% (3.83)	80.5% (3.83)	83.0% (3.25)
PS 7	80.4% (4.50)	81.1% (3.90)	80.4% (4.43)	80.9% (4.08)	81.1% (3.90)	80.8% (4.38)	82.3% (2.81)
PS 8	81.3% (3.79)	81.5% (3.67)	79.5% (5.65)	81.5% (3.32)	81.5% (3.67)	81.2% (3.63)	80.5% (4.27)
PS 9	72.8% (4.97)	72.9% (4.61)	72.9% (4.56)	73.0% (3.96)	72.9% (4.61)	73.2% (3.54)	74.7% (1.75)
PS 10	84.0% (4.68)	85.6% (3.42)	79.4% (6.55)	85.3% (3.71)	85.6% (3.42)	85.3% (3.53)	86.6% (2.69)
Nice	81.7% (4.33)	83.0% (3.79)	78.4% (5.30)	82.9% (3.84)	83.1% (3.71)	83.1% (3.74)	84.6% (3.28)
Path	81.3% (4.92)	85.6% (3.40)	78.5% (5.62)	85.1% (3.53)	85.7% (3.36)	85.3% (3.46)	85.0% (3.70)

Table 8.14: *Algorithmic results for the best of the algorithms in this chapter for the MBSBPP for various sets of benchmark instances. The utilisation achieved for the two problems by Wang are followed by the mean ranks of the algorithms for the Hopper and Turton (labelled* M*), Pisinger and Sigurd (labelled* PS*) and new benchmark instances (split into* Nice *and* Path *instances). The mean ranks for each set of instances is shown in parentheses.*

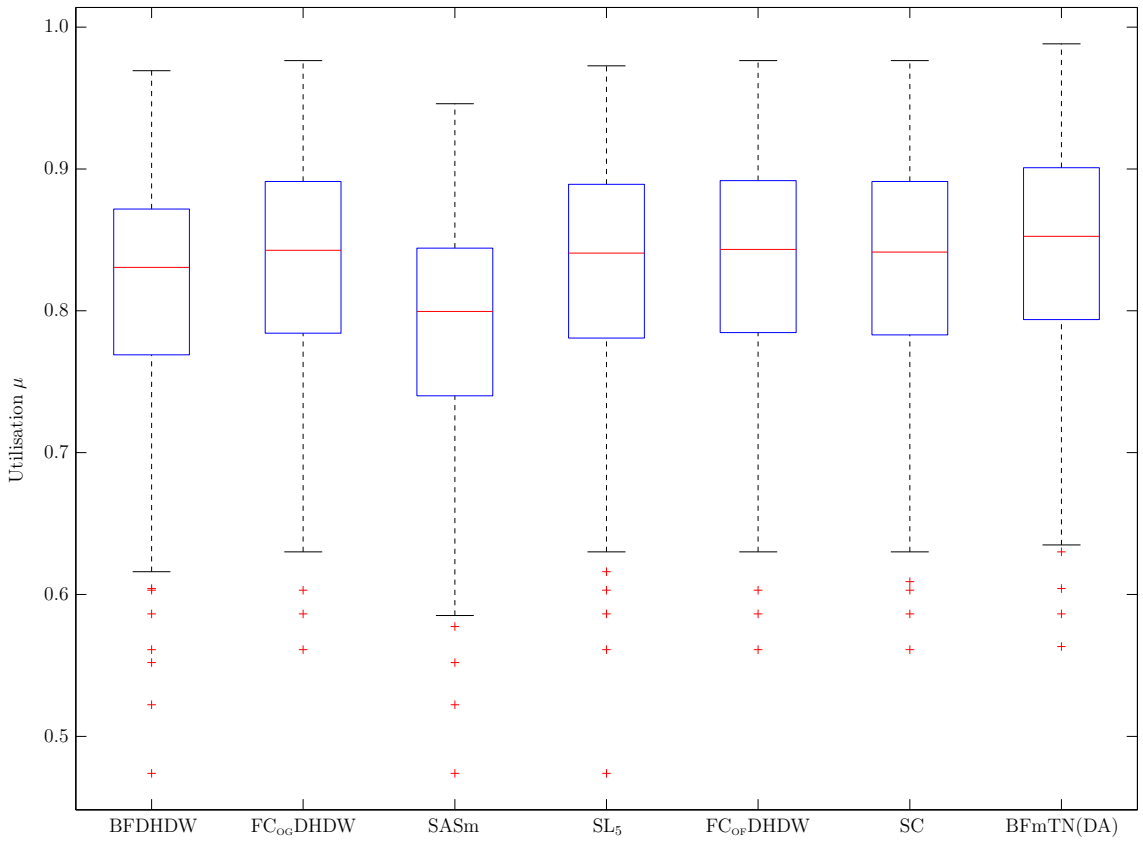


Figure 8.5: *Box plot of the results for the best of heuristics in for the MBSBPP in this chapter when applied to all MBSBPP benchmark instances described in §8.1.*

suggesting that it is significantly worse than the other algorithms in the set. The quartile, median and total bins packed values for the BFDHDW confirm that it is worse than the five remaining algorithms in the set and the Nemenyi test suggests that it is significantly worse than these algorithms with respect to utilisation and fitness results. The test also suggests that there is no significant difference between the pseudolevel-packing algorithms, regardless of whether they pack guillotine feasible solutions or not. However, it is interesting that the FC algorithms yield better mean ranks than the new algorithms. The BFmTN(DA) algorithm yields results that are significantly better than those of the other algorithms in this set. This is an expected result, because the BFmTN(DA) algorithm is not restricted to packing items into levels as the other algorithms are. The BFmTN(DA) algorithm is also faster than all the other algorithms, almost half a second faster than the next fastest algorithm, the SASm algorithm.[2] The BFDHDW, SL_5 and SC algorithms achieve very similar solution times — in the range $[5.87, 5.98]$. The two FC algorithms required the most time to find a solution to the largest MBSBPP — more than double the time of any of the other algorithms in this set.

A closer look at the results show that even though the BFmTN(DA) may be the best algorithm overall, it still can yield worse results than the remaining algorithms for some problem instances.

[2]The SASm algorithm was shown to be twice as fast for the strip packing problem (see Table 6.18) and the speed of the BFmTN(DA) algorithm may be due to the fact that the level and pseudolevel algorithms were accessed by the 2SMBSBP algorithm, which was written generically in order to access any of the level or pseudolevel strip packing algorithms. The BFmTN(DA) algorithm, due to its plane-packing nature, required a more integrated programming approach (in which a new algorithm was programmed that combined the BFmTN and 2SMBSBP algorithms into one) which may have given it a performance advantage to the other algorithms.

For example, the mean rank of the algorithm is better only than that of the SASm algorithm for all three of the benchmark instances by Hopper [75], and worse than the SASm algorithm for M2. However, it is significantly better than the other algorithms for the PS 1 and PS 9 instances (the CD is 1.27 for a 95% confidence interval) and significantly better than all the algorithms excluding the FC algorithms for the PS 5 and PS 7 instances (for which the SL_5 algorithm was also not significantly different). The BFmTN(DA) algorithm achieved the second worst ranking for the PS 8 benchmark instances (70% of the items have their height uniformly selected in the range $[67, 100]$ and their widths in the range $[1, 50]$). The SL_5 algorithm achieved the best mean rank for this set of instances. The BFmTN(DA) algorithm has the best mean rank for the *nice* instances, but is only ranked fifth by mean rank for the *pathological* instances. This is an unexpected result, because of the fact that the algorithm does not require the items to be packed into levels.

The results for the SBSBPP benchmark instances are similar to those for the MBSBPP. The SASm algorithm, when combined with the 2SMBSBP algorithm, yields results that are significantly worse than the remaining algorithms according to the Nemenyi test (the CD is 0.40 for 7 algorithms and 500 instances for a 95% confidence interval), while the BFmTN(DA) algorithm yields significantly better results than the other algorithms in this set. However, the Nemenyi test suggests that the BFDHDW algorithm is not significantly worse than the pseudolevel-packing algorithms for these benchmark instances, even though this was the case for the MBSBPP instances. The BFmTN(DA) algorithm is significantly better than all algorithms for the BW 1 set (the CD is 1.27 for 50 instances at a 95% confidence level), significantly better than the SC algorithm for the BW 5 set of instances and significantly better than the SASm algorithm for the BW3, BW 5, MV 8 and MV 10 sets of benchmark instances.

In an attempt to find a ranking for the algorithms in terms of both utilisation and time, the multiple bin size bin packing efficiency Γ presented in §2.3.2 may be used to compare the algorithms. The results are shown in Table 8.16. The BFmTN(DA) algorithm is ranked first in terms of mean utilisation and mean time over all benchmark instances and is hence ranked first for all values of ℓ. The SASm algorithm begins at second place due to its speed, but rapidly looses its ranking as the utilisation becomes more important (as ℓ increases), ranked fifth by the time $\ell = 10$. At this stage the SC algorithm has moved up the rankings to second place, overtaking the SASm and SL_5 algorithms. The SL_5 algorithm is ranked third at this stage, followed by the BFDHDW algorithm in fourth place. However, as the value of ℓ increases and solution time becomes less important, the ranks of the FC algorithms increase, relegating the SASm and BFDHDW algorithms to last place. This suggests that one would typically use the SL_5 algorithm to find solutions to the guillotine multiple bin size bin packing problem rapidly, unless time is not a factor, in which case use of the FC_{OG} algorithm is desirable. If the guillotine constraint is not required, then the BFmTN(DA) algorithm would typically be the best choice.

8.6 Chapter Summary

In this chapter a comparison was performed between the various algorithms for the MBSBPP. First, a brief overview of the benchmarks available for the MBSBPP was given in §8.1.1 and the method used to generate a new set of benchmark instances was described in §8.1.2. These sections were included in fulfilment of Dissertation Objective XII(a), and also included a set of benchmark instances for the SBSBPP in fulfilment of Dissertation Objective XII(b). This was followed by an appraisal of selected level-packing algorithms when combined with the 2SMBSBP algorithm in §8.2, of pseudolevel algorithms in §8.3 (which were split into guillotine algorithms in

	BFDHDW	FC_{OG}DHDW	SASm	SL_5	FC_{OF}DHDW	SC	BFmTN(DA)
Mean p Rank	3.97 (5)	3.83 (3)	5.19 (7)	3.89 (4)	3.82 (2)	4.02 (6)	3.28 (1)
Nem. p Class	B	B	A	B	B	B	C
100 t (ms)	1.8920	4.2915	1.6787	2.0243	3.9288	2.1285	1.6618
BW 1	20.62 (3.97)	20.60 (3.90)	21.24 (5.74)	20.62 (3.96)	20.60 (3.90)	20.62 (3.95)	20.20 (2.58)
BW 2	2.64 (4.09)	2.60 (3.95)	2.70 (4.30)	2.60 (3.95)	2.60 (3.95)	2.60 (3.95)	2.56 (3.81)
BW 3	14.72 (3.52)	14.72 (3.52)	16.26 (6.74)	14.80 (3.75)	14.72 (3.52)	14.86 (3.93)	14.54 (3.02)
BW 4	2.60 (4.09)	2.56 (3.95)	2.70 (4.44)	2.56 (3.95)	2.54 (3.88)	2.54 (3.88)	2.52 (3.81)
BW 5	18.70 (3.76)	18.70 (3.76)	19.66 (6.03)	18.70 (3.76)	18.70 (3.76)	18.88 (4.28)	18.34 (2.65)
BW 6	2.36 (3.98)	2.36 (3.98)	2.44 (4.26)	2.36 (3.98)	2.36 (3.98)	2.36 (3.98)	2.32 (3.84)
MV 7	17.18 (4.24)	17.10 (3.96)	17.22 (4.36)	17.12 (4.03)	17.10 (3.96)	17.18 (4.24)	16.88 (3.21)
MV 8	17.52 (3.93)	17.50 (3.86)	18.08 (5.47)	17.54 (3.99)	17.50 (3.86)	17.52 (3.93)	17.22 (2.96)
MV 9	42.78 (3.85)	42.78 (3.85)	43.00 (4.62)	42.78 (3.85)	42.78 (3.85)	42.90 (4.27)	42.74 (3.71)
MV 10	10.74 (4.27)	10.52 (3.57)	11.42 (5.98)	10.54 (3.63)	10.52 (3.57)	10.58 (3.75)	10.42 (3.23)
Total Bins	7 493	7 472	7 736	7 481	7 471	7 502	7 387

Table 8.15: *Algorithmic results for the best of the algorithms in this chapter for the SBSBPP for various sets of benchmark instances. The row labelled* 'Mean p Rank' *shows the mean rank over the 500 benchmark instances in terms of the number of bins packed, while the row* 'Nem. p Class' *shows which algorithms are not significantly different by placing them in the same class, indicated by a letter. Finally, the row labelled* '100 t (ms)' *shows the mean time (in milliseconds) that the algorithms required to solve the SBSBP benchmark instances with 100 items. The results below these rows are the mean numbers of bins for each problem class.*

ℓ	BFDH	FC_{OG}	SASm	SL_5	FC_{OF}	SC	BFmTN
$\bar{t}$ Rank	5	7	2	3	6	4	1
$\ell = 1$	5	7	2	3	6	4	1
$\ell = 2$	5	7	3	2	6	4	1
$\ell = 3$	5	7	4	2	6	3	1
$\ell = 4$	4	7	5	2	6	3	1
$\ell = 5$	4	7	5	2	6	3	1
$\ell = 10$	4	7	5	3	6	2	1
$\ell = 20$	4	6	7	3	5	2	1
$\ell = 50$	6	5	7	3	4	2	1
$\ell = 300$	6	5	7	4	3	2	1
$\ell = 500$	6	4	7	5	2	3	1
$\overline{\mu}$ Rank	6	3	7	5	2	4	1

Table 8.16: *Ranks of the best algorithms, based on the multiple bin size bin packing efficiency. The row labelled '$\bar{t}$ Rank' contains the algorithms' ranks with respect to the mean solution time over the 857 MBSBPP benchmark instances. The row labelled '$\overline{\mu}$ Rank' contains the algorithms' ranks with respect to the mean utilisation over the 857 benchmark instances.*

§8.3.1 and free-packing algorithms in §8.3.2) and of selected variations of the BFmTN algorithm in §8.4, in fulfilment of Dissertation Objectives XIII(a) and XIII(b). Finally, the best algorithms from each set were compared in §8.5.

CHAPTER 9

Conclusion

Contents

The purpose of this chapter is to summarise work contained in this dissertation. In §9.1 a detailed dissertation summary is provided, followed by a presentation of the main contributions of this dissertation to the field of C&P problems in §9.2. Finally, an appraisal of the dissertation contributions is performed in §9.3.

9.1 Dissertation Summary

A brief introduction to the history of C&P problems was given in the first chapter of this dissertation. This was followed by a brief consideration of the various names that have been given to C&P problems in the literature and some of the applications of these problems. The objectives pursued in this dissertation were outlined and the chapter closed with a presentation of the structure of the dissertation.

The second chapter contained a more detailed study of C&P problems. It began with a presentation of two typologies of C&P problems by Dyckhoff [43] and Wäscher *et al.* [157] in §2.1.1 and §2.1.2, respectively, which were followed by the subtypologies of Lodi *et al.* [101, 105] in §2.1.3 and Ntene [125] in §2.1.4, during which the concepts of orthogonality and guillotine packings, as well as regular versus irregular shapes were introduced. An attempt was made to clarify how the typologies are related, and they were then used to describe the scope of C&P problems considered in this dissertation in fulfilment of Dissertation Objective I, as described in §1.3. A discussion on available packing problem solution methodologies followed in §2.2 with brief descriptions of the notions of heuristic, metaheuristic and exact methods, and to which C&P problems they have been applied in the past, in fulfilment of Dissertation Objective II. This was followed by a description of the scope of the solution methodologies considered in the remainder of the dissertation. Finally, a description of various methods of evaluating algorithms was provided, including both theoretical evaluation methods (such as worst-case time complexity estimates and performance bounds in §2.3.1), and computational evaluation methods in §2.3.2, in fulfilment of Dissertation Objective III.

Chapter 3 was dedicated to a detailed description of various level-packing algorithms for the strip packing problem in the literature, in fulfilment of Dissertation Objective IV(a). After a brief introduction, the NFDH algorithm [32] was described in §3.2.1 and a pseudocode listing was given, before an example of the working of the algorithm was given in some detail. This was followed by the performance bounds for the NFDH algorithm that had been established by Coffman *et al.* [32], and a calculation of the worst-case time complexity of the algorithm. The NFDH algorithm was followed by the FFDH algorithm in §3.2.2, another algorithm proposed by Coffman *et al.* [32], and was studied in the same manner as the NFDH algorithm. The BFDH algorithm first described by Berkey and Wang [16], but subsequently studied in detail by Coffman and Shor [34], was described in a similar manner in §3.2.3. The KP algorithm by Lodi *et al.* [105] was also described in §3.2.4 by means of a written description, a pseudocode listing, a worked example and a calculation of the worst-case time complexity, but was further described by means of practical considerations for the computational implementation of the algorithm. This led naturally to a brief description of the time-restricted KP_{TR} algorithm. The JOIN algorithm (described in §3.2.5) proposed by Martello *et al.* [110] as a heuristic to find feasible solutions for their exact method was included in the description of the algorithms and some variations were proposed. This concluded the descriptions of the known level-packing algorithms.

Two new level algorithms were additionally proposed in Chapter 3 in fulfilment of Dissertation Objective V(a). The new WFDH algorithm (see §3.3.1) is based on the WFD algorithm for 1D packing by Johnson [84] and expands the principle of selecting the bin (level in the case of strip packing) with the largest remaining space. The novel B2FDH algorithm described in §3.3.2 is based on another 1D bin packing heuristic, the B2F algorithm by Friesen and Langston [55]. The algorithm packs levels until no unpacked items fit into the remaining space (if such a space exists). An attempt is then made to repack the last item to be packed into the level. This may happen in one of two ways: either two items replace the occupant if their combined width is greater than that of the occupant's width (and no greater than the space remaining when the last item is removed), or if their combined area is greater than that of the occupant.

The largest number of new algorithms appeared as a collection of novel pseudolevel algorithms in Chapter 4 (in fulfilment of Dissertation Objective V(b)), but a number of known pseudolevel algorithms were first described in fulfilment of Dissertation Objective IV(b). Practical considerations for the computational implementation of these algorithms were first proposed, including a method of implementation of a so-called skyline of the items that have been packed. The first known pseudolevel algorithms considered were the FC algorithms by Lodi *et al.* [106] described in §4.2.1, of which the guillotine and free-packing versions of the oriented variation were described in some detail. Bortfeldt [18] also proposed a pseudolevel-packing algorithm (and named it the BFDH* algorithm) that allows rotations, but the algorithm described in some detail in §4.2.2 was the variation that he designed for the oriented problem. In order to pack the items efficiently, two lists were generated that allow the packing of items sorted by height to be removed efficiently from the list of items sorted according to decreasing area. The section on the known pseudolevel-packing algorithms was concluded by a review of the SAS algorithm by Ntene and Van Vuuren [125, 127] in §4.2.3.

