ALGORITHMIC TECHNIQUES FOR THE MICRON AUTOMATA PROCESSOR

A Thesis
Presented to
The Academic Faculty

by

Indranil Roy

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science and Engineering

Georgia Institute of Technology
August 2015

ALGORITHMIC TECHNIQUES FOR THE MICRON AUTOMATA PROCESSOR

Approved by:

Professor Srinivas Aluru, Advisor
School of Computational Science and Engineering
Georgia Institute of Technology

Professor Alberto Apostolico
School of Computational Science and Engineering
Georgia Institute of Technology

Professor Sudhakar Yalamanchili
School of Computational Science and Engineering
Georgia Institute of Technology

Professor David A. Bader
School of Computational Science and Engineering
Georgia Institute of Technology

Professor Richard Vuduc
The School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: 24 April 2015

To my family, loved ones and friends.

PREFACE

This dissertation is the first in-depth study on the use of the Micron Automata Processor, a novel re-configurable co-processor that supports direct hardware implementation of a set of non-deterministic finite automata over a streaming input. This study is made interesting by the fact that this co-processor will be the only commercially available processor which is directly based on the Multiple Instruction Single Data architecture. This allows it to efficiently handle a class of applications which are inherently difficult to compute using existing processors.

By design, this processor is well-suited to accelerate applications which need to find all occurrences of thousands of complex string-patterns in the input data. This document details our research to implement such applications on this processor and evaluate the performance benefits. We have also significantly widened the scope of the applications which can be accelerated through this processor by finding ways to solve some classic graph problems such as finding maximal cliques and Hamiltonian cycle/paths in a graph, and problems which can be reduced to those graph problems, such as the boolean satisfiability (SAT) problem. We have outlined various strengths and weaknesses of the processor, and describes techniques to estimate performance. These algorithmic and estimation techniques are already being used by other researchers to provide solutions to a wide variety of problems such as *Brill tagging* and *Association Rule Mining*.

This processor has been in the making for the past eight to nine years. Our research group was the first outside the company, that the developers reached out to for application development in 2011. Subsequently, two other academic groups joined later. The processor was unveiled to the world in November 2013, at the *Supercomputing* conference. Since then, many other researchers have joined the effort through the *Center for Automata*

Processing.

Initially, the aim of our group was to accelerate problems in string pattern matching from the field of bioinformatics. We started with accelerating the search for known string patterns called *motifs* in protein sequences and extended it to accelerate the discovery of unknown motifs in biological sequences, a known *NP*-complete problem.

While coming up with a solution for the latter, it became evident that this processor can be used to solve various problems on unweighted graphs. However, the following transformations are imperative for solving graph problems on the Automata Processor. First, the nodes and edges in the graph must be represented using strings which can be streamed to the co-processor. Second, the graph problem must be converted into a string-pattern matching problem which can be evaluated using automata over the streaming data. We developed algorithmic techniques to represent nodes and edges of any generic graph using strings, and create efficient automata which can be used to solve a variety of graph problems including finding cliques, Hamiltonian path and cycles, connected components, and single-source shortest paths; and other problems like boolean satisfiability which can be reduced to graph problems. These techniques are detailed in this dissertation.

It is hoped that the the advent of this processor will galvanize the field of automata-based applications by providing a means to support their implementation in hardware. Towards that end, this dissertation contributes many algorithmic techniques and automata-design strategies which can be used to develop applications which run efficiently on this new processor.

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my sincerest gratitude to those who contributed towards the completion of this thesis.

First and foremost, I thank my principal advisor Dr. Srinivas Aluru. His guidance, patience and support throughout my doctoral studies has been exemplary. Without him this dissertation would not have been possible.

I am also grateful to Dr. David A. Bader, Dr. Professor Alberto Apostolico, Dr. Professor Richard Vuduc, and Dr. Sudhakar Yalamanchili for their valuable time and suggestions. Their varied experience and in depth knowledge has often provided guiding light and provided validation to the research that I conducted.

I will forever be indebted to my parents. Their sacrifices in getting me here are momentous. I must also thank my roommates Yuzhu and Rahul for being with me through thick and thin over the past few years. They have developed mutual respect for each other solely on the capability of tolerating me. Besides them, I owe gratitude to my delightful labmates, with whom I passed the best part of most days. They have made my workload considerably lighter.