A modification to the SAS algorithm initiated the section describing the new pseudolevel algorithms in Chapter 4. Ntene and Van Vuuren did not exploit certain opportunities for improving the packing density and those identified by the author were added to the algorithm, as described in §4.3.1. This includes a change to the method of sorting items, a change to the method of comparing wide and narrow items when initialising a level, and two changes to the stacking rules. In addition a new BFS algorithm was proposed in an attempt to improve on the BFDH* algorithm

proposed by Bortfeldt [18] and was presented in §4.3.2. The spaces defined by Bortfeldt are filled by stacking as many items as possible, instead of only packing the largest items by area along the floor of those spaces. The novel SL algorithm described in §4.3.3, was proposed in an attempt to utilise the stacking strategy for the BFS algorithm with the added advantage of a wider stacking platform gained by the joining of items in a manner similar to that in the JOIN algorithm by Martello *et al.* [110]. This concluded the presentation of new pseudolevel-packing algorithms that guarantee a guillotine layout. The chapter was concluded in §4.3.4 by the description of two new non-guillotine algorithms that attempt to stack items from the ceiling, with the aim of utilising some advantages of the FC and stacking algorithms. The SC algorithm fills the floor of a level as much as it can before stacking items downwards from the ceiling. The SCR algorithm performs the same procedure, with the items being re-sorted according to decreasing width before the ceiling packing takes place. The unpacked items are then re-sorted according to decreasing height for the floor-packing of the next level.

Plane-packing algorithms do not follow the rule of packing items into levels in the manner of the level-packing algorithms of Chapter 3 and the pseudolevel-packing algorithms of Chapter 4. Chapter 5 contained descriptions of a number of the plane-packing algorithms in the literature, in fulfilment of Dissertation Objective IV(c). Sleator's algorithm [148], described in §5.1.1, was the first known algorithm reviewed. It packs all items wider than half the strip width first, before packing one level of the remaining items sorted according to decreasing height. The level is then split vertically in half and the two halves become substrips which are filled in a FFDH manner, where the substrip with the lowest height relative to the main strip is filled first.

The SF algorithm by Coffman *et al.* [32], described in §5.1.2, splits the items into two groups; namely wide and narrow items. The wide items are packed according to the FFDH algorithm, and the resulting levels are shifted in the hope of opening up a space, which may be filled with narrow items by means of the FFDH algorithm. The unpacked items may be packed above the wide items in an FFDH manner. The description of the algorithm was followed by a section on practical considerations regarding the implementation of shifting levels in a computationally efficient manner.

The BL algorithm by Baker *et al.* [6] is a famous plane-packing algorithm that, as described in §5.1.3, sorts items according to decreasing width and packs them as low, and far to the left as possible. The description of the algorithm was followed by a detailed discussion on how one may keep track of packed items with the use of skylines, and how to solve the problem of possible overhangs.

Two algorithms proposed by Golan [62] were reviewed in some detail in §5.1.4 and §5.1.5. The SP algorithm begins by sorting the items according to decreasing width and with the packing of items the strip is split into regions that become narrower as the packed items become narrower. Items are packed into regions that are lowest and wide enough to accommodate them. Two improved variations of the algorithm were proposed in partial fulfilment of Dissertation Objective V(c). The first variation packs items into spaces that would be wasted by the original algorithm in a stacking manner similar to that in the BFS and SL algorithms, yielding a guillotine feasible layout. The second proposed variation moves items downward until they encounter another item. This variation does not guarantee a guillotine feasible layout. Golan's M algorithm partitions the items into five sets, which are all packed into various regions by means of the FFDH algorithm.

The UD algorithm by Baker *et al.* [5], as described in §5.1.6, also partitions the items into five sets according to their sizes. These sets of items are then packed into specific regions by means of the BL algorithm, or a generalised NFDH algorithm that packs items into a space that is not

necessarily rectangular.

Chazelle's BLF algorithm [25], described in §5.1.7, was originally proposed in an attempt to improve the BL algorithm by storing the locations of empty spaces in the strip or bin, and packing items into the lowest, leftmost spaces that are large enough to accommodate them. The description of the algorithm was followed by a note on the management of memory when implementing the algorithm computationally (a problem previously encountered by Berkey and Wang [16] when they proposed algorithms for the SBSBPP).

The GCS algorithm by MacLeod *et al.* [109], while designed for the cutting stock problem, was described in §5.1.8 as an example of a plane-packing algorithm that guarantees a guillotine feasible layout. This required careful control of the placement of items, including repeated investigations of existing cuts for various item locations in order to find the lowest, leftmost position for the item that would yield a guillotine feasible layout of items.

The final algorithm described in Chapter 5 was the BFLM algorithm by Burke *et al.* [22, 23] in §5.1.9. The original BFLM algorithm allowed for item rotations, but Ntene [125] had previously demonstrated how the algorithm may be applied to oriented strip packing. After a description of the algorithm as well as two variations of the algorithm, a modification to the algorithm was proposed that would allow the items to be sorted in any manner, not only decreasing width, and would pack the first item in the list that fits into the lowest skyline segment. This was done in partial fulfilment of Dissertation Objective V(c).

Chapter 5 was concluded with a proposal of a new classification of algorithms in terms of their dependence on the order in which the items have been sorted in §5.2. Sorting-dependent algorithms yield poor results when items are not sorted in a specific order, or they assign items into groups which may then be sorted according to a specific manner. Sorting-independent algorithms, described in §5.2.2, may yield very good results if a randomly-sorted list of items is packed. Two new sorting strategies were proposed for these algorithms with the aim of finding lower packing heights.

An appraisal of the various known and new strip packing algorithms of Chapters 3–5 was presented in Chapter 6, in fulfilment of Dissertation Objective VIII. The sources of the benchmark problem instances used to test the algorithms were listed in §6.1 in fulfilment of Dissertation Objective VII, and the methods used to generate these instances were described in some detail. Thereafter, a brief description of a number of statistical methods used to compare the algorithms was given. The following conclusions were reached during a numerical comparison of the algorithms at a 95% level of significance with respect to the benchmark problem instances:

- The NFDH, FFDH, BFDH and WFDH algorithms and their variations were the first set of algorithms to be described (see §6.2.1). The distributions of the results for the FFDH and BFDH algorithms were closer to optimal than those of the NFDH and WFDH algorithms, and of these algorithms, the BFDHDW algorithm resulted in the best mean rank and was shown to be significantly better than nine of the eleven other algorithms.

- In §6.2.2 the results of applying the KP family of algorithms to the benchmark instances were summarised, with the conclusion that two of the three time-restricted algorithms are not significantly different from the algorithm that has no time restriction on the solution of knapsack problems (only incorporating a one-hour time-out restriction). However, the time-restricted algorithms were shown to find solutions in significantly shorter times.

- In §6.2.3 it was demonstrated that the JOIN algorithm variants joining items horizontally performed better than those that joined items vertically, that the lower values of δ are

likely to yield better packing heights more often than larger values of δ, and that the JOIN_0DHDW algorithm yields the best mean rank and belongs to the subset of fastest algorithms within the set of JOIN algorithms.

- The results from the B2F algorithms, presented in §6.2.4, comprise the largest set of algorithms compared to one another. It was shown that the B2FA algorithms outperformed the B2FW algorithms by a significant margin, and that of these algorithms, those that sorted the items according to DHDW yielded the best results on average. The algorithm with the best mean ranking was the B2FA_{10}DHDW algorithm.

- The section on the results of the level-packing algorithms was concluded with a section comparing the best algorithms from each set in §6.2.5. No significant difference could be found between the BFDHDW algorithm and the B2FA_{10}DHDW algorithm in terms of packing density and time.

The pseudolevel algorithms were separated into two groups in §6.3; those that are guaranteed to yield a guillotine feasible layout, and those that do not adhere to the guillotine constraint. In §6.3.1, the guillotine pseudolevel algorithms were compared, and it was found that:

- The SL_5 algorithm yielded the best mean ranking, but it was not significantly different to the FC_{OG}DHDW, SL_0 and SL_{10} algorithms.

- The two SAS algorithms yielded the worst results in terms of packing height for this set, but they were the two fastest algorithms tested in this dissertation for large problems.

- The BFS algorithm was shown to be significantly better than the BFDH* algorithms for these benchmark instances. However, it was not significantly different to the FC_{OG}DH, FC_{OG}DHIW and SL_{15} algorithms in terms of packing height, but it is faster than the FC_{OG} algorithms for large data sets.

The results for the free-packing pseudolevel algorithms in §6.3.2 suggest that:

- The SC algorithm finds significantly better solutions than the FC_{OF} algorithms, and finds these solutions in approximately half the time for large instances.

- The SCR algorithm yielded significantly worse results than the FC_{OF}DHDW algorithm, but significantly better results than the FC_{OF}DHIW algorithm, which was the worst in the set by a significant margin. The SCR algorithm was shown to be the slowest algorithm in the set by a significant margin.

The comparison of plane-packing algorithms began in §6.4. The purpose of this section was to determine which algorithms, in a number of sets, were the best in terms of packing height and execution time.

- First, the smaller set of algorithms that are not guaranteed to yield guillotine results were compared to one another, including the M and UD algorithms, the modified version of the SP algorithm and Sleator's algorithm. It was found that the SPmF and M algorithms yield significantly better results than the other algorithms, but the M algorithm was the best of the algorithms, because it required significantly less time to find a solution than did the SPmF algorithms.

- The guillotine algorithms, including the SF and SP algorithms, and the modified versions of the SP algorithms, were compared in §6.4.2. It was found that the SPmG algorithms yielded significantly better results than the other algorithms, but they required significantly more execution time. The SP algorithms were significantly faster than the other algorithms, but were significantly worse in terms of packing height achieved.
- The application of 23 sorting methods to the BL algorithm (see §6.4.3) was performed with the aim of finding a good sorting strategy for this sorting-independent algorithm. It was found that the item lists sorted by height, or according to the new xWDWDH sorting strategy yielded the best results, with the BL$\frac{1}{2}$WDWDH algorithm yielding the lowest mean ranking.
- The results for the BLF and GCS algorithms, presented in §6.4.4 and §6.4.5, respectively, showed a similar pattern with the BLF$\frac{2}{5}$WDHDW algorithm resulting in the lowest mean ranking of the BLF algorithms, and the GCS$\frac{1}{2}$WDWDH yielding the best mean rank of the various GCS algorithms.
- The results of the leftmost variation of the 2D BF algorithm were the first of the three variations to be presented (see §6.4.6). The BFmLM$\frac{1}{2}$WDWDH algorithm may have yielded the best mean rank, but the Nemenyi test suggested that sorting the items according to decreasing area yields results that are not significantly different, but in significantly less time.
- The results of the BFmTN algorithm were presented in §6.4.7, and it was found that the BFmTN$\frac{1}{3}$WDWDH algorithm yielded the best results in terms of packing height, but at the cost of slower execution time. The variations that sorted items according to decreasing area yielded close (yet significantly different) results in significantly better times.
- The results for the BFmSN algorithm, presented in §6.4.8, were different in that the sortings according to decreasing area yielded the best mean ranks and the fastest solution times. It was demonstrated that the xRDWDH sorting method is not an efficient packing strategy for the 2D BF algorithms.

The chapter closed with a comparison of the best algorithms from the pseudolevel and plane-packing sets in §6.4.9. The results suggested that the SASm algorithm yields the fastest solutions, but it is also one of the worst algorithms in terms of packing height achieved. An unexpected result was that the pseudolevel algorithms SL_5 and SC yielded better mean ranks in this set than did plane-packing algorithms such as the M algorithm, the BL(DHDW) and BL$\frac{1}{2}$WDWDH algorithms, and the BFmSN(DADW) algorithm, which was already an improvement on the original oriented BFSN algorithm. These pseudolevel algorithms were able to find results in less time than the plane-packing algorithms, excluding the modified BF algorithms which sorts items according to decreasing area. The BFmTN$\frac{1}{3}$WDWDH algorithm was demonstrated to yield the best mean rank and the Nemenyi test suggested that it was significantly better than the other algorithms. The BFmTN(DADW) and BLF$\frac{2}{5}$WDWDH algorithms were shown to be the next best algorithms and not significantly different from one another, but the BFmTN(DADW) found results in less time.

Chapter 7 contained a summary of relevant literature on the use of heuristics for solving the 2D SBSBPP (in fulfilment of Dissertation Objective IX(b), as described in §1.3) and the 1D MBSBPP, as well as non-heuristic methods with which the 2D MBSBPP has been solved in fulfilment of Dissertation Objective IX(a). The literature on the 2DSBSBPP was presented in §7.1.1, with a brief description of the HFF algorithm proposed by Chung *et al.* [28] in which the

FFDH algorithm (to solve the strip packing problem) was combined with the FFD algorithm to pack the levels into bins (a 1D SBSBPP). This was followed by description of the algorithm proposed by Bengtsson [14], who developed a packing algorithm that iteratively repacked items in an attempt to use fewer bins. The HNF algorithm by Frenk and Galambos [52] packs items into bins in a next-fit manner, while Berkey and Wang [16] developed a number of heuristics for the SBSBPP, the best of which appeared to be the FBS (HBF) algorithm. Lodi *et al.* [101,105] developed three new algorithms for the oriented problem, including the FC (packing items from the ceiling into levels in a strip, then repacking the levels), KP (which packs items into levels of a strip by means of a knapsack problem for each level) and AD (which packs items as low as possible from left-to-right and right-to-left, alternating until no items remain unpacked) algorithms. The HBP algorithm by Boschetti and Mingozzi [19] was designed to pack items of which some may be rotated and others may not, and included an iterative process of changing item prices for each iteration in the hope of finding better solutions. The final algorithm described in this section was the IMA algorithm by El Hayek *et al.* [67]. This algorithm divided the spaces in bins into rectangles that were then filled by means of a best-fit approach which assigned a score to the item/space pair, and packed the item into the space where the score of the pair was largest.

The description of heuristics for the MBSBPP commenced in §7.1.2. First, the FFDLR strategy by Friesen and Langston [54] was described in which items are packed into the largest bins first before an attempt is made to repack the items in those bins into smaller bins. This was followed by a description of their FFDLS algorithm in which items were shifted into smaller bins under certain conditions. The four packing strategies by Chu and La [27], in which items are packed according to relative or absolute waste, using only the largest available, or any bins, were briefly described, before summarising the work by Kang and Park [87] on the MBSBPP. They developed the IFFD and IBFD algorithms in which items are packed into bins by means of the BFD or FFD strategies, before being repacked into smaller bins. The section was concluded with a brief summary of various non-heuristic methods that other researchers have used to solve the 2D MBSBPP.

A new heuristic for the 2D MBSBPP was presented in §7.2 in fulfilment of Dissertation Objective X. The design of the algorithm utilised the idea of repacking from the FFDLR strategy by Friesen and Langston [54]. The algorithm packs items into a strip, before repacking the resulting levels into bins (that have been sorted according to decreasing area) in a manner similar to the hybrid packing algorithms for the 2D SBSBPP. If a bin has been filled by the levels, then the remaining levels are packed into the next bin, if the bin has the same width as the previous bin. If the bin width changes, then the items are packed into a new strip of the same width as the width of the new bin, before the new levels are packed into bins. Once all the items are packed, the bin with the smallest combined area of items is selected and the smallest bin of area no smaller than the area of the items is selected. An attempt is made to repack those items into the smaller bin and if they fit they remain in the smaller bin. If the items do not fit, then an attempt is made to repack the items into another bin. Once the items have been repacked, or if they could not be repacked, then the bin with the next smallest area of items is selected for repacking. This process continues until an attempt has been made to repack all bins. The detailed description of the algorithm was followed by a worked example and a calculation of its worst-case time complexity. The chapter closed with a brief description of modifications made to the best strip packing algorithm of Chapter 6, the BFmTN algorithm, for it to solve instances of the MBSBPP.

The results obtained by applying the 2SMBSBP algorithm in conjunction with the strip packing algorithms were provided in Chapter 8, in fulfilment of Dissertation Objective XIII(a). The

chapter opened with a description of the benchmark instances that were available in the literature for the MBSBPP, in fulfilment of Dissertation Objective XII(a), including those by Wang [155], Hopper [75] and Pisinger and Sigurd [137]. Sets of instances regularly used for the SBSBPP by Berkey and Wang [16] and by Martello and Vigo [112] were also included in fulfilment of Dissertation Objective XII(b). A new generator of benchmark instances was proposed in §8.1.2 which uses the principles of "nice" and "pathological" data from Wang and Valenzuela [156] in order to test the 2SMBSBP algorithm on a greater variety of problems. It was used to generate 340 new benchmark instances; half of which are "nice" instances and the other half are "pathological" instances.

These benchmark instances were used to evaluate the algorithms and the results were provided in the remainder of Chapter 8. First, the results from the level-packing algorithms combined with the 2SMBSBP algorithm were discussed in some detail in §8.2. Mean ranks were found for all algorithms and the best from each family of algorithms were selected to be compared to one another. This included the BFDHDW, KP_{TR}DHDW, $JOIN_0$DHDW, $B2FA_{10}$DHDW and $B2FW_2$DHDW algorithms. It was found that the JOIN algorithm performed worst within the set in terms of packing density, followed by the new $B2FW_2$DHDW algorithm. Of the remaining algorithms the BFDHDW algorithm was found to be significantly better than the others only by means of the mean rank of the fitness scores. No significant difference could be found between the KP_{TR}DHDW and $B2FA_{10}$DHDW algorithms in terms of either utilisation or fitness scores. However, the KP_{TR}DHDW algorithm was more than one hundred times slower than the other algorithms for the largest instance, which was an expected result. The results were very similar for the SBSBPP.

The results obtained via the pseudolevel algorithms when combined with the 2SMBSBP algorithm were presented in §8.3, with the guillotine algorithms being separated from the free-packing algorithms into §8.3.1 and §6.3.2, respectively. The worst of the guillotine algorithms was the SASm algorithm, which was an expected result considering its performance for the strip packing problem (see §6.3.1). The mean ranks for the fitness scores were able to show a significant difference between the BFDH*(DW) algorithm and the three other algorithms, between which there was no significant difference. Of these three algorithms the FC_{OG}DHDW achieved the lowest mean rank, followed by the SL_5 and the BFS algorithms. However the FC_{OG}DHDW algorithm required more time to find a solution to the largest benchmark instance than did the BFS or SL_5 algorithms, while the SASm algorithm remained the fastest. Of the free-packing pseudolevel algorithms, the FC_{OF}DHDW algorithm achieved the lowest mean rank, but was not significantly different from the other algorithms in terms of utilisation. However, it was significantly better than the SCR algorithm when using the fitness scores to calculate the mean rank. As expected, the FC_{OF}DHDW required more time than did the SC algorithm to find a solution to the largest benchmark instance. The results for the SBSBPP yielded the same pattern, with no significant difference between algorithms in terms of the numbers of bins packed, but with the FC_{OF}DHDW algorithm yielding the best mean rank (at a greater time cost).

The results of the BFmTN algorithm combined with the 2SMBSBP algorithm were presented in §8.4. An unexpected result was encountered when the DA sorting yielded significantly better results than the $\frac{1}{3}$WDWDH sorting method for the MBSBPP benchmark instances, which had been demonstrated to be significantly better than the DA sorting method in §6.4.7. The BFmTN(DA) algorithm was also the fastest of the three sorting methods for the largest benchmark instance by a large margin. The same result was found for the SBSBPP instances.

The chapter closed with a comparison of some of the best algorithms from each class. As expected, the representative algorithm from the set of BFmTN algorithms yielded the best

results. The FC_{OF}DHDW algorithm may have achieved the second best mean rank, but it was not significantly different from the FC_{OG}DHDW, SL_5 and SC algorithms. In order to compare the algorithms to one another in terms of utilisation and solution time, the multiple bin size bin packing score Γ was used to show that the BFmTN(DA) would be the best algorithm to use if a guillotine layout is not desired, while the SL_5 algorithm finds a good balance between speed and packing density if a guillotine feasible layout is desired.

9.2 Main Contributions of this Dissertation

An attempt is made in this section to clarify the contributions made in this dissertation to the field of C&P problems. The contributions are listed in the order in which they appear in the dissertation.

Contribution 1 *A summary of typologies for C&P problems in §2.1.*

The first contribution that is made in this dissertation is the summary of the four typologies and subtypologies from the literature. These typologies were used to define an aspect of the scope of the problems that are addressed in this dissertation, but this contributed to the most complete summary of typologies that has been done since Ntene [125], who summarised the Dyckhoff [43] and Wäscher *et al.* [157] typologies and proposed a typology of her own. She did not consider the typology by Lodi *et al.* [101, 105].

Contribution 2 *The definition of a pseudolevel packing in §2.2.1.*

The difference between level and pseudolevel algorithms was explained in order to differentiate between those level or shelf algorithms that pack all items along the floor, and those that may pack items anywhere within levels. It was shown how the pseudolevel algorithms are a subset of plane-packing algorithms, and how level-packing algorithms are a subset of pseudolevel algorithms. Pseudolevel-packing algorithms include the FC_{OG}, FC_{OF}, BFDH* and SAS algorithms from the literature. Five new pseudolevel algorithms were proposed, namely the SASm, BFS, SL, SC and SCR algorithms.

Contribution 3 *The definition of the strip packing efficiency* Γ^{SP} *and the multiple bin packing efficiency* Γ^{MS} *in §2.3.2.*

These two measures were introduced so that the packing performance of an algorithm (strip height for the SP problem, and utilisation for the MBSBPP) could be combined with computation time in order to compare algorithms to one another in terms of two measures. The scores resulting from the application of the efficiency formula to the results of the algorithms may be used to rank algorithms. The higher the score, the better the algorithm. The ranks for the algorithms being compared may be used by a decision maker to decide which algorithms to use for certain applications.

Contribution 4 *The WFDH algorithm described in §3.3.1, and the variations thereof.*

While well-known algorithms such as the NFD, FFD and BFD algorithms for 1D SBSBP problems were adapted for the strip packing problem, the author was unable to find any references to a strip packing adaptation of the WFD algorithm by Johnson [84]. The algorithm was shown to yield worse results than the FFDH and BFDH algorithms, and their variations.

Contribution 5 *The B2FA$_k$DH and B2FW$_k$DH algorithms and their variations, as described in §3.3.2.*

In an attempt to find an algorithm that would find better solutions than the BFDHDW algorithm, the method of replacing an incumbent item in a level with two smaller items from the B2F algorithm by Friesen and Langston [55] was introduced. The poor performance of the B2FW family of algorithms was perhaps not surprising, because the items following the incumbent in the sorted list are often likely to be shorter than the incumbent. This means that while some space may have been used width-wise that had been wasted before the swap, a taller item (the incumbent) may initialise the next level, thereby possibly wasting vertical space. The B2FA family of algorithms are expected to yield better results, because they only make an attempt to replace an item if the combined area of the new items is larger than that of the incumbent. This may reduce the likelihood of an unnecessarily tall item initialising the next level, while preventing the wasting of horizontal space. Unfortunately the new algorithm was unable to find significantly better solutions than the BFDHDW algorithm over all 1 170 benchmark instances. The B2FA$_{10}$DHDW algorithm was able to find a better solution than the BFDHDW algorithm for 234 benchmark instances, and the BFDHDW algorithm found better solutions 285 times (the two algorithms found the same solutions for 651 benchmark problem instances).

Contribution 6 *The improvements to the SAS algorithm by Ntene and Van Vuuren [125,127], as described in §4.3.1.*

During the design of the SAS algorithm a few improvement opportunities remained unexploited and an attempt was made to include these in the SASm algorithm. The results for the strip packing problem suggested that the new algorithm is significantly better (see §8.3.1), and the time results suggest that the SASm algorithm is also faster than the SAS algorithm. However, the significance test could not distinguish between the solution times at a 95% level of significance. The SASm algorithm was the fastest of all the algorithms for large strip packing problem instances and the large instances of the MBSBPP. However, the improvements to the SAS algorithm were not sufficient for finding better solutions than the other pseudolevel algorithms, relegating the two algorithms to last place in terms of packing height for the strip packing problem.

Contribution 7 *The BFS algorithm presented in §4.3.2.*

In an attempt to make more use of the space between the floor and ceiling, and ensure the best-fit principle was used, the BFS algorithm was designed to stack items onto an item when it is packed onto the floor of a level. It resembles the BFDH* algorithm by Bortfeldt [18], but it was shown to be significantly better for the strip packing, MBSBP and SBSBP problems. This is an expected result because the BFS algorithm may stack items any number of times until the stacking height is constrained by the ceiling of the level, while the BFDH* algorithm only packs one layer of items onto the floor-packed items. The FC$_{\text{OG}}$DHDW algorithm was shown to be significantly better than the BFS algorithm, while the original FC$_{\text{OG}}$DH algorithm was not.

Contribution 8 *The SL algorithm presented in §4.3.3.*

One weakness of the BFS algorithm is that the stacking procedure limits the width within which items may be stacked to the width of the floor-packed item. The design of the SL algorithm was an attempt at allowing a larger width for items to be stacked in, thereby allowing for the possibility of short and wide items being stacked onto taller, but narrower items. By allowing the items that are "joined" to have a slightly different height, some vertical space is sacrificed in order to waste less horizontal space during the stacking procedures. The SL_5 algorithm was shown to result in the lowest mean rank in its class for the strip packing problem, with results that are significantly better than the original $FC_{OG}DH$ algorithm, but not significantly better than the $FC_{OG}DHDW$ algorithm. However, it required significantly less computation time than the FC_{OG} algorithms to find solutions to large problems, and was only slower than the two SAS algorithms in this class.

Contribution 9 *The SC and SCR algorithms presented in §4.3.4.*

The SC algorithms were designed in an attempt to make use of the space between items packed on the floors and ceilings of levels when implementing the FC_{OF} algorithms. The attempt was successful, because the SC algorithm was shown to yield significantly better results than the FC_{OF} algorithms for the strip packing problem. The SC algorithm also required approximately half the time of the FC_{OF} algorithms to find solutions to the largest strip packing problem instances. However, the SCR algorithm was not as successful, yielding results significantly better than the $FC_{OF}DHIW$ algorithm, but significantly worse than the $FC_{OF}DHDW$ algorithm. The computation time it required to find solutions to the largest benchmark problem instances was significantly more than the corresponding computation times of the FC_{OF} algorithms. The SC algorithm was shown not to be significantly better than the $FC_{OF}DHDW$ algorithm for the MBSBPP and the SBSBPP, with the mean rank of the $FC_{OF}DHDW$ algorithm being better for both problems. The SCR algorithm appears to be significantly worse than the $FC_{OF}DHDW$ algorithm for the MBSBPP when using the fitness score to determine the ranks. However, the SC and SCR algorithms were shown to require less computation time than the $FC_{OF}DHDW$ algorithm for large benchmark instances of both problems.

Contribution 10 *The SPmG and SPmF modifications to the SP algorithm by Golan [62], as described in §5.1.4.*

The SP algorithm wastes some space when an item does not fit adjacent to the previous item in a region, because the height of the closed space is raised to the bottom of the newly-packed item. The SPmG algorithm attempts to improve the utilisation of space by packing items into the space that would remain empty in the original algorithm, while still guaranteeing a guillotine feasible layout. The SPmF algorithm does the same, but attempts to drop any packed item as low as possible in an attempt to make further use of space that may be wasted. The SPmF algorithm yields significantly better results than Sleator's algorithm and the UD algorithm, but is not significantly better than the M algorithm (see Table 6.10). The new algorithm is very slow due to the search for lower packing positions and hence would not be desirable for use when other algorithms find better solutions in less time. The SPmG algorithm is significantly better than the SF and original SP algorithms in terms of strip packing heights achieved, but this improved performance is gained at a cost to the solution time.

Contribution 11 *The BFmLM, BFmTN and BFmSN modifications to the BF algorithm by Burke et al. [22], as described in the* Algorithmic Variations *section of §5.1.9.*

These algorithms were designed with the aim of improving on the oriented versions of the BFLM, BFTN and BFSN algorithms. The original algorithms sort items according to decreasing width (resolving ties by sorting the items according to decreasing height) and pack the first item in the list into the lowest skyline segment in a best-fit manner. The modifications, on the other hand, allow the first item that fits onto a skyline segment to be packed, thereby sacrificing the best-fit manner of packing, but gaining the ability to pack items in any order. The changes made yield successful results when comparing the modified algorithms with the original algorithms for oriented problems. The best BFmLM algorithm yields a median (upper quartile) packing height of 109.3% (114.0%), compared to the median (upper quartile) value of 113.5% (123.9%) for the oriented original (see Table 6.15). The results for the BFTN and BFSN algorithms were similar.

Contribution 12 *The identification of sorting-dependent and sorting-independent algorithms, as described in §5.2.*

Sorting-dependent algorithms are very rigid in their approach to packing problems. If the items are not sorted in a particular manner, the algorithm is either not able to pack all items, or the resulting packing is poor. Sorting-independent algorithms may yield good results for items that are randomly sorted. These algorithms are useful when comparing various sorting methods and would be the family of algorithms of choice for decoders of metaheuristics [75]. They may also prove useful when provided with multiple random sortings of a set of items until a timeout is reached (in a manner similar to that of MacLeod *et al.* [109]) and saving the best solution found.

Contribution 13 *Two new sorting methods: xWDWDH and xRDWDH, as described in §5.2.2.*

These are two new sorting methods designed to eliminate the weaknesses of sorting according to decreasing height (short and wide items packed last may waste some space) or decreasing width (the last item may be thin, but very tall, thereby wasting unnecessary space). The xWDWDH sorting method sorts the items that are wider than a fraction x of the strip width according to decreasing width (resolving ties by sorting them according to decreasing height), and the remaining items according to decreasing height (resolving ties by sorting them according to decreasing width). The xRDWDH sorting method sorts the items according to decreasing width — a fraction x of the widest items remain in that order, while the remaining items are sorted according to decreasing height. The xWDWDH sorting method proved to be successful for the strip packing problem. Its use yields the best results in terms of packing height for the BL, BLF, GCS, BFmLM and BFmTN algorithms.

Contribution 14 *The number and nature of algorithmic comparisons made for the strip packing problem in Chapter 6.*

A total of 252 algorithms or variations thereof (24 distinct algorithms) were compared to one another with respect to a total of 1 170 benchmark instances for the strip packing problem. This is, to the author's knowledge, the largest comparison of heuristics for the 2D oriented offline strip packing problem (previously 27 offline heuristics had been compared for 542 benchmark instances [125]). It appears that this dissertation may be one of few examples where the Friedman test was used to test for significance (other examples include those by Parreño *et al.* [133, 134]), and one of the first applications of the Nemenyi test as a post-hoc test for significant differences between packing algorithms. It was found that the BFmTN$\frac{1}{3}$WDWDH algorithm yields the

best results in terms of packing height. If the minimum packing height achieved by the 252 algorithms is taken over all benchmark instances, then the first quartile is located at 102.5%, the median at 104.6%, the third quartile at 107.0% (the IQR is 4.5%) and the maximum at 125.8%.

The results in Table 9.1 and Table 9.2 show that metaheuristics, or algorithms that allow rotations, will typically yield better results than heuristics, but heuristics find solutions very quickly. The BF algorithm [23] allows items to be rotated and is hence expected to pack items more densely than the new algorithms. The other algorithms are metaheuristics and provide better solutions than the heuristics, with the disadvantage of requiring more computational time. The results by Cui *et al.* [37] are not exact, because they introduced relaxations in order to find faster solutions. Burke *et al.* timed their algorithm on a PC with an 850 MHz CPU and 128 MB RAM [22, p. 664], Bortfeldt [18, p. 829] timed his algorithms on a PC with a 2 GHz CPU, Alvarez-Valdes *et al.* [4, p. 1075] used a 2 GHz Pentium 4 Mobile CPU, Belov *et al.* [12, p. 829] used Linux workstations with 2×2.4 GHz CPUs and 4 GB RAM, and Cui *et al.* [37, p. 1287] used a 2.8 GHz Pentium 4 CPU and 512 MB RAM.

Contribution 15 *The 2SMBSBP algorithm presented in §7.2.*

While heuristics exist for the 1D MBSBPP and for the 2D SBSBPP, the author was unable to find examples of heuristics for the 2D MBSBPP. Therefore, the 2SMBSBP algorithm was proposed in order to solve the 2D MBSBPP quickly. It was designed in a manner that any level or pseudolevel algorithm may be incorporated into it without modification. A modified version of the BFmTN algorithm was designed to incorporate the mechanism of the 2SMBSBP algorithm in order to solve the 2D MBSBPP, but there was no packing of levels, because of the plane-packing nature of the BFmTN algorithm.

Contribution 16 *The benchmark generator for the MBSBPP presented in §8.1.2.*

There are many benchmark instances for the strip packing problem (see Table 6.1), but very few benchmark instances for the MBSBPP (see §8.1). Hence, a new set of benchmark instances were generated for the problem in a manner similar to that of Wang and Valenzuela [156], who generated a popular set of benchmark instances for the strip packing problem. There are 340 instances in total, half of which are "nice" items and the other half are "pathological" items.

Contribution 17 *The comparisons in Chapter 8 of the algorithms when combined with the 2SMBSBP algorithm to solve the 2D MBSBPP and the 2D SBSBPP.*

It was found that the FC algorithms yield the best mean ranks of the algorithms that pack items into levels, but these algorithms were not significantly better than most of the new pseudolevel-packing algorithms according to the Nemenyi test. As expected, the plane-packing algorithm yields the best results. The multiple bin size bin packing score was used to show that while the BFmTN(DA) algorithm would be the algorithm of choice for problems in which a guillotine layout is not required, the speed of the SL_5 algorithm makes it attractive as the algorithm of choice when the guillotine constraint applies. A comparison of the FC_{OG}DHDW and BFmTN(DA) algorithms when combined with the 2SMBSBP algorithm to some of the algorithms from the literature in Table 9.3 show that the more specialised algorithms yield better solutions. However, the difference in mean solution times is large, and some of the speed differences are attributable to the difference in computers used. The results in this dissertation

	SL_5	BFmTN($\frac{1}{3}$W)	Min	BF	SPGAL		GRASP	SVC	BS	HRBB
Source	New	New	New	[23]	[18]	[18]	[4]	[12]	[12]	[37]
Alg T	H	H	H	H	MH	MH	MH	MH	MH	E
Packing	PL	P	P	P	P	P	P	P	P	P
Problem	OG	OF	OF	RF	OG	OF	OF	OF	OF	RG
C1	15.0	6.7	3.3	11.7	3.2	1.6	0.0	1.7	1.7	1.7
C2	8.9	6.7	4.4	13.3	3.3	3.3	0.0	0.0	0.0	0
C3	14.4	8.9	5.6	10.0	3.9	3.2	1.1	1.1	1.1	1.1
C4	10.6	5.6	3.9	5.0	3.8	3.5	1.6	1.7	1.7	2.2
C5	8.1	5.2	3.0	4.1	2.4	2.0	1.1	1.1	1.1	1.9
C6	7.5	4.2	2.5	3.3	1.9	1.7	1.6	0.8	1.4	1.4
C7	5.1	3.9	2.2	2.4	1.7	1.5	1.4	0.8	1.1	1.4
Mean t	0.593 ms	0.983 ms	4.28 s	±0.01 s	143 s	159 s	60 s	50 s	50 s	1.86 s
T1	36.8	25.2	12.8	—	—	—	0.0	0.9	1.0	—
T2	26.2	13.2	8.3	—	—	—	3.2	3.5	4.0	—
T3	21.0	10.2	8.3	—	—	—	3.7	3.3	4.3	—
T4	11.8	12.8	6.8	—	—	—	3.0	2.5	3.2	—
T5	11.4	8.4	6.0	—	—	—	2.4	2.1	2.7	—
T6	8.6	8.0	4.1	—	—	—	2.1	1.6	2.1	—
T7	5.9	5.6	2.2	—	—	—	1.5	1.0	1.4	—
Mean t	0.596 ms	1.035 ms	4.69 s	—	—	—	60 s	50 s	50 s	—
N1	23.9	15.8	9.3	—	—	—	0.9	3.3	4.4	—
N2	18.7	12.7	8.4	—	—	—	3.3	3.4	4.2	—
N3	20.3	11.6	8.4	—	—	—	3.6	3.5	4.2	—
N4	13.5	12.2	6.6	—	—	—	3.0	2.5	3.1	—
N5	10.1	8.0	6.1	—	—	—	2.6	2.1	2.7	—
N6	7.8	7.2	4.3	—	—	—	2.2	1.7	2.2	—
N7	5.3	5.5	2.4	—	—	—	1.3	1.0	1.0	—
Mean t	0.598 ms	1.052 ms	4.35 s	—	—	—	60 s	50 s	50 s	—

Table 9.1: *Comparison of the results by the best heuristics in this dissertation with other algorithms from the literature for the benchmark instances by Hopper and Turton [75, 79, 80]. The* Min *column is the minimum packing height achieved by the 252 algorithms for each instance and its mean times are the sum of the mean packing times achieved by the 252 algorithms. The* Alg T *row indicates whether the algorithm is a heuristic (H), metaheuristic (MH) or an exact (E) method. The* Packing *row indicates which algorithms result in pseudolevel (PL) or plane (P) solutions. The* Problem *row indicates the problem type, as defined by Lodi et al. [105] and described in §2.1.3. The HRBB algorithm does not find an optimal solution even though it is labelled an exact method, because Cui et al. [37] introduced relaxations.*

were obtained on a Windows XP PC with a 3.0 GHz Intel Core 2 Duo CPU and 4 GB RAM, while Boschetti and Mingozzi used a Pentium III 933 MHz PC [19, p. 146] and El Hayek *et al.* used a Pentium 4 2.66 GHz PC [67, p. 3195].