Finally, I would like to thank Paul Dlugosch, Paul Glendenning and Michael Leventhal from Micron Technology, Inc. for various collaborations which were required to bring this work to fruition.

Contents

List of Tables

List of Figures

SUMMARY

Our research is the first in-depth study in the use of the Micron Automata Processor, a novel re-configurable streaming co-processor which is designed to execute thousands of Non-deterministic Finite Automata (NFA) in parallel. By design, this processor is well-suited to accelerate applications which need to find all occurrences of thousands of complex string-patterns in the input data. We have validated this by implementing two such applications, one from the domain of network security and the other from the domain of bioinformatics. Both these applications are significantly faster than their state-of-art counterparts based on existing accelerators. Our research has also widened the scope of the applications that can be handled using this processor by designing algorithms to solve graph problems. In order to make this possible, we developed fundamental techniques to represent nodes and edges in the graph using strings, and convert the graph problem into a string-pattern matching problem. Subsequently, we created efficient automata which identify the patterns representing output graph structures like maximal-cliques, connected components, Hamiltonian paths, Hamiltonian cycles, etc. These techniques have been used in applications such as discovery of new *motifs* in biological sequences, and solving challenging instances of the *boolean satisfiability* problem. These applications and algorithms have not only guided other developers who joined the program later, but also yielded valuable feedback and design-inputs for the next generation of the chip which is currently in the design phase. We hope that this work paves the way for the early adoption of this upcoming architecture and continues to aid in the development of efficient solutions to some of the problems which are currently computationally challenging.

Chapter I

INTRODUCTION

The capability to perform complex and numerous pattern-matching operations in parallel is commonly found in nature. For example, it lies at the basis of natural language processing, object recognition and analytical skills required to compose and understand the text and diagrams in this document. Even insects and animals use pattern-matching subconsciously to search for food, hunt, identify threats etc. However, our current computers are not adept at this kind of computing. This is partially because of the rarity of processors based on the Multiple Instruction Single Data (MISD) architecture, necessary to perform multiple pattern matching operations on a single input data in parallel. In fact, barring the *systolic arrays*, whose classification as MISD is questionable, there are no known commercially available processors for this basic classification of processors described in Flynn's taxonomy [30].

The most challenging part of building high performance MISD machines is to provide routing capability to stream the same data to thousands of processing units operating in parallel. Hardware implementations have been proposed since 1982 [29]. However, not much has come to fruition till date. Most recent efforts [75, 11, 54, 74, 72] concentrate on the utilization of high-performance reconfigurable Field-Programmable Gate Array (FPGA) processors to implement parallel pattern-recognition engines in hardware. However limitations still remain in these implementations. In [25], the authors have provided an extensive review of the related literature and categorized these limitations on ten key criteria, the most important of which being space and power efficiency.

However, some very interesting developments have been reported recently. IBM introduced its *SyNAPSE* chip [40] in 2014 whose processing units are highly connected

'brain-like' neurons. The most recent chip has one million such neurons. Similarly, Hewlett Packard announced that they would release a device in 2017, grandly named *The Machine* [38], which uses *memristors* to unify memory and compute operations. With large scale interconnectivity within these memristors, they hope to perform large scale graph pattern-matching operations at high speeds. *DNA-computers* [49] employ biological molecules as processing units which interact with other molecules swimming around in the same chemical solution. Because, even a small volume of solution contains an extremely large number of biological molecules, it is expected that the *right* sequence of interactions will be carried out in at least a few instances. The idea is to use an exponential number of processing elements to compute an exponential number of operations in polynomial time. This method can be used to solve very complex problems such as finding the *Hamiltonian path* in a graph [6].

Another idea proposed since the 1990s has been the implementation of processing in memory [47]. Although, this idea has promise, all attempts to manufacture such a chip failed in the 1990s. This is partially attributed to the manufacturing technology at the time which limited the amount of circuitry that could be fitted inside a single chip. Since then, the capacity of memory chips have grown exponentially based on Moore's law, but the growing disparity of speeds between the CPU and the memory has also led to the so-called *memory-wall*. Therefore, memory manufacturers are having a re-look at this technology.