9.3 An Appraisal of the Dissertation Contributions

A large number of novel algorithms, and variations on these novel algorithms as well as on known algorithms were proposed in this dissertation in an attempt to further the knowledge in the field of solution heuristics for the 2D oriented, offline, regular strip packing problems. Attempts at finding a level-packing heuristic that consistently outperforms the BFDHDW algorithm were unsuccessful. The best of the novel level-packing heuristics, the $B2FA_{10}DHDW$ algorithm, yields results that are equivalent to the BFDHDW algorithm in terms of achieved packing

	SL_5	BfmTN($\frac{1}{3}$W)	Min	BF	BF+SA	GRASP
Source	New	New	New	[23]	[23]	[4]
Alg T	H	H	H	H	MH	MH
Packing	PL	P	P	P	P	P
Problem	OG	OF	OF	RF	RF	OF
Nice25	21.0	14.5	10.2	8.0	4.0	3.7
Nice50	16.3	12.3	9.1	9.7	4.4	4.6
Nice100	11.1	10.0	7.7	7.9	5.0	4.0
Nice200	8.6	8.4	6.3	6.9	4.7	3.6
Nice500	5.8	6.5	4.7	3.4	3.5	2.2
Nice1t	4.3	5.3	3.5	3.8	2.9	2.2
Nice2t	3.0	4.0	2.6	—	—	—
Nice5t	2.0	3.6	1.8	—	—	—
Mean t	0.111 s	0.230 s	2 202 s	—	—	60 s
Path25	29.1	18.5	10.1	10.2	3.1	4.2
Path50	25.1	12.9	6.7	13.7	3.4	1.8
Path100	17.7	10.2	5.1	6.8	3.0	2.6
Path200	10.9	8.4	4.5	4.1	3.4	2.0
Path500	6.3	7.1	4.5	3.8	3.5	3.1
Path1t	5.1	6.3	3.8	3.1	2.9	2.5
Path2t	3.7	4.3	3.2	—	—	—
Path5t	2.8	3.8	2.5	—	—	—
Mean t	0.113 s	0.229 s	1 437 s	—	—	60 s

Table 9.2: *Comparison of the results by the best heuristics in this dissertation with other algorithms from the literature for the instances by Wang and Valenzuela [156]. The* Min *column is the minimum packing height achieved by the 252 algorithms for each instance and its mean times are the sum of the mean packing times achieved by the 252 algorithms. The* Alg T *row indicates whether the algorithm is a heuristic (H) or metaheuristic (MH) method. The* Packing *row indicates which algorithms result in pseudolevel (PL) or plane (P) solutions. The* Problem *row indicates the problem type, as defined by Lodi et al. [105] and described in §2.1.3.*

	FC_{OG}DHDW	BFmTN(DA)	HBP		IMA
Source	New/ [101, 105]	New	[19]		[67]
Alg T	H	H	H		H
Packing	PL	P	P		P
Problem	OG	OF	OF	RF	RF
BW 1	20.60	20.20	19.46	19.46	19.46
BW 2	2.60	2.56	2.48	2.48	2.48
BW 3	14.72	14.54	14.06	13.64	13.64
BW 4	2.56	2.52	2.50	2.48	2.44
BW 5	18.70	18.34	17.98	17.38	17.30
BW 6	2.36	2.32	2.26	2.24	2.20
MV 7	17.10	16.88	16.64	15.70	15.36
MV 8	17.50	17.22	16.78	15.76	15.44
MV 9	42.78	42.74	42.60	42.38	42.38
MV 10	10.52	10.42	10.22	10.04	10.02
Mean t	4.292 ms	1.679 ms	2.102 s	1.824 s	0.319 s

Table 9.3: *A comparison of the new algorithms with algorithms from the literature for the SBSBPP. The* Alg T *row indicates that the algorithms are all heuristics (H). The* Packing *row indicates which algorithms result in pseudolevel (PL) or plane (P) solutions. The* Problem *row indicates the problem type, as defined by Lodi et al. [105] and described in §2.1.3.*

heights and solution times. The WFDH family of algorithms were shown to yield results that had a distribution shifted further from the optimal packing heights than the results of the BFDH and FFDH families of algorithms.

The search for pseudolevel-packing algorithms proved more successful. For the problem where a guillotine feasible layout was required, two sorting modifications were proposed for the FC_{OG} and BFDH* algorithms and the DHDW variations proved to yield consistently better mean ranks of packing heights than the original algorithms that do not resolve ties in height during the sorting procedure. The modifications proposed for the SAS algorithm resulted in an algorithm that yields significantly better results with respect to packing height, and yielded results for large benchmark problem instances consisting of "nice" items in significantly less time. The newly proposed BFS algorithm was able to find significantly better solutions than the BFDH* algorithms, but these results were significantly worse than those of the FC_{OG}DHDW algorithm. However, the BFS algorithm was shown to be significantly faster than the FC_{OG} and BFDH* algorithms. The SL algorithm was shown to yield results significantly better than all other algorithms in the set, excluding the FC_{OG}DHDW algorithm, and in a time that was significantly faster than that of the FC_{OG} and BFDH* algorithms. The search for a better pseudolevel-packing heuristic that does not guarantee a guillotine layout proved successful. The application of the novel SC algorithm to the 1 170 benchmark problem instances yields significantly better results than the FC_{OF} algorithms with respect to both packing height and execution time for large problem instances.

The search for improved plane-packing heuristics also proved successful. The proposed modifications to the SP algorithm yield results that are significantly better than the original algorithm, but at the cost of additional solution time (the free-packing version is prohibitively slow). The new xWDWDH and xRDWDH sorting methods were shown to yield significantly better results with respect to packing heights than the original DW (respectively DA) sorting methods for the BL and BLF (respectively GCS) algorithms. The proposed modifications to the BFLM, BFTN and BFSN algorithms also proved to yield better solutions than the original algorithms when combined with the new sorting methods, or when combined with a sorting method that sorted according to decreasing area. In fact, the BFmTN$\frac{1}{3}$WDWDH algorithm was shown to yield the best results with respect to packing height of all algorithms in this dissertation, better even than the BLF$\frac{2}{5}$WDWDH algorithm, which was shown to be the best of the BLF algorithms, one of the "most documented heuristic approaches" for this problem [22, p. 656]. Table 9.4 is a summary of the best guillotine and free-packing algorithms for the strip packing problem with respect to packing height and solution time.

Priority	Guillotine	Free
Speed	SASm	SASm
Speed & Packing Height	SL_5	BFmTN(DADW)
Packing Height	GCS$\frac{1}{2}$WDWDH	BFmTN$\frac{1}{3}$WDWDH

Table 9.4: *Algorithms recommended for the strip packing problem.*

The 2SMBSBP algorithm, designed to be combined with the level and pseudolevel-packing algorithms in order to find fast feasible solutions to the MBSBPP, appears to be the first heuristic for the 2D MBSBPP. The level and pseudolevel strip packing algorithms were successfully combined with the 2SMBSBP algorithm to yield good solutions to the MBSBPP and the SBSBPP. It was shown that the FC_{OG}DHDW algorithm yields the best guillotine solutions and a modification to the BFmTN algorithm yields the best (and fastest) solutions to these bin packing problems. The novel pseudolevel-packing algorithms yield solutions that are not significantly

different according to the Nemenyi test (yet the FC algorithms yield a better mean rank for both bin packing problems), but they are faster. Table 9.5 is a summary of the best guillotine and free-packing algorithms for the MBSBPP with respect to packing density and solution time. It is hoped that this approach to the MBSBPP may prove useful to researchers requiring initial feasible solutions or bounds for other, more accurate solution methods for this problem, including metaheuristics.

Priority	Guillotine	Free
Speed	SASm	BFmTN(DA)
Speed & Packing Density	SL_5	BFmTN(DA)
Packing Density	FC_{OG}DHDW	BFmTN(DA)

Table 9.5: *Algorithms recommended to be combined with the 2SMBSBP algorithm.*

Many researchers have compared their 2D strip or bin packing algorithms to known algorithms by means of benchmark problem instances in the past, but this has typically been done on an instance-by-instance manner, or by means of averages of related instances. If one algorithm resulted in lower packing heights than another algorithm for these individual instances for the strip packing problem, or fewer bins for the SBSBPP, then the former was declared a better algorithm. However, this approach is limited, because the number of comparisons that can be made in this manner is small. If one wanted to perform a large-scale comparison of algorithms, this method of presenting the results becomes impractical. It would require many lists of results and it would be very difficult to reach a conclusion, or see a pattern, from the large amount of data.

In an attempt to determine which algorithms performed well, algorithmic results were presented in this dissertation by means of box plots and in terms of quartiles. One would reasonably expect strip packing algorithms resulting in low quartile values to typically yield better results than algorithms with higher quartile values. In an attempt to find further clarity when comparing items, the algorithms were ranked relative to each other for each benchmark instance. One may reasonably expect that the algorithm yielding the best mean rank over the set of benchmark instances would typically result in better packings than the algorithms with worse mean ranks. However, in order to ascertain whether these rank differences were significant, the Friedman and Nemenyi tests, as described by Demšar [40], were employed. Using these various methods of comparison allowed for the testing of a large number of algorithms over a large number of benchmark instances at a 95% level of significance. These comparisons should prove useful to researchers in search of efficient algorithms for the strip and MBSBP problems.

These tests were performed on a large number of novel algorithms, many of which outperformed equivalent algorithms in the literature in terms of speed and/or packing height. Many of the new algorithms did not perform better than the known algorithms and it is hoped that these investigations may serve as a discouragement to other researchers with respect to following the same routes of investigation in an attempt to find good heuristics. An article [130] published on some of these strip packing algorithms and the 2SMBSBP algorithm, as well as papers presented at local [126, 128, 131] and international [129] conferences, yielded positive feedback from peers, and the independent, anonymous reviewers of the article.

CHAPTER 10

Possible Future Work

During the process of compiling this dissertation, the author became aware of some possibilities for further work that may be done in the field of packing problem heuristics. This chapter contains brief descriptions of some of these possibilities for further research. First, a modification to the GCS algorithm is proposed, and this is followed by a new packing strategy. Then a combination of the BFmTN algorithm with a metaheuristic is proposed, followed by a proposal for the use of pricing strategies for the MBSBPP. Finally, a brief description is given of possible changes to the novel algorithms presented in this dissertation for problems that allow item rotations.

Proposal 1 *Reducing the GCS algorithm to a pseudolevel algorithm.*

The GCS algorithm described in §5.1.8 proved to be slow in comparison to many of the other algorithms, as was shown in Tables 6.14 and 6.18. This may be due to the author's lack of programming prowess, or because the number of cuts that have to be stored and tested for guillotine feasibility, and the number of locations at which items may be packed due to existing cuts, result in a large number of calculations that must be carried out in order to find the best location for an item. In an attempt to reduce the complexity of the problem, the GCS packing algorithm may be converted to a pseudolevel algorithm. The horizontal lines representing level separators are minimal cuts, separating the cuts and items in the levels from each other, as if the items in the levels were in separate bins. The algorithm would remain sorting-independent, which is likely to give it an advantage over the BFS, SL and SC algorithms.

The newly proposed, converted algorithm may begin by finding the tallest item in the set and packing it onto the floor of the strip. A dummy item of zero height and a width equal to the strip width should be placed onto this item, creating a rectangular area into which items may be packed. The only exposed border is the left-hand edge of the item that initialised the level; hence an attempt should be made to pack the first item in the list against this edge in a bottom-left manner. If enough space were to remain for the item, it is placed there, otherwise the next item in the list should be investigated for packing. There may now be two exposed borders adjacent to which items may be located. This process should continue until an attempt has been made to pack all items into the level. If any items were to remain unpacked, a new level should be initialised by the tallest remaining unpacked item. This level should then be filled according to the method described, and the process should continue until all items are packed.

For example, consider using the GCS_{PL}(DWDH) algorithm to pack the items listed in Table 3.1. When using the merge-sort algorithm to sort the set of items according to decreasing width, and resolving any ties by sorting them according to decreasing height, the list $\mathcal{I} = \{\mathcal{I}_{13}, \mathcal{I}_{11}, \mathcal{I}_5, \mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_6, \mathcal{I}_4, \mathcal{I}_9, \mathcal{I}_{12}, \mathcal{I}_7, \mathcal{I}_1, \mathcal{I}_8, \mathcal{I}_2\}$ results. The tallest item in the set is $\mathcal{I}_1$ and it initialises the first level. Its right-hand edge is the only exposed border, and the space between it and the right-hand boundary of the strip is wide enough for the first item in the list, namely $\mathcal{I}_{13}$. The top edge of $\mathcal{I}_{13}$ is the only exposed border and items $\mathcal{I}_{11}$, $\mathcal{I}_5$ and $\mathcal{I}_{10}$ are too large to fit into the space between the top edge of $\mathcal{I}_{13}$ and the ceiling of the level. The first item that fits into this space is $\mathcal{I}_3$ and it fills the space to the ceiling of the level, rendering its right-hand edge the only exposed border. Item $\mathcal{I}_6$ follows in the list and it fits adjacent to $\mathcal{I}_3$, creating two exposed borders: the right-hand and top edges of $\mathcal{I}_6$. The right-hand edge is likely to yield a lower packing than the top edge, and is investigated first as a packing location for an item. The only item that fits into this space width-wise is $\mathcal{I}_2$, but it is too tall; its height combined with that of $\mathcal{I}_{13}$ is larger than the height of $\mathcal{I}_1$ (and hence the level). Item $\mathcal{I}_4$ follows $\mathcal{I}_6$ in the list and may be packed onto it. The item that follows, $\mathcal{I}_9$, may be packed onto $\mathcal{I}_4$, as the right-hand edge of $\mathcal{I}_4$ is at the same horizontal coordinate as the right-hand edge of $\mathcal{I}_6$, rendering it an unsuitable exposed border for any unpacked items. The only remaining exposed border is the right-hand edge of $\mathcal{I}_9$ and the space is too small for any of the remaining items to be packed.

Of the remaining items, $\mathcal{I}_{11}$ is the tallest and it initialises the second level. The only exposed border is the right-hand edge of $\mathcal{I}_{11}$, and $\mathcal{I}_{12}$ is the first item in the list that fits into the remaining space in the level. The top edge of $\mathcal{I}_{12}$ is now the only exposed border because its right-hand edge is adjacent to the right-hand boundary of the strip. Item $\mathcal{I}_7$ follows in the list and may be packed onto $\mathcal{I}_{12}$. Two exposed borders are established due to this packing, namely the top and right-hand edges of $\mathcal{I}_7$. None of the items are narrow enough to fit between the right-hand edge of $\mathcal{I}_7$ and the boundary of the strip; only $\mathcal{I}_8$ fits onto $\mathcal{I}_7$. The space between the right-hand edge of $\mathcal{I}_8$ (the only remaining exposed border) and the boundary of the strip is wide enough to accommodate $\mathcal{I}_2$, but the space is not tall enough, resulting in the closing of the level.

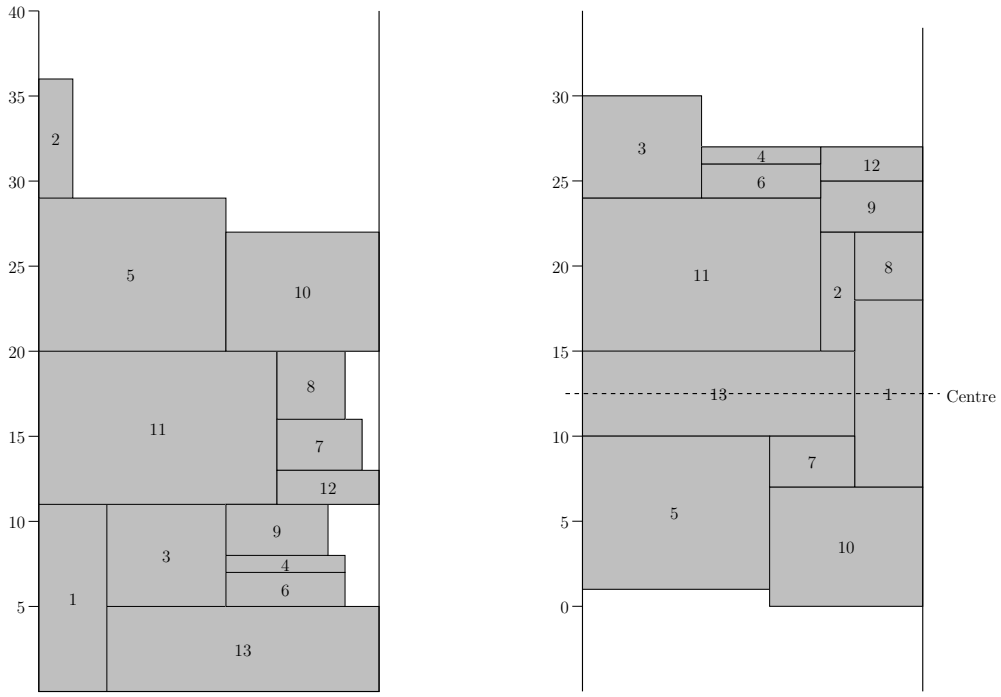
Item $\mathcal{I}_5$ is the tallest remaining item and it initialises the third level. Item $\mathcal{I}_{10}$ fits into the space to the right of $\mathcal{I}_5$, leaving only its top edge as an exposed border. The remaining space is too small for $\mathcal{I}_2$ and it initialises a fourth level. The resulting strip height is 36 and the packing is shown in Figure 10.1(a).

Proposal 2 *Design of a double-sided (DS) strip packing algorithm.*

The newly proposed algorithm should ignore the location of the bottom of the strip and pack the first item adjacent to the left-hand side boundary. The remaining items should be packed to the right of the item, or above it, or below it in such a manner that the strip height increases by the smallest distance for each item packed. If the packing of an item does not change the height of the strip overall, then packing from the top is preferable.

For example, consider combining the BFmLM(DWDH) heuristic with the DS approach to pack the items listed in Table 3.1. When using the merge-sort algorithm to sort the set of items according to decreasing width, and resolving any ties by sorting them according to decreasing height, the list $\mathcal{I} = \{\mathcal{I}_{13}, \mathcal{I}_{11}, \mathcal{I}_5, \mathcal{I}_{10}, \mathcal{I}_3, \mathcal{I}_6, \mathcal{I}_4, \mathcal{I}_9, \mathcal{I}_{12}, \mathcal{I}_7, \mathcal{I}_1, \mathcal{I}_8, \mathcal{I}_2\}$ results. First, $\mathcal{I}_{13}$ is packed adjacent to the left-hand boundary of the strip. The space remaining between $\mathcal{I}_{13}$ and the right-hand boundary of the strip is wide enough to accommodate $\mathcal{I}_1$, which is packed in such a manner that it protrudes on either side of $\mathcal{I}_{13}$ by the same distance.[1] No further items

[1] If the item dimensions are integer and the protrusions (which may have a negative length if the second item is shorter than the first) are not of integer length, then the item may be moved up or down by the distance



(a) $\mathrm{GCS}_{\mathrm{PL}}$ (H = 36)

(b) DS (H = 30)

Figure 10.1: *Results obtained when packing the items in Table 3.1 into a strip of width 20 using the newly proposed algorithms described in Proposals 1 and 2. The resulting packing heights* H *are also shown.*

fit between those items that initialised the strip and the right-hand boundary; hence two sets of skyline segments are established, one describing the top boundary and another along the bottom of the packed items.