In this dissertation, we will look at one such attempt by *Micron Technology, Inc*. Christened the *Automata Processor* (AP) [25], it has been in the works for close to a decade now. For the majority of this duration, the project was shrouded in secrecy. In 2011, our research group was the first group outside Micron, whom the developers talked to about the technology. The goal of the ensuing collaboration was to find innovative solutions to accelerate a wide variety of applications using this chip.

Subsequently, in the November of 2013, the chip was unveiled to the rest of the world during the *Supercomputing* conference at Denver, Colorado. Since then, many research

groups have joined this effort and the chip production has reached advanced stages. Some initial batches have already been manufactured and tested within the company premises. It is expected that the chip will become available in the market in late 2015.

The AP chip is a reconfigurable co-processor which is designed to compute thousands of Non-deterministic Finite Automata (NFA) in parallel on a single input data stream. It uses Dynamic Random Access Memory (DRAM) technology to provide the underlying MISD architecture. As this chip is specialized to carry out only string-pattern recognition operations, it is both space and power-efficient. Besides, it is much easier to program for these specific operations than a general-purpose FPGA.

However, although the programming environment is very simple, coming up with efficient algorithms and automata for this processor may need ingenuity and innovation on behalf of programmers who are trained on traditional processing platforms. The AP provides new operations and new avenues of parallelism to exploit, while curtailing the capability to execute others. This might require take some time and learning to get used to.

This research was the first attempt to understand these concepts and establish some basic ground rules for the design algorithmic techniques and automata for the AP. It is hoped that this will aid developers in future to master this technology quickly and create solutions for problems which are challenging to solve using existing processor technologies.

In our endeavour, we progressively tackled harder and harder problems. We started with accelerating pattern-matching applications for which the use of the chip follows naturally, i.e. applications that need to find all occurrences of thousands of complex string-patterns in an input string. We graduated to handling various classical problems in unweighted graphs, for which the solutions were not intuitive. Along the way, we devised techniques to optimize the automata in terms of size, as well as execution time, by overcoming critical performance and routing bottlenecks.

Besides these, we developed assessment techniques to accurately estimate the run-times of applications, by taking into account the overheads of various steps in different stages in

the execution pipeline. By analysing the run-times of various parts of the applications using these techniques, we could provide critical feedback to the developers of the chip to make various refinements. Some of these refinements have already made their way into the current generation of the chip, while others will be incorporated in the next generation of the chip, which is currently in the design phase.

The rest of this dissertation is organized as follows. In Chapter 2, we have described the AP hardware from an application developer's perspective. Although limited, this description is succinctly captures the prominent features of the chip required to understand the algorithm designs in the rest of the document. Readers who are interested to know more about the architectural features of the hardware are directed to the online portal [5] and the following research paper [25].

In Chapter 3, simple ways to convert NFAs expressed as classical state-diagrams to *ANML-NFA* is described. ANML (pronounced as animal) stands for *Automata Network Markup Language* which is used to define automata for the AP. We have also outlined various optimization strategies for ANML-NFA. This may be useful to automata theory experts to convert existing automata designs to ANML-NFA. The goal is to galvanize the huge body of work which currently lies dormant for want of a suitable hardware.

Next, we delve into an intuitive use of the AP to accelerate applications which search for all occurrences of thousands of complex string-patterns in input data-streams. We chose two applications, one from the network security domain and the other from the field of bioinfomatics. The first application is called Fast-SNAP (for Fast-SNort using AP) and searches for all occurrences of over 4394 *signatures* of network intrusion in network data. These signatures have been obtained from the Snort [61] database. Using the resources on a single AP board, Fast-SNAP is estimated to support *deep packet inspection* of network data packets using all these signatures from the Snort database at $8-11$ Gbps. This is at least an order of magnitude faster than other applications using GPUs and FPGAs.

The second application is called PROTOMATA (for PROTein autOMATA) and looks

for all occurrences of 1308 protein *motifs* in protein sequences. These motifs have been downloaded from the PROSITE [66] database. Depending on the size of the input data and search settings, PROTOMATA can be about half a million times faster than its single-CPU-based counterpart. The above two applications are described in Chapter 4, and can serve as model implementations for other similar applications. Most of the automata parts are designed as modular *macros* which can be imported as is into future applications or with few minor modifications.