The skyline segments along the top and bottom of $\mathcal{I}_{13}$ both have the same distance from the centre of $\mathcal{I}_{13}$, thus the top skyline segment is selected for further packing. The first unpacked item in the sorted list that fits into this skyline segment is $\mathcal{I}_{11}$ and it is packed there, yielding three skyline segments along the top of the packed items. The two skyline segments on either side nearest the centre have the same distance from the centre; the top one is wide enough to accommodate $\mathcal{I}_2$ and the bottom one is wide enough to accommodate $\mathcal{I}_5$. Item $\mathcal{I}_2$ is packed onto the top skyline segment because it does not contribute to the length of the strip (the top edge of $\mathcal{I}_{11}$ is further from the centre than the top edge of $\mathcal{I}_2$). The lowest skyline segment along the top is above $\mathcal{I}_1$, while the best skyline segment along the bottom is below $\mathcal{I}_{13}$. The top segment may accommodate $\mathcal{I}_8$, which, when packed, does not contribute to the height of the strip, while packing $\mathcal{I}_5$ along the bottom would contribute more to the strip height. Therefore, $\mathcal{I}_8$ is packed above $\mathcal{I}_1$ and the lowest skyline segment at the top becomes the conjoined top edges of $\mathcal{I}_2$ and $\mathcal{I}_8$. This is wide enough to accommodate $\mathcal{I}_9$, which is packed there due to its small contribution to the strip height.

The best skyline segments on either side of the strip are wide enough to accommodate $\mathcal{I}_5$ (the first item in the list) and it is packed along the bottom because it contributes a length of 6 to the strip height when packed there, as opposed to a height of 8 when packed on the top. The best top segment is wide enough for $\mathcal{I}_{10}$, which would contribute a length of 6 if packed there, while

necessary to render the protrusion lengths integer values.

the best skyline segment on the bottom may accommodate $\mathcal{I}_7$, which does not contribute to the current strip height. Its location there yields a skyline segment wide enough to accommodate $\mathcal{I}_{10}$ and its location there would result in a smaller increase in strip height than if it were packed along the top (a distance of 1 compared to 6 if it were packed along the top).

If $\mathcal{I}_3$ were packed onto either of the best skylines on either side of the centre, it would result in the same increase in strip height. Therefore, it is packed onto the top segment. The resulting best skyline segment above the centre has the same width as $\mathcal{I}_6$, which does not add to the strip height when packed there. For similar reasons, $\mathcal{I}_{12}$ is packed onto $\mathcal{I}_9$ and $\mathcal{I}_4$ is packed onto $\mathcal{I}_6$, yielding the solution shown in Figure 10.1(b).

It may be possible to further improve the algorithm if one were to implement a form of best-fit strategy when packing the items. For example, consider the packing in Figure 10.1(b) as an intermediate packing, and suppose that the next item has a height of 1 and a length of no more than the width of $\mathcal{I}_5$. It may be packed on either side of the centre and the algorithm, as it is currently described, would pack it onto the best skyline above the centre. However, it would be more desirable to pack the item below $\mathcal{I}_5$ because the difference in distances from the centre to the nearest and furthest skyline segments for the bottom skyline is smaller than for the top skyline. It is a vertical equivalent of the minimum residual horizontal space that is made use of in the best-fit algorithms.

Proposal 3 *Use a metaheuristic to order the items before they are packed into a strip (or into multiple-size bins) by the BFmTN (or the 2SMBSBP-BFmTN) algorithm.*

There are many examples of hybrid algorithms where a metaheuristic is used to order sets of items (often called *permutations*) and a heuristic is used to *decode* these permutations (for example, see Hopper and Turton [75–79]). The algorithm may begin by sorting the list of items according to various methods; such as the DA, DH, DW, $\frac{1}{2}$WDWDH and $\frac{1}{2}$RDWDH sorting methods. These lists of items may be used as the initial population of permutations for a genetic algorithm. The new BFmLM, BFmTN or BFmSN algorithm may be used to pack these lists of items and determine their quality in terms of packing height. These lists of items may then be crossed where, for example, a number $0 < x < n$ of the better of two lists forms the first part of the new list, and the remaining $n - x$ items are inserted into the new list in the same order in which they appear in the second list. A mutation, with a small probability of occurring, may then swap the location of two randomly-selected items. The BFm algorithms will then continue to decode the ordered lists of items created by the genetic algorithm. The best solution is saved and replaced when a better solution is found. The algorithm may be stopped when a time limit is reached.

This method may also be used to sort the items for packing by the 2SMBSBP-BFmTN algorithm, for example, or the genetic algorithm may determine the bins into which the items should be packed, while the BFmTN algorithm may decode the item/bin pairs to determine whether the items all fit into their assigned bins. If items are assigned to a bin, but do not all fit into the bin, the items could either be repacked, or selected items may be repacked into bins that have sufficient space (these bins may contain other items). In this manner feasible solutions may be guaranteed.

One possible drawback of the BFm family of algorithms, when using them in combination with a metaheuristic, is the fact that they are not guaranteed to pack items in the order in which they are found in the list passed to the algorithm. If the lowest skyline segment is short, then many large or wide items may be overlooked for packing, while a smaller item is packed onto the segment because it fits there. This may interfere with the order of items supplied by the

metaheuristic. Heuristics such as the BL, BLF and GCS algorithms pack items in the order in which they appear in the list. This may give them an advantage over the BFmLM, BFmTN and BFmSN algorithms when used in this manner.

Proposal 4 *Use the pricing scheme by Boschetti and Mingozzi [19] to solve the MBSBPP.*

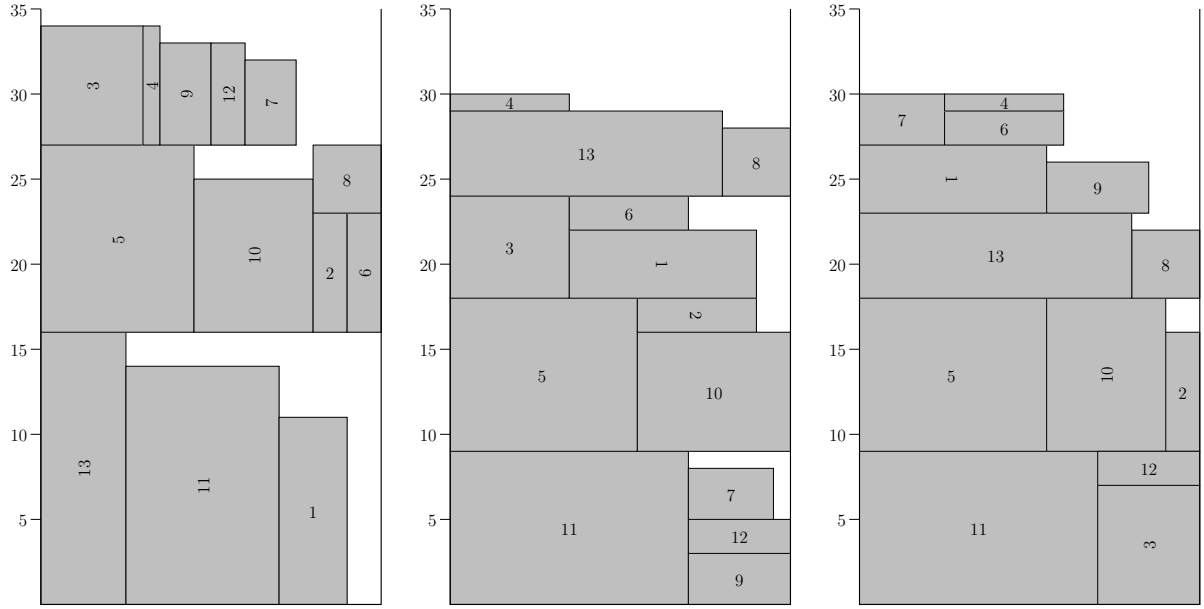
This may be done in one of two ways. The HBP algorithm may either be modified to pack the largest bins first and then attempt to repack them in the manner of the 2SMBSBP algorithm, or one may use the strategy of changing item prices to yield different packing solutions for various iterations of the 2SMBSBP-BFmTN algorithm. For example, one could assign all the items a price equal to their height, area or width, or a weighted combination of these dimensions. The items may then be sorted according to their price and the 2SMBSBP-BFmTN algorithm may pack these items into the relevant bins. Once the packing is complete, the items that are in densely-packed bins may have their prices adjusted up/down, while those items in bins that are less densely filled may have their prices adjusted down/up. The packing may then be performed again, with the items sorted according to these new prices, saving the best solution of all the iterations and continuing until some timeout condition is reached.

Proposal 5 *Adapt the BFS, SL and SC algorithms to pack items that may be rotated.*

There are a number of options as to how rotation may be incorporated in packing heuristics. The items could either all be rotated in such a manner that their longest dimension becomes the item heights (called *vertical packing*), or they may be rotated in such a manner that their shortest dimension becomes the item heights (as proposed by Lodi *et al.* [103], and called *horizontal packing*). Lodi *et al.* also propose initialising a level with the item with the longest short dimension by packing it horizontally before packing the remaining items. If an item can be packed vertically, then it should be; otherwise it should be packed horizontally. This may be achieved with some efficiency if the items were stored in a list in which both orientations are represented, but only the horizontal orientation is selected for level initialisations and both orientations are removed from the list when an item is packed (this may be done efficiently via linked-lists). This suggestion is based somewhat on the strategy of Bengtsson [14, p. 354] to list all dimensions of the items in a single list, guaranteeing that both orientations of the item will be tested.[2] Examples of these rotation strategies, when applied to the SL algorithm (called the SL_{RG} algorithm), may be found in Figure 10.2 and, when applied to the SC algorithm (called the SC_{RF} algorithm), in Figure 10.3.

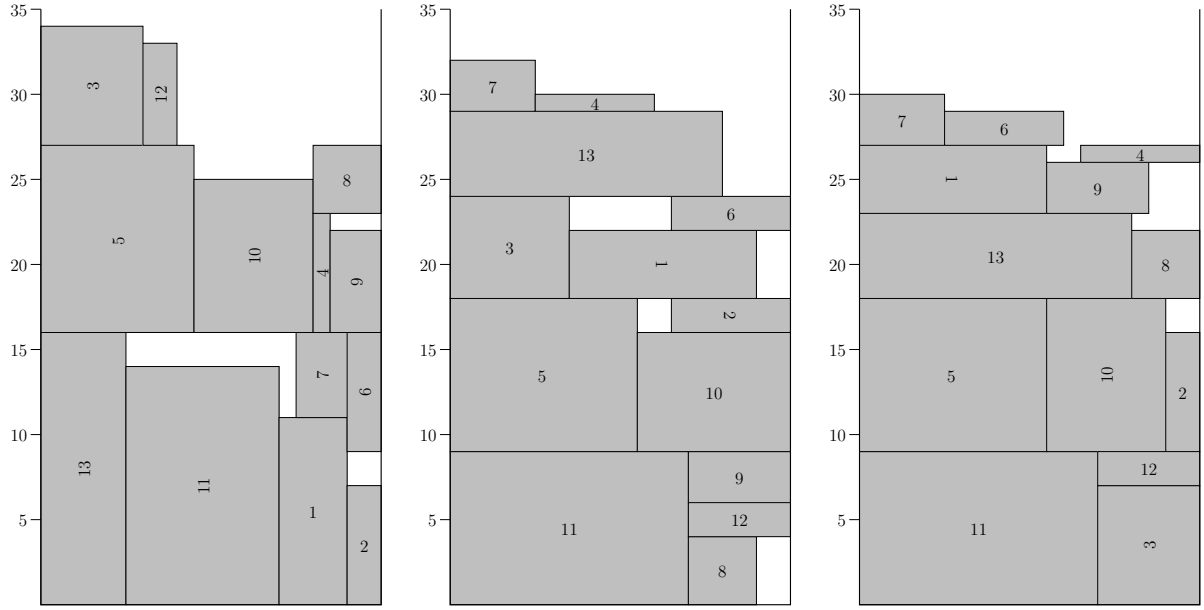
The BFmLM, BFmSN and BFmTN algorithms may also be adapted to pack items that may be rotated. In a manner similar to Bengtsson's method of sorting items, a list of items of size $2n$ may be generated, containing both orientations of each item. The items may then be sorted according to decreasing height (see Figure 10.4) or according to decreasing width, as shown in Figure 10.5, and packed into the strip, or into bins. These algorithms may benefit from the repacking strategy employed by the original BFLM, BFTN and BFSN algorithms by Burke *et al.* [22] which attempts to rotate those items that are tall (this is described in the *Algorithmic Variations* section of §5.1.9).

[2]Bengtsson [14] allowed iterative improvements to the solutions, thereby guaranteeing that both orientations would be tested. In the algorithms in this dissertation the orientation best suited to the current packing location is selected. If the item is initialising a level or packing into a space that is shorter than the item's longest dimension, then the item is packed horizontally. The item is packed vertically in other cases.



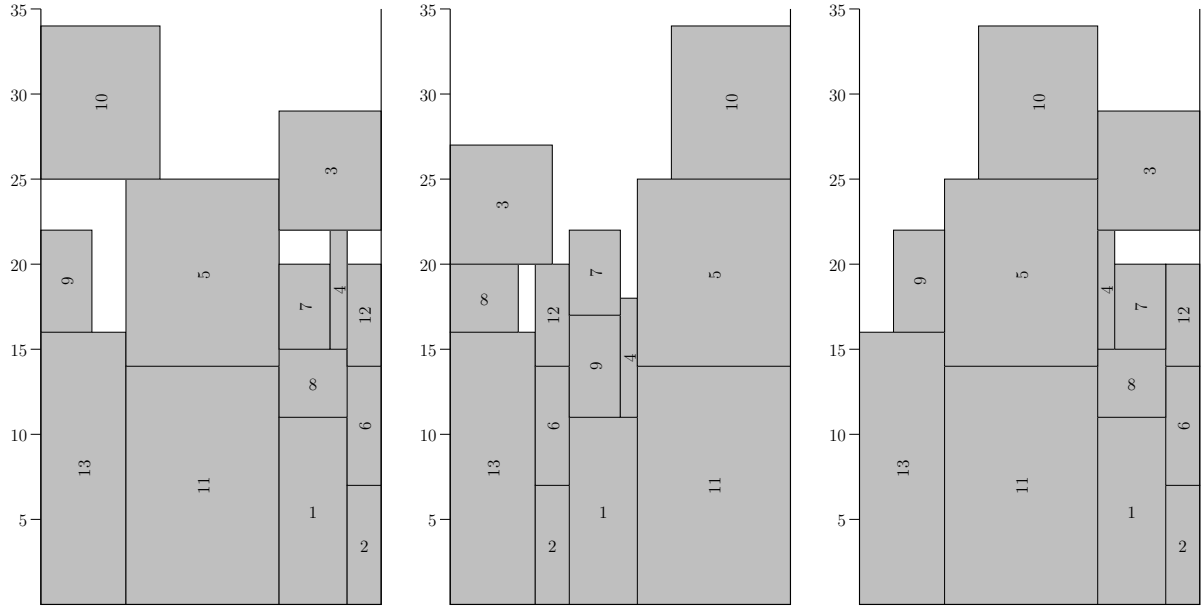
(a) SL_{RG} (vertical) (H = 34) (b) SL_{RG} (horizontal) (H = 30) (c) SL_{RG} (mixed) (H = 30)

Figure 10.2: *Results obtained when packing the items in Table 3.1 into a strip of width 20 using the newly suggested versions of the SL algorithm that allow rotation as described in Proposal 5. The resulting packing heights* H *are also shown.*



(a) SC_{RF} (vertical) (H = 34) (b) SC_{RF} (horizontal) (H = 32) (c) SC_{RF} (mixed) (H = 30)

Figure 10.3: *Results obtained when packing the items in Table 3.1 into a strip of width 20 using the newly suggested versions of the SC algorithm that allow rotation as described in Proposal 5. The resulting packing heights* H *are also shown.*

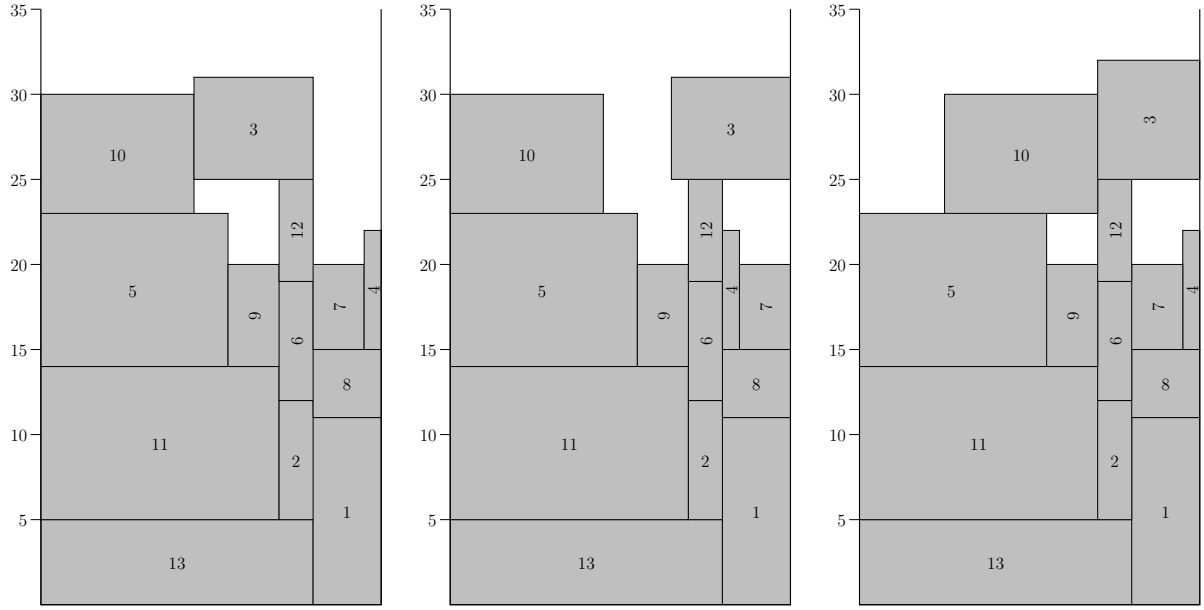


(a) BFmLM_{RF}(DH) (H = 34)

(b) BFmTN_{RF}(DH) (H = 34)

(c) BFmSN_{RF}(DH) (H = 34)

Figure 10.4: *Results obtained when packing the items in Table 3.1 into a strip of width 20 using the newly suggested versions of the BFm_{RF}(DH) algorithms that allow rotation as described in Proposal 5. The resulting packing heights* H *are also shown.*



(a) BFmLM_{RF}(DW) (H = 31)

(b) BFmTN_{RF}(DW) (H = 31)

(c) BFmSN_{RF}(DW) (H = 32)

Figure 10.5: *Results obtained when packing the items in Table 3.1 into a strip of width 20 using the newly suggested versions of the BFm_{RF}(DW) algorithms that allow rotation as described in Proposal 5. The resulting packing heights* H *are also shown.*

References

[1] Alvarez-Valdes R, Parajon A & Tamarit JM, 2002, *A computational study of LP-based heuristic algorithms for two-dimensional guillotine cutting stock problems*, OR Spectrum, **24(2)**, pp. 179–192.

[2] Alvarez-Valdes R, Parajon A & Tamarit JM, 2002, *A tabu search algorithm for large-scale guillotine (un)constrained two-dimensional cutting problems*, Computers and Operations Research, **29(7)**, pp. 925–947.

[3] Alvarez-Valdes R, Parreño F & Tamarit JM, 2007, *A tabu search algorithm for a two-dimensional non-guillotine cutting problem*, European Journal of Operational Research, **183(3)**, pp. 1167–1182.

[4] Alvarez-Valdes R, Parreño F & Tamarit JM, 2007, *Reactive GRASP for the strip-packing problem*, Computers and Operations Research, **35(4)**, pp. 1065–1083.

[5] Baker BS, Brown DJ & Katseff HP, 1981, *A 5/4 algorithm for two-dimensional packing*, Journal of Algorithms, **2(4)**, pp. 348–368.

[6] Baker BS, Coffman EG & Rivest RL, 1980, *Orthogonal packings in two dimensions*, SIAM Journal on Computing, **9(4)**, pp. 846–855.

[7] Beasley JE, 1985, *Algorithms for unconstrained two-dimensional guillotine cutting*, Journal of the Operational Research Society, **36(4)**, pp. 297–306.

[8] Beasley JE, 1985, *An exact two-dimensional non-guillotine cutting tree search procedure*, Operations Research, **33(1)**, pp. 49–64.

[9] Beasley JE, 2009, *OR-Library*, [Online], [Cited August 25th, 2009], Available from `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`

[10] Bekrar A & Kacem I, 2009, *An exact method for the 2D guillotine strip packing problem*, Advances in Operations Research, [Online Serial], [Cited October 24th, 2009], Available from `http://www.hindawi.com/journals/aor/2009/732010.html`

[11] Belov G, 2003, *Problems, models and algorithms in one- and two-dimensional cutting*, PhD Dissertation, Technische Universität Dresden, Dresden.

[12] Belov G, Scheithauer G & Mukhacheva EA, 2008, *One-dimensional heuristics adapted for two-dimensional rectangular strip packing*, Journal of the Operational Research Society, **59(6)**, pp. 823–832.

[13] Beisiegel B, Kallrath J, Kochetov Y & Rudnev A, 2006, *Simulated annealing based algorithm for the 2D bin packing problem with impurities*, pp. 309–314 in Haasis H-D, Kopfer H & Schönberger J (Eds), *Operations research proceedings 2005*, Springer, Heidelberg.

[14] Bengtsson BE, 1982, *Packing rectangular pieces — A heuristic approach*, The Computer Journal, **25(3)**, pp. 353–357.

[15] Berkelaar M, Eikland K & Notebaert P, 2004, *lp_solve: Open source (mixed integer) linear programming system*, [Online], [Cited December 10th, 2008], Available from `http://sourceforge.net/projects/lpsolve`

[16] Berkey JO & Wang PY, 1987, *Two-dimensional finite bin-packing algorithms*, Journal of the Operational Research Society, **38(5)**, pp. 423–429.

[17] Blum C & Roli A, 2003, *Metaheuristics in combinatorial optimization: Overview and conceptual comparison*, Association for Computing Machinery Computing Surveys, **35(3)**, pp. 268–308.

[18] Bortfeldt A, 2006, *A genetic algorithm for the two-dimensional strip packing problem with rectangular pieces*, European Journal of Operational Research, **172(3)**, pp. 814–837.

[19] Boschetti MA & Mingozzi A, 2003, *The two-dimensional finite bin packing problem. Part II: New lower and upper bounds*, 4OR (Quarterly Journal of the Belgian, French and Italian Operations Research Societies), **1(2)**, pp. 135–147.

[20] Brown DJ, 1980, *An improved BL lower bound*, Information Processing Letters, **11(1)**, pp. 37–39.

[21] Burke E & Kendall G, 1999, *Applying simulated annealing and the no fit polygon to the nesting problem*, pp. 27–30 in *Proceedings of the WMC '99*, World Manufacturing Congress, Durham.

[22] Burke EK, Kendall G & Whitwell G, 2004, *A new placement heuristic for the orthogonal stock-cutting problem*, Operations Research, **52(4)**, pp. 655–671.

[23] Burke EK, Kendall G & Whitwell G, 2006, *A new placement heuristic for the orthogonal stock-cutting problem*, (Unpublished) Technical Report NOTTCS-TR-2006-3, School of Computer Science and Information Technology, University of Nottingham, Nottingham.

[24] Chartrand G & Oellermann O, 1993, *Applied and algorithmic graph theory*, McGraw-Hill Inc., Singapore.

[25] Chazelle B, 1983, *The bottom-left bin packing heuristic: An efficient implementation*, IEEE Transactions on Computers, **32(8)**, pp. 697–707.

[26] Christofides N & Whitlock C, 1977, *An algorithm for two-dimensional cutting problems*, Operations Research, **25(1)**, pp. 31–44.

[27] Chu C & La R, 2001, *Variable-sized bin packing: tight absolute worst-case performance ratios for four approximation algorithms*, SIAM Journal on Computing, **30(6)**, pp. 2069–2083.

[28] Chung FRK, Garey MR & Johnson DS, 1982, *On packing two-dimensional bins*, SIAM Journal on Algebraic and Discrete Mathematics, **3(1)**, pp. 66–76.

[29] The Coca-Cola Company, 1934, *US delivery truck from 1934*, [Online], [Cited October 14th, 2009], Available from `http://www.thecoca-colacompany.com/presscenter/presskit_120_image_library.html`

[30] Coffman EG, Galambos G, Martello S & Vigo D, 1998, *Bin packing approximation algorithms: Combinatorial analysis*, pp. 151–208 in Du D-Z & Pardalos PM (Eds), *Handbook of combinatorial optimization*, Kluwer Academic Publishers, Boston (MA).

[31] Coffman EG, Garey MR & Johnson DS, 1996, *Approximation algorithms for bin packing: A survey*, pp. 46–93 in Hochbaum DS (Ed), *Approximation algorithms for NP-hard problems*, PWS Publishing Company, Boston (MA).

[32] Coffman EG, Garey MR, Johnson DS & Tarjan RE, 1980, *Performance bounds for level-oriented two dimensional packing algorithms*, SIAM Journal on Computing, **9(4)**, pp. 808–826.

[33] Coffman EG & Lagarias JC, 1989, *Algorithms for packing squares: A probabilistic analysis*, SIAM Journal on Computing, **18(1)**, pp. 166–185.

[34] Coffman EG & Shor PW, 1990, *Average-case analysis of cutting and packing in two dimensions*, European Journal of Operational Research, **44(2)**, pp. 134–144.

[35] Cui Y, 2008, *Heuristic and exact algorithms for generating homogenous constrained three-staged cutting patterns*, Computers & Operations Research, **35(1)**, pp. 212–225.

[36] Cui Y, 2009, *CutWeb*, [Online], [Cited August 25th, 2009], Available from `http://www.gxnu.edu.cn/Personal/ydcui/English/index.htm`

[37] Cui Y, Yang Y, Cheng X & Song P, 2008, *A recursive branch-and-bound algorithm for the rectangular guillotine strip packing problem*, Computers and Operations Research, **35(4)**, pp. 1281–1291.

[38] Dagli CH & Poshyanonda P, 1997, *New approaches to nesting rectangular patterns*, Journal of Intelligent Manufacturing, **8(3)**, pp. 177–190.

[39] DEIS — Operations Research Group, 2004, *Library of instances*, [Online], [Cited August 25th, 2009], Available from `http://www.or.deis.unibo.it/research.html`

[40] Demšar J, 2006, *Statistical comparisons of classifiers over multiple data sets*, Journal of Machine Learning, **7**, pp. 1–30.

[41] Dowsland KA, 1993, *Some experiments with simulated annealing techniques for packing problems*, European Journal of Operational Research, **68(3)**, pp. 389–399.

[42] Dowsland KA & Dowsland WB, 1992, *Packing problems*, European Journal of Operational Research, **56(1)**, pp. 2–14.

[43] Dyckhoff H, 1990, *A topology of cutting and packing problems*, European Journal of Operational Research, **44(2)**, pp. 145–159.

[44] Eglese RW, 1990, *Simulated annealing: A tool for operational research*, European Journal of Operational Research, **46(3)**, pp. 271–281.

[45] Eisemann K, 1957, *The trim problem*, Management Science, **3(3)**, pp. 279–284.

[46] ESICUP, 2007, *Listing gallery: Data Sets 2D — Rectangular*, [Online], [Cited October 29th, 2007], Available from `http://paginas.fe.up.pt/~esicup/tiki-list_file_gallery.php?galleryId=3`

[47] Faina L, 1999, *An application of simulated annealing to the cutting stock problem*, European Journal of Operational Research, **114(3)**, pp. 542–556.

[48] Falkenauer E, 1996, *A hybrid grouping genetic algorithm for bin packing*, Journal of Heuristics, **2(1)**, pp. 1381–1231.

[49] Falkenauer E & Delchambre A, 1992, *A genetic algorithm for bin-packing and line balancing*, Proceedings of the 1992 IEEE International Conference on Robotics and Automation, Nice, pp. 1186–1192.

[50] Fekete SP & Van der Veen J, 2009, *Instances PackLib²*, [Online], [Cited April 15th, 2008], Available from `http://mo.math.nat.tu-bs.de/packlib/instances.shtml`

[51] Fernandez de la Vega W & Lueker GS, 1981, *Bin packing can be solved within* $1 + \eta$ *in linear time*, Combinatorica, **1(4)**, pp. 349–355.

[52] Frenk JBG & Galambos G, 1987, *Hybrid next-fit algorithm for the two-dimensional rectangle bin-packing problem*, Computing, **39(3)**, pp. 201–217.

[53] Friedman M, 1937, *The use of ranks to avoid the assumption of normality implicit in the analysis of variance*, Journal of the American Statistical Association, **32(200)**, pp. 675–701.

[54] Friesen DK & Langston MA, 1986, *Variable sized bin packing*, SIAM Journal on Computing, **15(1)**, pp. 222–230.

[55] Friesen DK & Langston MA, 1991, *Analysis of a compound bin packing algorithm*, SIAM Journal on Discrete Mathematics, **4(1)**, pp. 61–79.

[56] Garey MR, Graham RL, Johnson DS & Yao AC, 1976, *Resource constrained scheduling as generalized bin packing*, Journal of Combinatorial Theory, **21(3)**, pp. 257–298.

[57] Gilmore PC & Gomory RE, 1961, *A linear programming approach to the cutting-stock problem*, Operations Research, **9(6)**, pp. 849–859.

[58] Gilmore PC & Gomory RE, 1963, *A linear programming approach to the cutting-stock problem — Part II*, Operations Research, **11(6)**, pp. 863–888.

[59] Gilmore PC & Gomory RE, 1965, *Multistage cutting stock problems of two and more dimensions*, Operations Research, **13(1)**, pp. 94–120.

[60] Girkar M, MacLeod B & Moll R, 1992, *Performance bound for bottom-left guillotine packing of rectangles*, Journal of the Operational Research Society, **43(2)**, pp. 169–175.

[61] Glover F, 1986, *Future paths for integer programming and links to artificial intelligence*, Computers and Operations Research, **13(5)**, pp. 533–549.

[62] Golan I, 1981, *Performance bounds for orthogonal oriented two-dimensional packing algorithms*, SIAM Journal on Computing, **10(3)**, pp. 571–582.

[63] Gonçalves JF & Resende MGC, 2006, *A hybrid heuristic for the constrained two-dimensional non-guillotine orthogonal cutting problem*, INFORMS Journal on Computing (Submitted), [Online], [Cited on April 22nd, 2008], Available from `http://www.research.att.com/%7Emgcr/doc/2d-cutting.pdf`

[64] Gradišar M, Resinovič G & Kljajić M, 2002, *Evaluation of algorithms for one-dimensional cutting*, Computers and Operations Research, **29(9)**, pp. 1207–1220.

[65] Gu X, Chen G & Xu Y, 2005, *Average-case performance analysis of a 2D strip packing algorithm — NFDH*, Journal of Combinatorial Optimization, **9**, pp. 19–34.

[66] Haessler RW & Talbot FB, 1983, *A 0-1 model for solving the corrugator trim problem*, Management Science, **29(2)**, pp. 200–209.

[67] El Hayek J, Moukrim A & Negre S, 2008, *New resolution algorithm and pretreatments for the two-dimensional bin-packing problem*, Computers and Operations Research, **35(10)**, pp. 3184–3201.

[68] Henning MA & van Vuuren JH, 2008, *Introduction to network optimization: An undergraduate text*, (Unpublished) Lecture Notes, University of Stellenbosch, Stellenbosch.

[69] Hifi M, 1998, *Exact algorithms for the guillotine strip cutting/packing problem*, Computers and Operations Research, **25(11)**, pp. 925–940.

[70] Hifi M, 1999, *The strip cutting/packing problem: Incremental substrip algorithms-based heuristics*, Pesquisa Operacional, **19(2)**, pp. 169–188.

[71] Hifi M, 2003, *Library of instances*, [Online], [Cited on August 25th, 2009], Available from `ftp://cermsem.univ-paris1.fr/pub/CERMSEM/hifi/OR-Benchmark.html`

[72] Hifi M & Zissimopoulos V, 1997, *Constrained two-dimensional cutting: An improvement of Christofides and Whitlock's exact algorithm*, Journal of the Operational Research Society, **48(3)**, pp. 324–331.

[73] Hinton PR, 2004, *Statistics Explained*, 2nd Edition, Routledge, Hove.

[74] Hinxman AI, 1980, *The trim-loss and assortment problems: A survey*, European Journal of Operational Research, **5(1)**, pp. 8–18.

[75] Hopper E, 2000, *Two-dimensional packing utilising evolutionary algorithms and other meta-heuristic methods*, PhD Dissertation, University of Wales, Cardiff.

[76] Hopper E & Turton BCH, 1997, *Application of genetic algorithms to packing problems — a review*, pp. 279–288 in Chawdry PK, Roy R & Kant RK (Eds), *Proceedings of the second on-line world conference on soft computing in engineering design and manufacturing*, Springer, London.

[77] Hopper E & Turton BCH, 1999, *A genetic algorithm for a 2D industrial packing problem*, Computers in Engineering, **37(1)**, pp. 375–378.

[78] Hopper E & Turton BCH, 2001, *An empirical investigation of meta-heuristic and heuristic algorithms for a 2D packing problem*, European Journal of Operational Research, **128(1)**, pp. 34–57.

[79] Hopper E & Turton BCH, 2001, *A review of the application of meta-heuristic algorithms to 2D strip packing problems*, Artificial Intelligence Review, **16(4)**, pp. 257–300.

[80] Hopper E & Turton BCH, 2002, *Problem generators for rectangular packing problems*, Studia Informatica Universalis, **2(1)**, pp. 123–136.

[81] Hwang SM, Cheng YK & Horng JT, 1994, *On solving rectangle bin packing problems using genetic algorithms*, Proceedings of the 1994 IEEE International Conference on Systems, Man and Cybernetics, Part 2 (of 3), San Antonio (TX), pp. 1583–1590.

[82] IC#Code, 2009, *SharpDevelop 3.1*, [Online], [Cited November 10th, 2009], Available from `http://www.icsharpcode.net/OpenSource/SD/`

[83] Jakobs S, 1996, *On genetic algorithms for the packing of polygons*, European Journal of Operational Research, **88(1)**, pp. 165–181.

[84] Johnson DS, 1974, *Fast algorithms for bin packing*, Journal of Computer and System Sciences, **8(3)**, pp. 272–314.

[85] Johnson DS, Demers A, Ullman JD, Garey MR & Graham RL, 1974, *Worst-case performance bounds for simple one-dimensional packing algorithms*, SIAM Journal on Computing, **3(4)**, pp. 295–325.

[86] Johnsonbaugh R & Schaefer M, 2004, *Algorithms*, International Edition, Pearson Education Inc., Upper Saddle River (NJ).

[87] Kang J & Park S, 2003, *Algorithms for the variable sized bin packing problem*, European Journal of Operational Research, **147(2)**, pp. 365–372.

[88] Kantorovich LV, 1960, *Mathematical methods for organising planning production* (translated from a report in Russian, dated 1939), Management Science, **6(4)**, pp. 366–422.

[89] Karmarkar N & Karp RM, 1982, *An efficient approximation scheme for the one-dimensional bin-packing problem*, Proceedings of the 23rd Symposium on the Foundations of Computer Science (FOCS), Tucson (AZ), pp. 312–320.

[90] Kenmochi M, Imamichi T, Nonobe K, Yagiura M & Nagamochi H, 2009, *Exact algorithms for the two-dimensional strip packing problem with and without rotations*, European Journal of Operational Research, **198(1)**, pp. 73–83.

[91] Kenyon C & Rémila E, 2000, *A near-optimal solution to a two-dimensional cutting stock problem*, Mathematics of Operations Research, **25(4)**, pp. 645–656.

[92] Kinnersley NG & Langston MA, 1989, *Online variable-sized bin packing*, Discrete Applied Mathematics, **22(2)**, pp. 143–148.

[93] Kirkpatrick S, Gelatt CD & Vecchi MP, 1983, *Optimization by simulated annealing*, Science, **220(4598)**, pp. 671–680.

[94] Kourentzes N, 2009, PhD Candidate at the *Department of Management Science, Lancaster University*, [Personal Communication], Contactable at nikolaos@kourentzes.com

[95] Lai KK & Chan JWM, 1997, *Developing a simulated annealing algorithm for the cutting stock problem*, Computers and Industrial Engineering, **32(1)**, pp. 115–127.

[96] Lamont MMC, 2009, Lecturer, *Department of Statistics and Actuarial Science, Stellenbosch University*, [Personal Communication], Contactable at `mmcl@sun.ac.za`

[97] Lee CC & Lee DT, 1985, *A simple on-line bin-packing algorithm*, Journal of the Association for Computing Machinery, **32(3)**, pp. 563–572.

[98] Lee HL & Nahmias S, 1993, *Single-product, single-location models*, pp. 3–55 in Graves SC, Rinnooy Kan AHG & Zipkin PH (Eds), *Logistics of production and inventory*, North-Holland, Amsterdam.

[99] Lesh N, Marks J, McMahon A & Mitzenmacher M, 2003, *New Heuristic and interactive approaches to 2D rectangular strip packing*, Technical Report TR2003-18, Mitsubishi Electrical Research Laboratories, Cambridge (MA), [Online], [Cited August 26, 2009], Available from `http://www.eecs.harvard.edu/~michaelm/postscripts/ijcai2003.pdf`

[100] Liu D & Teng H, 1999, *An improved BL-algorithm for genetic algorithms of the orthogonal packing of rectangles*, European Journal of Operational Research, **112(2)**, pp. 413–419.

[101] Lodi A, 1999, *Algorithms for two-dimensional bin packing and assignment problems*, PhD Dissertation, Università di Bologna, Bologna.

[102] Lodi A, Martello S & Monaci M, 2002, *Two-dimensional packing problems: A survey*, European Journal of Operational Research, **141(2)**, pp. 241–252.