Once PROTOMATA was developed, which looks for occurrences of known motifs in the input data, the next logical step was to use the AP to accelerate the search for new unknown motifs. This second problem is significantly more difficult. In fact, the search for one of the simplest kinds of motifs called *(l,d)-motifs* is known to be *NP*-complete. The computational problem was first formulated by Pevzner in the year 2000 [57] and is called the *planted motif search* problem. Both *exact* and *approximate* solutions to this problem exist. Approximate algorithms are significantly faster, but they sometimes miss out on subtle motifs which are biologically important. Although exact algorithms to find all motifs exist, they take exponential time.

One of the ways to find these motifs is to reduce the problem to finding maximal cliques in an n-partitioned graph. A clique is a sub-graph wherein any two nodes are connected to each other. In an n-partitioned graph, the nodes are divide into n disjoint partitions, and no two nodes from the same partition are connected to one another. Therefore, the size of a maximal clique in a n-partitioned graph is at most n.

One aspect of this transformed graph problem was that the nodes in the graph represent strings, and the edges between them represented a bounded number of mismatches between these strings. We started with developing an automaton to count the number of mismatches between any two strings, by programming the first string as an automaton and streaming the other string to see which accept state the automaton reached. This automaton was then extended to a clique-finding automaton. The automaton was designed in such way

that its structure was agnostic of the input string with which it was designed. Multiple instances of such an automaton can be compiled and loaded quickly, once the input strings are provided. The complete application called MOTOMATA (for MOtif auTOMATA) is described in Chapter 5.

Developing MOTOMATA gave us a critical insight into solving graph problems using the AP. If graph nodes represent strings and edges represent a relation between those strings, then automata can be developed to check these properties in parallel leading to significant speed-up. This led us to developing solutions for other graph problems using generic graphs.

We designed a strategy to assign string labels to nodes in a generic graph, and define a relationship between those labels to signify the existence of an edge. The label of nodes which are connected by an edge exhibit this relationship, otherwise not. Some graph problems can be described as string-pattern-matching problems in this transformed notation. Automata designed to check for these relationships can then be executed in parallel in the AP to provide speed-up. We used this technique to solve various graph problems.

For example, the boolean satisfiability (SAT) problem checks whether there is some assignment of values $TRUE$ and $FALSE$ to the variables of a boolean formula, such that the formula evaluates to $TRUE$. This is a well-known NP-complete problem which can also be reduced to finding cliques of size n in an n-partitioned graph. We have already solved this problem in MOTOMATA. However, the characteristics of the input in the SAT problem is different from those of the input in MOTOMATA, lending the former to be non-viable solution as a SAT solver. Therefore, we had to come up with a new solution as described in Chapter 6.

In the planted motif search problem and the boolean SAT problem, the input itself is not a graph, but the algorithm converts it into one by representing different parts of the input as nodes in a graph. The edges between the nodes are not present in the input either, but a condition for the existence of an edge is provided. The objective of the algorithms is to find

a set of nodes which satisfy the connectivity constraint of a certain output graph structure, i.e. cliques of a certain size. Therefore, the primary goal of the automata is to check for the existence of all edges that are required to satisfy this connectivity constraint.

However in many other graph problems, all the nodes and edges are already listed in the input. Therefore, checking for the existence of edges is not the computational challenge. Instead, the problem is to arrange nodes and edges in the input graph into output graph structures such as search trees and acyclic paths. Besides, the graphs in the previous two problems where specialized i.e. the nodes were distributed into n-partitions. This is not the case with problems on generic graphs.

In Chapter 7, we describe methods to transform any input graph into a string notation which can be handled on the AP. Further, we define automata to use this string notation to identify the placement of the nodes and edges into the output structures for applications such as finding *single-source shortest paths*, *connected components*, and Hamiltonian paths and cycles in the graph. Some of these methods are estimated to be significantly faster than state-of-art implementations on other platforms, while others serve in demonstrating fundamental algorithmic techniques for future research.