[103] Lodi A, Martello S & Vigo D, 1998, *Neighborhood search algorithm for the guillotine non-oriented two-dimensional bin packing problem*, pp. 125–139 in Voß S, Martello S, Osman IH & Roucairol C (Eds), *Meta-heuristics: Advances and trends in local search paradigms for optimization*, Kluwer Academic Publishers, Boston (MA).

[104] Lodi A, Martello S & Vigo D, 1999, *Approximation algorithms for the oriented two-dimensional bin packing problem*, European Journal of Operational Research, **112(1)**, pp. 158–166.

[105] Lodi A, Martello S & Vigo D, 1999, *Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems*, INFORMS Journal on Computing, **11(4)**, pp. 345–357.

[106] Lodi A, Martello S & Vigo D, 2002, *Recent advances on two-dimensional bin packing problems*, Discrete Applied Mathematics, **123(1)**, pp. 379–396.

[107] Lodi A, Martello S & Vigo D, 2004, *Models and bounds for two-dimensional level packing problems*, Journal of Combinatorial Optimization, **8(3)**, pp. 363–379.

[108] Lodi A, Martello S & Vigo D, 2004, *TSpack: A unified tabu search code for multi-dimensional bin packing problems*, Annals of Operations Research, **131(1–4)**, pp. 203–213.

[109] MacLeod B, Moll R, Girkar M & Hanifi N, 1993, *An algorithm for the 2D guillotine cutting stock problem*, European Journal of Operational Research, **68(3)**, pp. 400–412.

[110] Martello S, Monaci M & Vigo D, 2003, *An exact approach to the strip-packing problem*, INFORMS Journal on Computing, **15(3)**, pp. 310–319.

[111] Martello S, Pisinger D & Vigo D, 2000, *The three-dimensional bin packing problem*, Operations Research, **48(2)**, pp. 256–267.

[112] Martello S & Vigo D, 1998, *Exact solution of the two-dimensional finite bin packing problem*, Management Science, **44(3)**, pp. 388–399.

[113] The MathWorks, 2004, *MATLAB 7.0.1*, [Online], [Cited October 24th, 2009], Available from `http://www.mathworks.com/`

[114] The MathWorks, 2009, *Statistics Toolbox: Boxplot*, [Online], [Cited October 12th, 2009], Available from `http://www.mathworks.com/access/helpdesk/help/toolbox/stats/index.html?/access/helpdesk/help/toolbox/stats/boxplot.html`

[115] Microsoft Corporation, 2007, *ReDim Statement (Visual Basic)*, [Online], [Cited October 14th, 2008], Available from `http://msdn.microsoft.com/en-us/library/w8k3cys2.aspx`

[116] Microsoft Corporation, 2009, *ArrayList Class*, [Online], [Cited June 25th, 2009], Available from `http://msdn.microsoft.com/en-us/library/system.collections.arraylist.aspx`

[117] Microsoft Corporation, 2009, *.NET Framework Developer Center*, [Online], [Cited November 10th, 2009], Available from `http://msdn.microsoft.com/en-gb/netframework/default.aspx`

[118] Microsoft Corporation, 2009, *Visual Basic Developer Center*, [Online], [Cited June 25th, 2009], Available from `http://msdn.microsoft.com/en-us/vbasic/default.aspx`

[119] Monaci M, 2001, *Algorithms for packing and scheduling problems*, PhD Dissertation, Università di Bologna, Bologna.

[120] Morrow SJ, 1876, *Cinching and loading pack mule with flour during starvation march of Gen. George Crook's expedition into the Black Hills*, [Online], [Cited October 14th, 2009], Available from `http://www.archives.gov/research/american-west/`

[121] Murgolo FD, 1987, *An efficient approximation scheme for variable-sized bin packing*, SIAM Journal on Computing, **16(1)**, pp. 149–161.

[122] Nahmias S, 2005, *Production and operations analysis*, 5th Edition, McGraw-Hill, Singapore.

[123] Nel JH, 2009, Associate Professor, *Department of Logistics, Stellenbosch University*, [Personal Communication], Contactable at jhnel@sun.ac.za

[124] Nordcapital, 2004, *Ship christened at HHI in Ulsan, Korea: A premiere — giant vessel of the Super Post-Panamax class entering service with COSCO — Chinas number one liner company*, Media Release, Hamburg/Ulsan, 11 June.

[125] Ntene N, 2007, *An algorithmic approach to the 2D oriented strip packing problem*, PhD Dissertation, University of Stellenbosch, Stellenbosch.

[126] Ntene N, Ortmann FG & van Vuuren JH, 2007, *Strip & bin packing part I*, Paper presented at the 37th Annual Conference of the Operations Research Society of South Africa, Cape Town.

[127] Ntene N & Van Vuuren JH, 2009, *A survey and comparison of guillotine heuristics for the 2D oriented offline strip packing problem*, Discrete Optimization, **6(2)**, pp. 174–188.

[128] Ortmann FG, Ntene N & Van Vuuren JH, 2007, *Strip & bin packing part II*, Paper presented at the 37th Annual Conference of the Operations Research Society of South Africa, Cape Town.

[129] Ortmann FG, Ntene N & Van Vuuren JH, 2008, *Four new algorithms for the two-dimensional two-phase variable-sized bin packing method*, Paper presented at the 18th Triennial Conference of the International Federation of Operations Research Societies, Sandton.

[130] Ortmann FG, Ntene N & Van Vuuren JH, 2010, *New and improved level heuristics for the rectangular strip packing and variable-sized bin packing problems*, European Journal of Operational Research, **203(2)**, pp. 306–315.

[131] Ortmann FG & Van Vuuren JH, 2009, *Finding good solutions to the 2D multiple bin size bin packing problem by modifying strip packing heuristics*, Paper presented at the 38th Annual Conference of the Operations Research Society of South Africa, Stellenbosch.

[132] Pandit SNN, 1962, *The loading problem*, Operations Research, **10(5)**, pp. 639–646.

[133] Parreño F, Alvares-Valdes R, Oliveira JF & Tamarit JM, 2008, *A hybrid GRASP/VND algorithm for two- and three-dimensional bin packing*, Annals of Operations Research, [Online Serial], [Cited November 3rd, 2009], Available from `http://www.springerlink.com/content/kv805478206j7432/`

[134] Parreño F, Alvares-Valdes R, Tamarit JM & Oliveira JF, 2008, *A maximal-space algorithm for the container loading problem*, INFORMS Journal on Computing, **20(3)**, pp. 412–422.

[135] Pearl J, 1984, *Heuristics: Intelligent search strategies for computer problem solving*, Addison-Wesley Publishing Company, Menlo Park (CA).

[136] Performance Metal Fabricators, Inc., 2004, *Services*, [Online], [Cited October 26th, 2009], Available from `http://www.performancemetalfab.com/services.htm`

[137] Pisinger D & Sigurd M, 2005, *The two-dimensional bin packing problem with variable bin sizes and costs*, Discrete Optimization, **2(2)**, pp. 154–167.

[138] Pisinger D & Sigurd M, 2007, *Using decomposition techniques and constraint programming for solving the two-dimensional bin-packing problem*, INFORMS Journal on Computing, **19(1)**, pp. 36–51.

[139] Porter E, 2003, *VB.NET: Dynamic arrays*, [Online], [Cited December 11th, 2008], Available from `http://weblogs.asp.net/eporter/archive/2003/08/07/22943.aspx`

[140] Pureza V & Morabito R, 2006, *Some experiments with a simple tabu search algorithm for the manufacturer's pallet loading problem*, Computers and Operations Research, **33(3)**, pp. 804–819.

[141] Ratanapan K & Dagli CH, 1998, *An object-based evolutionary algorithm: The nesting solution*, pp. 581–586 in *Proceedings of the International Conference on Evolutionary Computation*, IEEE, Piscataway (NJ).

[142] Reeves CR, 1993, *Modern heuristic techniques for combinatorial problems*, Blackwell Scientific Publications, Oxford.

[143] Reingold EM, 1999, *Algorithm design and analysis techniques*, pp. **1**-1–**1**-27 in Atallah MJ (Ed), *Algorithms and theory of computation handbook*, CRC Press LLC, Boca Raton (FL).

[144] Runarsson TP, Jonsson MT & Jensson P, 1996, *Dynamic dual bin packing using fuzzy objectives*, Proceedings of the IEEE International Conference on Evolutionary Computation, Nagoya, pp. 219–222.

[145] SAS Institute Inc., 2009, *SAS 9.1.3*, [Online], [Cited October 24th, 2009], Available from `http://www.sas.com/`

[146] Scheithauer G, 1998, *Equivalence and dominance for problems of optimal packing of rectangles*, Ricerca Operativa, **27(83)**, pp. 3–34.

[147] Scheithauer G, Riehme J, Rietz J & Belov G, 2006, *Test instances & results*, [Online], [Cited August 25th, 2009], Available from `http://www.math.tu-dresden.de/~capad/`

[148] Sleator DDKDB, 1980, *A 2.5 times optimal algorithm for packing in two dimensions*, Information Processing Letters, **10(1)**, pp. 37–40.

[149] Steinberg A, 1997, *A strip-packing algorithm with absolute performance bound 2*, SIAM Journal on Computing, **26(2)**, pp. 401–409.

[150] Sun Microsystems Inc., 2009, *OpenOffice.org 3.1.1*, [Online], [Cited November 10th, 2009], Available from `http://www.openoffice.org/`

[151] Sweeney PE & Paternoster ER, 1992, *Cutting and packing problems: A categorised, application-oriented research bibliography*, Journal of the Operational Research Society, **43(7)**, pp. 691–706.

[152] Tukey JW, 1977, *Exploratory data analysis*, Addison–Wesley Publishing Company, Manila.

[153] Valenzuela CL & Wang PY, 2001, *Heuristics for large strip packing problems with guillotine patterns: An empirical study*, Metaheuristic International Conference 2001, Porto, [Online], [Cited April 22nd, 2008], Available from `http://users.cs.cf.ac.uk/C.L.Mumford/papers/binpaper.pdf`

[154] Van Vuuren JH & Ortmann FG, 2009, *Benchmarks*, [Cited August 25th, 2009], Available from `http://www.vuuren.co.za/main.php` → *benchmarks*

[155] Wang PY, 1983, *Two algorithms for constrained two-dimensional cutting stock problems*, Operations Research, **31(3)**, pp. 573–586.

[156] Wang PY & Valenzuela CL, 2001, *Data set generation for rectangular placement problems*, European Journal of Operational Research, **134(2)**, pp. 378–391.

[157] Wäscher G, Haußner H & Schumann H, 2007, *An improved typology of cutting and packing problems*, European Journal of Operational Research, **183(3)**, pp. 1109–1130.

[158] Wei L, Zhang D & Chen Q, 2009, *A least wasted first heuristic algorithm for the rectangular packing problem*, Computers and Operations Research, **36(5)**, pp. 1608–1614.

[159] Yanasse HH, Zinober ASI & Harris RG, 1991, *Two-dimensional cutting stock with multiple stock sizes*, Journal of the Operational Research Society, **42(8)**, pp. 673–683.

APPENDIX A

Packing Software

Contents

In order to test all the algorithms listed in Chapters 3–5, and those in Chapter 7, it was necessary to write a software program to read the benchmark instances from a file, perform the packing tasks and report the results. This was done in fulfilment of Dissertation Objectives VI, XI and XIV. The resulting packing software is described in the first section of this appendix. In the second section the benchmark generator for the MBSBPP (see §8.1.2), which was designed in partial fulfilment of Dissertation Objective XII(a), is briefly described.

A.1 A Decision Support System

When starting the program the user is greeted by the window shown in Figure A.1. In the top-left corner the user may select one of three options. Selecting the *Strip Packing* radio button (selected by default) allows the user access to all the algorithms described in Chapters 3–5, and the benchmark instances used to arrive at the results reported in Chapter 6. Selecting the *Bin Packing* radio button gives the user access to all the strip packing algorithms combined with the 2SMBSBP algorithm described in §7.2, and the benchmark instances listed in §8.1. If the radio button labelled *Other* is selected, the user is given the choice of executing all strip/bin packing algorithms on all strip/bin packing benchmark instances, either to determine only the strip height or bin utilisation, or to find a packing and measure the time required by every algorithm to find a solution to each problem instance.

The top-right corner of the window contains three check boxes. The first, labelled *Guillotine*, restricts the algorithms available to the user to the set of algorithms that guarantee a guillotine packing. This check box is not ticked by default in order to allow the user to select any of the algorithms. The second check box is labelled *Display Time*. If this box is checked, then a window appears when an algorithm has found the solution to a problem. This window contains the time required by the algorithm to find a solution to the problem. Finally, the check box labelled *Keep Aspect Ratio*, which is checked by default, restricts the picture box displaying a packing solution to showing the solution with items and a strip/box that has the correct height/width ratio. If this box is not checked, then the heights and widths of the items may change to fill the picture box maximally, thereby changing the height/width ratio of the items.

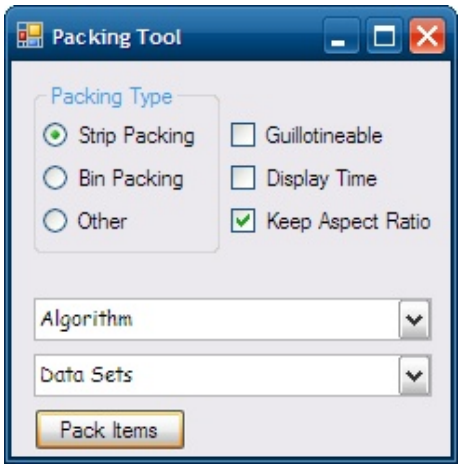
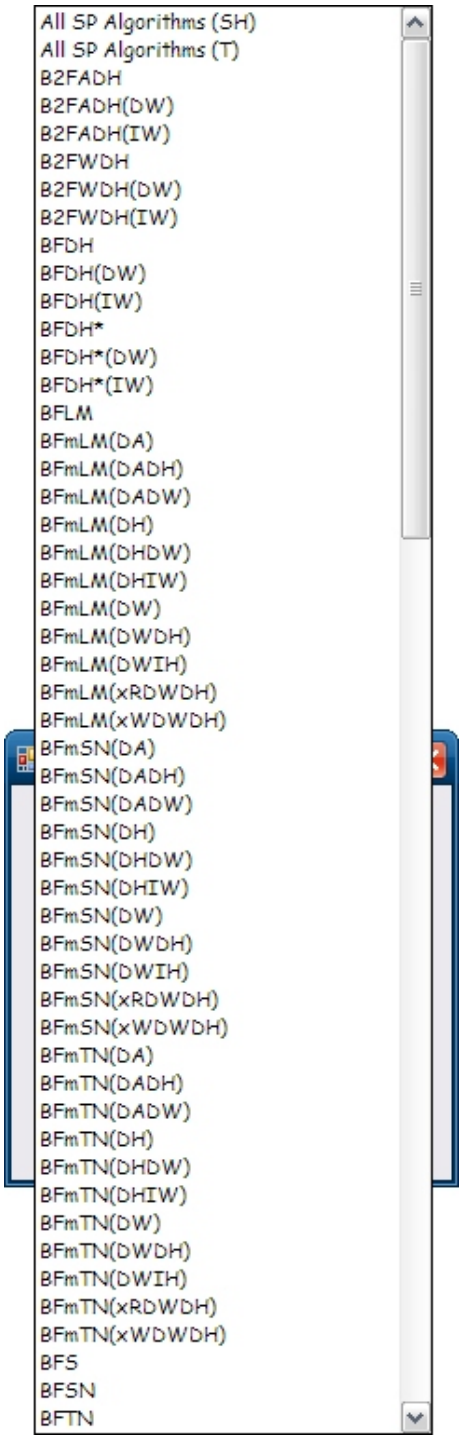


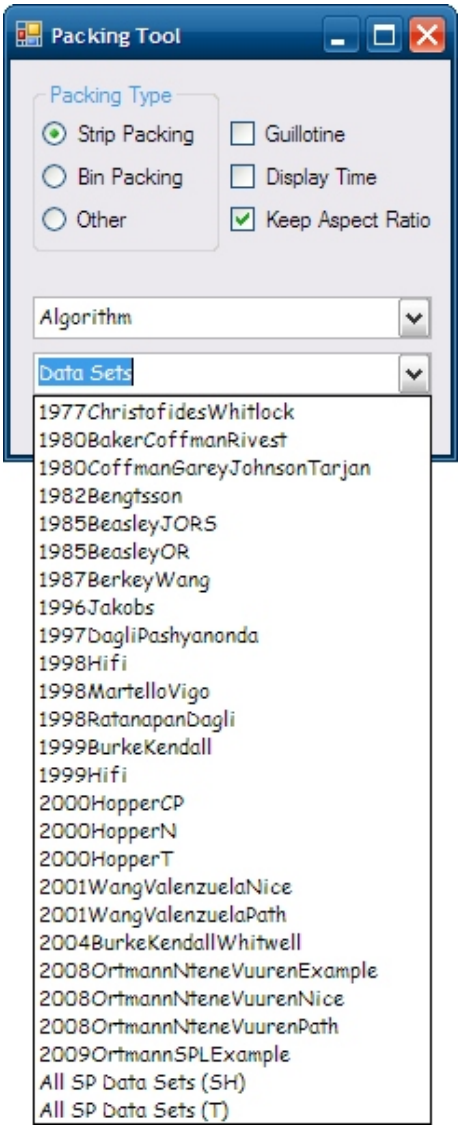
Figure A.1: *The first window that opens when the packing software is executed.*

The bottom half of the window has four text boxes (which remain hidden until required), two combo boxes and a single button. The top combo box may be used to select the algorithm (as shown in Figure A.2(a)), and the bottom combo box may be used to select the benchmark instance to which the algorithm should be applied (see Figure A.2(b)). The user's choice is recorded by a mouse-click. Depending on the choice of algorithm, the upper text box shown in Figure A.2(c) may appear. This text box is used to input the necessary parameters for the algorithms: the join percentage for algorithms JOIN and SL, the search space for the B2F algorithms or the split point for the xW and xR sorting algorithms. The bottom three text boxes appear, or disappear, depending on the selected set of benchmark instances. For example, selecting the benchmark instances by Christofides and Whitlock [26] would cause a single text box to appear. Selecting the benchmark instances by Hopper and Turton [75, 79, 80] would cause the appearance of two text boxes, while selecting the instances by Berkey and Wang [16], or Martello and Vigo [112] would result in the appearance of three text boxes.

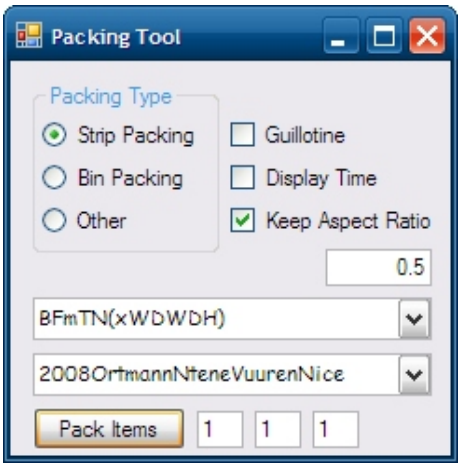
Once the algorithm and benchmark instance have been selected, the button labelled *Pack Items* may be clicked. When the algorithm has completed the necessary calculations a new window opens. It has the appearance of the screen shot in Figure A.3(a). The title of the window contains the name of the algorithm selected, followed by the name of the benchmark instance to which it was applied. The packing is shown in a picture box in the centre of the new window, and resizes with resizing of the window. The items retain their aspect ratios if the *Keep Aspect Ratio* check box was checked, or change shape to fill the picture box maximally if the check box was not checked. Above the picture box is a text label that shows the width of the packing. Below the bottom-right corner of the picture box is another text label containing the height of the strip. Below the bottom-left corner of the picture box are a number of labels. The topmost label contains the name of the algorithm that was used to perform the packing and the label under that contains the name of the benchmark instance represented in the picture box. Below the instance name is the number of items in the instance, followed by the number of bins used (the text *Strip Packing* appears here in the case of the strip packing problem). The routine that displays the packing calculates the sum of the items' areas and displays it in the third row from the bottom. The row below that is a display of the area of the strip that is used, calculated by multiplying the strip width by the strip height. Finally, the utilisation is presented in the final row and is calculated by dividing the total item area by the strip area. Every packing causes a new window to be opened, allowing the user to view multiple packings at the same time.



(a) *Algorithm combo box*

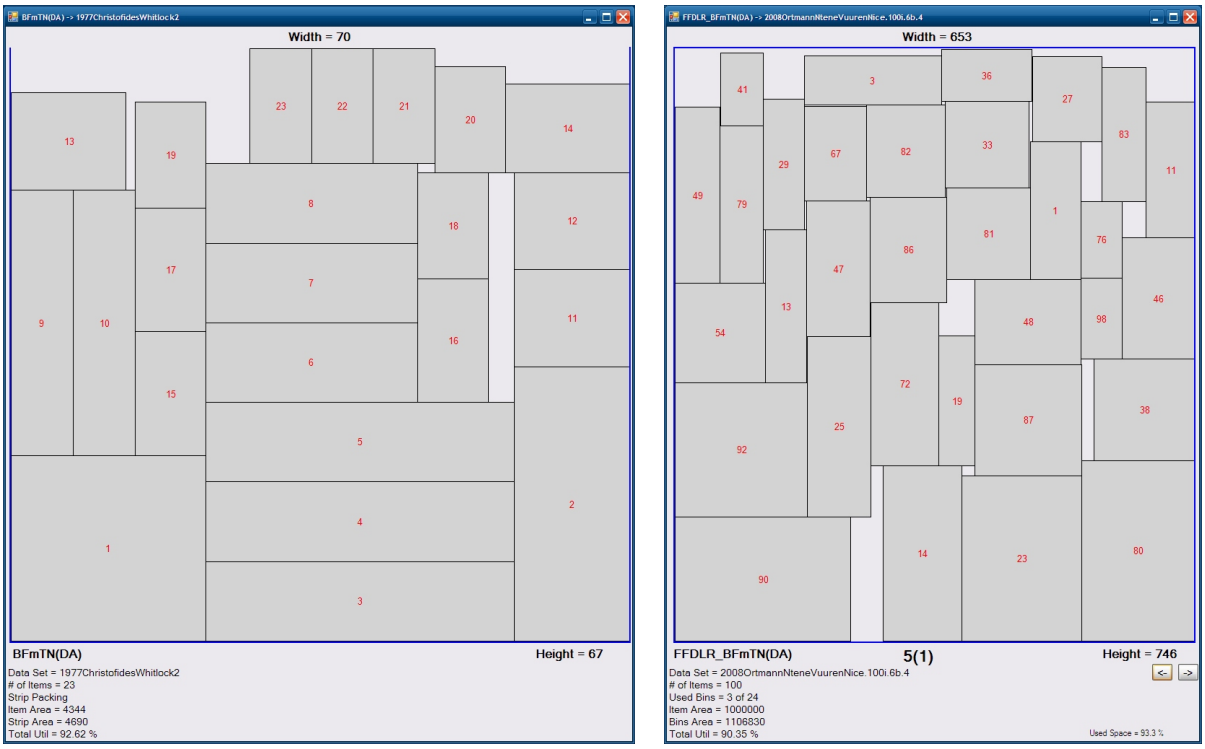


(b) *Benchmark combo box*



(c) *Text boxes*

Figure A.2: *Screen shots of the combo and text boxes on the main window.*



(a) Strip packing results window (b) Bin packing results window

Figure A.3: *Screen shots of the results windows for strip and bin packing.*

The results window has a very similar appearance in the case of bin packing, with some minor differences (see Figure A.3(b)). The window title takes the same form as for the strip packing problem; the bin packing algorithm is followed by the name of the benchmark instance. Above the picture box the width of the bin is given, while the bin height is given below the bottom-right corner of the picture box. Two buttons with arrows are located below the bin height label. Clicking on the left-hand button results in a display of the previous bin containing items in the sorted bin list. If the current bin is the first in the list, clicking on the left-hand button will show the packing in the last item-containing bin in the list. Clicking on the right-hand button shows the next item-containing bin in the list. The name of the bin is shown below the centre of the picture box. If there are many bins with the same reference name, then the number of the copy is shown in parentheses. The name of the algorithm may be found below the bottom-left corner of the picture box. Directly below the algorithm name is the name of the benchmark instance, followed by the number of items in that instance. The fourth row shows how many bins have been used of the total number available. The next two rows contain the total area of items and the sum of the areas of the bins that contain items. These two values are used to calculate the utilisation, shown in the last row. To the right of the final row is the utilisation of the bin that is currently shown in the picture box. In the example in Figure A.3(b), 93.3% of the current bin is covered by items, while the overall utilisation is only 90.35%.

If the user selects the options *All SP Algorithms (SH)* or *All SP Algorithms (T)* in conjunction with any benchmark instance, or the *All SP Data Sets (SH)* or *All SP Data Sets (SH)* options in conjunction with an algorithm, and clicks on the *Pack Items* button, a new window appears. This window is shown in Figure A.4 and is labelled *Comparison Running* in order to indicate that multiple packings are taking place. The same window appears when *Compare All SP Algorithms (SH)*, *Compare All SP Algorithms (T)*, *Compare All BP Algorithms (T)* or *Compare All BP*

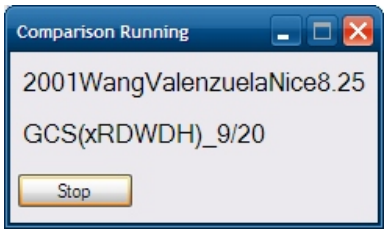
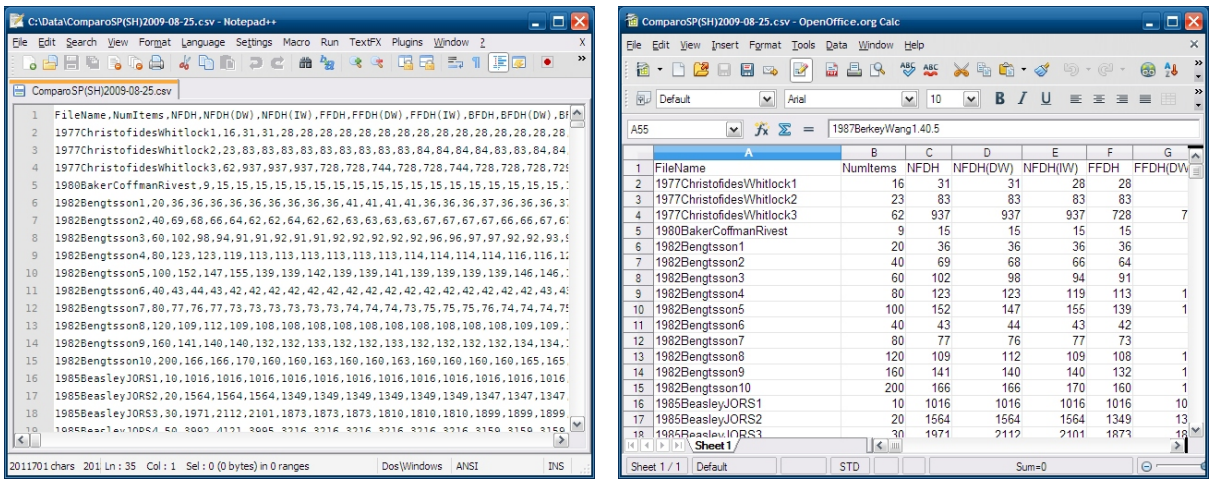


Figure A.4: *A screen shot of the window that appears when multiple packings are taking place.*

Algorithms (U) are selected. These options become available when the radio button labelled *Other* is selected. There are two labels in the window. The upper one lists the name of the benchmark instance currently being packed, while the lower one lists the name of the algorithm that is performing the packing. The button labelled *Stop* allows the user to interrupt the packing procedure. If the user had selected an option that only finds the strip packing height (SH) or bin utilisation (U), then all the calculations are performed in a new thread that is given a very low priority on the CPU in order to allow the user to use the computer resources with minimal interference from the packing program. If the user selected any of the options that measure time (T), then the calculations are performed by a thread that runs at the highest priority. This is done in an attempt to minimise any interference from any other processes running on the CPU at the same time. If there are other processes running, then they may interfere with the completion time of the algorithms.[1]



(a) Output in a text editor

(b) Output in spreadsheet software

Figure A.5: *Screen shots of the results output for text editors and spreadsheet software.*

The results of these comparative runs are written to a file. These files are *comma-separated values* (CSV) files that may be opened in most text editors (see Figure A.5(a)) or spreadsheet programs (see Figure A.5(b)). All numerical values or text are separated by means of commas, and these commas may be interpreted as column separators. If an algorithm was tested on all benchmark instances, then the resulting file name consists of the name of the algorithm, plus `(SP)` (or `(U)`) if the packing height (or utilisation) was sought, or `(T)` if the packing height (or utilisation) and execution time were to be found. The date that the com-

[1]The tests were completed on a dual-core CPU, which further minimised the chances of interference from the other processes, as they could run on the core not being used by the packing software. No attempt was made to allow the packing software to use both cores.

parison began is added to the end of the file name. Therefore, if the BLF(DA) algorithm was applied to all benchmark instances in order to find only the resulting packing heights, the file name should read `BLF(DA)(SH)2010-01-01.csv`. If the packing heights (and times) were sought for a specific benchmark instance, the file name begins with the name of the benchmark instance. For example, if all the algorithms were applied to the benchmark instance N12 by Burke *et al.* [22] in order to find both the packing height and execution time, then the file name should read `2004BurkeKendallWhitwellN12(T)2010-01-01.csv`. If all strip packing algorithms were applied to all benchmark instances, then the resulting file name would be `ComparoSP(SH)2010-01-01.csv`, while the multiple bin size bin packing algorithms and benchmark instances would yield a file name resembling `ComparoMBSBP(U)2010-01-01.csv`. The files typically have the following structure. The first row contains the names of the algorithms, while the first column contains the names of the benchmark instances. The second column contains the number of items that are packed. This is followed by rows of strip packing heights, or bin packing utilisations. If the time was required, these columns are followed by further columns that contain the running times for each algorithm/instance pair.

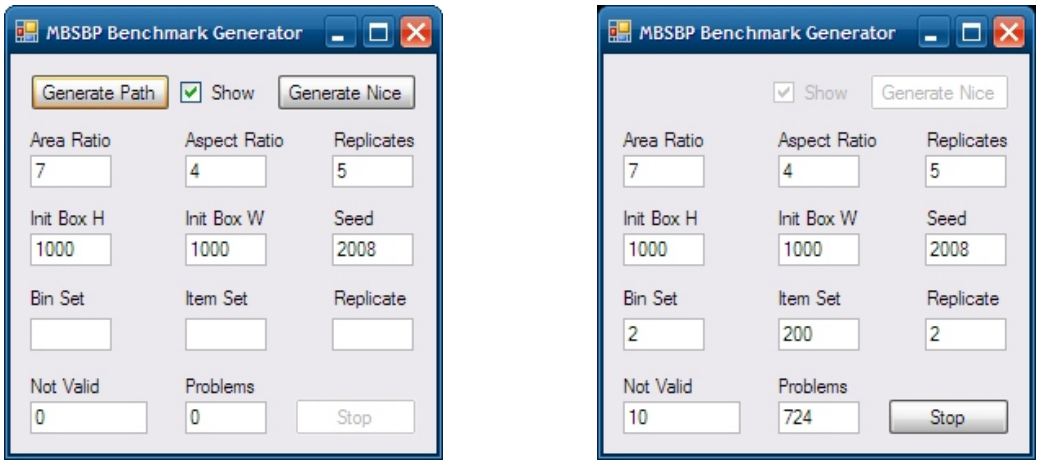
A.2 An MBSBPP Benchmark Generator

The software was designed to be able to generate benchmark instances for the MBSBPP. When the program is started, the window shown in Figure A.6(a) welcomes the user. It allows the user to choose the area ratio restriction (the ratio between the largest and smallest items by area), and the aspect ratio constraint (the ratio between the item's height and width), for "nice" items. The default values are set to the values used by Wang and Valenzuela [156] for their benchmark instances for the strip packing problem. The window also allows the user to enter the number of replicates that are required for each n/M pair, where n denotes the number of items and M denotes the number of bin sizes. The second row of text boxes allows the user to set the initial size of the rectangle from which the bins and items are cut, and the seed of the random number generator. The text boxes in the third and fourth rows are not for user input; these text boxes are reserved for data output. The third row of text boxes shows the user which instance the software is attempting to generate. The left-hand text box shows the number of bin sizes, the centre box shows the number of items for the problem, and the right-hand box shows which replicate is sought.

The two text boxes in the final row list the number of problems that occur, or the number of invalid instances that are found. There are four reasons why an instance may be labelled not valid:

- The tallest item has a larger height than the shortest bin.
- The widest item has a width larger than the smallest bin width.
- Any of the items have a dimension of length 0.
- The area of the bins (before multiple copies are generated) does not add up to the area of the original bin from which the bins were cut. This should not happen, but was included as a safety measure.

There are constraints on where the item may be cut in order to result in two "nice" items, if such items are required. In their paper on the creation of such benchmark instances, Wang and Valenzuela proved that an item of width W cannot be sliced vertically into two items that both



(a) Initial window (b) Window when finding benchmarks

Figure A.6: *Screen shots of the main window for the benchmark generator.*

have the correct aspect ratio when $W > 2\rho H$, where ρ denotes the maximum aspect ratio and H denotes the item height. The same is true when $W < 2H/\rho$. Similarly, items may not have a height $H > 2\rho W$ or $H < 2W/\rho$ in order for it to be split horizontally into two valid items. If any such items are found during the generation of a benchmark instance, the generation of the instance is aborted and a new search for an instance is initiated. These problems are added together to give the value shown in text box labelled *Problems*.

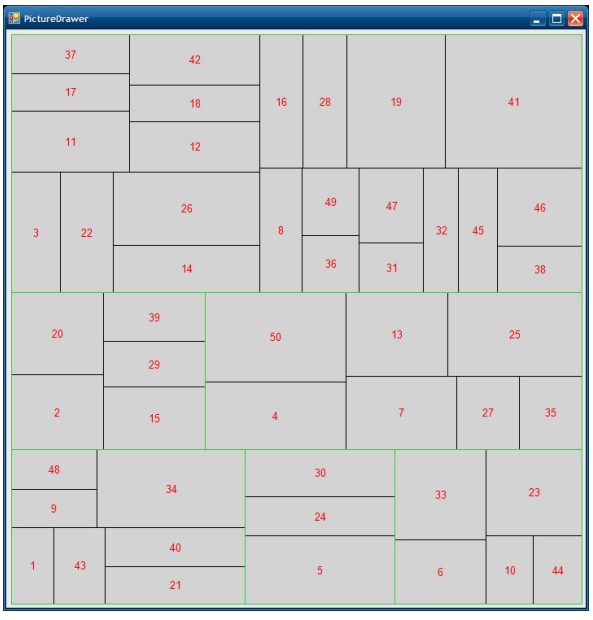


Figure A.7: *A screen shot of the window that appears to show a newly-created benchmark instance for the MBSBPP.*

There are three buttons on the main window. The button in the top-left corner initiates the search for "pathological" benchmark instances. Selecting the button in the top-right corner initiates the search for "nice" benchmark instances. If the check box between the two items is ticked, then a window resembling the example in Figure A.7 will appear and show each of the benchmark instances as they are generated. The button in the bottom-left corner of the

window may be clicked during the benchmark generation process in order to stop the search for these instances. The output is written to CSV files in the path "`C:\MBSBPdata\`". Each instance consists of two files, one containing the items' dimensions and another containing the dimensions of the bins. The files take the form `MBSBP`x`.`n`i.`M`b.`yz`.csv`, where x may be either `nice` or `path`, the variable n is the number of items in the instance, M denotes the number of bin sizes, y is the replicate number and z is either `i` or `b`, indicating that the file contains either the items' or bins' dimensions. The files for the "nice" data have the values used for the area and aspect ratio constraints in parentheses before the file extension.

APPENDIX B

Contents of the Compact Disc Accompanying this Dissertation

Included with this dissertation is a compact disc containing some results and source code for the software presented in Appendix A. In this appendix the information on this disc is clarified. There are five directories on this compact disc, namely *Electronic Dissertation*, *Packing Software*, *Required Software*, *Results* and *Source Code*. The contents of these folders are described below.

Electronic Dissertation. This folder contains electronic copies of this dissertation in PostScript and PDF formats.

Packing Software. This folder contains the software described in §A.1. Before this software is used, the folder should be copied to a convenient location on a hard disk drive. The software may be started by double-clicking on the file named `Packing.exe`. Any results from multiple comparisons will be written into the subfolder labelled *Data*. This subfolder contains the benchmark problem instances used in this dissertation in folders labelled with the name of the problem, *i.e. MBSBPP Data Sets*, *SBSBPP Data Sets* and *SP Data Sets*. The file labelled `lpsolve.dll`, supplied by Berkelaar *et al.* [15], is required by the packing software to solve knapsack problems during the execution of the KP family of algorithms. This program may require the Microsoft .NET 3.5 Framework [117] to be installed on the computer on which it is executed.

Required Software. This folder contains the software used to compile the source code (supplied in the folder labelled *Source Code*), and the software required to view the spreadsheets of data. The software was written in the SharpDevelop Integrated Development Environment [82] and the data was analysed in OpenOffice.org [150].

Results. This folder contains three subfolders. The folder labelled *Bin Packing Results* contains the results achieved by the 2SMBSBP algorithm when applied to the MBSBPP and SBSBPP. The folder labelled *SAS Files* contains all the input and output files for the statistical tests that were performed by means of the SAS Software Suite [145] on the packing results. The folder labelled *Strip Packing Results* contains the results obtained by the packing software when applied to the strip packing benchmark instances.

Source Code. This folder contains the source code for the MBSBPP benchmark generator described in §A.2, and the source code for the packing software described in §A.1. The

corresponding projects may be opened in SharpDevelop [82] by opening the files with the `.vbproj` extension. These software projects were compiled on a 32bit Windows XP (SP3) PC.