In future, these methods may be improved to provide better performance or support a wider variety of graph problems, e.g. problems involving weighted graphs. Also, there is significant potential for research into exploring streaming solutions for graph problems using a single compute node connected to one or more AP(s), a fast flash-based I/O, and a large main-memory bank. Not only can such a system reduce the demands on cost, power and space, but may also provide efficient solutions to the large graph problems where exploiting data-locality may be difficult.

We have come a long way, starting with basic string-pattern matching applications to accelerating classic NP-hard graph problems. Some of the methods and techniques developed during this research have already started to aid other researchers to provide innovative solutions [73, 78] using the AP. It is hoped that this trend will continue into the future.

Chapter II

MICRON AUTOMATA PROCESSOR

The Automata Processor is one of the very few, if not the only processor based on the MISD architecture. The implementation of this architecture is done by using the Dynamic Random Access Memory (DRAM) technology. In a DRAM chip, a memory address and operation can be broadcast on every clock cycle to every *memory cell* in the chip using a system-wide bus. In AP, the memory-cells are replaced by *processing elements* and instead of a memory address and operation, one byte from the input data stream is broadcast in every clock cycle. A programmable *routing network* is used to connect the processing elements to one other. Through this routing network, elements activate each other in every clock cycle. Finally, an *output handling unit* is added to report whenever the occurrence of a pattern is detected.

The software required for programming the AP is available from an online developers portal [5]. A Software Development Kit (SDK) can be downloaded which describes methods to define and compile the automata and load the same into an AP. The SDK also provides run-time environment to handle all run-time operations.

Although, the chip is still not available, the SDK provides provides limited capability to simulate the execution of small automata inside the AP chip using a command-line interface. If a visual interface is desired, a *work-bench* can be downloaded to create, edit and simulate ANML-NFA through a graphical user interface. and visualize their execution over test data streams.

2.1 Programming Model

Executing programs on the AP involves two stages, namely the *configuration* stage and the *execution* stage. In the configuration stage, the user-defined automata are programmed into

the AP; and in the execution stage, these automata are executed in parallel on streaming data. If an automaton reaches its accept state, an output is created along with the offset in the stream where the event occurred.

2.1.1 Configuration Stage

The configuration stage can be further broken down into the *compilation* and *loading* steps. In the compilation step, the user-defined automata are converted into machine-loadable *images* using the AP compiler which comes with the SDK. This step involves executing complex place-and-route algorithms which may take considerable time based on the input automata. On the other hand, loading images into the AP board is extremely fast. For example, the entire AP board can be loaded with new automata in about 50 milliseconds.

According to well-known theory, finite automata and regular expression (regex) are equivalent, i.e. any finite automata can be represented as a regex, and vice-versa. The AP compiler also accepts regex defined in PCRE. In this case, the compiler internally converts them into their equivalent automata and then creates the loadable image from the same.

2.1.2 Execution Stage

During the execution stage, all the data strings to be checked are organized into a *data flow*. The host application running on the CPU uses the run-time environment to stream the data flow to the AP. On the AP, one byte from the data flow is broadcast to all the processing elements in every clock cycle. Henceforth, the data byte is called a *symbol* and the clock cycle is called a *symbol-cycle*. 128 million symbols can be processed per second, giving rise to a processing rate of 1 Gbps.

At the beginning of the first symbol-cycle, only the processing elements marked as *start-elements* are *active*. Each start-element processes the first symbol in the data flow. If a *match* occurs, then the start element activates all elements connected to its outgoing lines. All elements which are activated in the current symbol-cycle, process the next symbol from the data flow in the next symbol-cycle, and the process continues.

If an element is programmed as a *reporting* element, then on a match, an *output event* is generated, wherein an *output vector* is recorded in an *output buffer*. The execution continues from the next symbol-cycle as long as there are symbols left in the data flow and at least one processing element is active.

Asynchronously, the run-time environment reads the output vector from the output buffer to the main memory, decodes it and presents the host application with the id(s) of the reporting element(s) which generated the output and the offset in the data flow where the output event occurred. In case, the input patterns were defined as regexes, id(s) of the matched regex(es) are returned instead of the id(s) of the reporting element(s).

2.1.3 Automata Elements

The PCRE syntax is well known to the programming community. However, the ANML language is new. Therefore, we briefly introduce it here.

The processing elements in ANML are very similar to the classical *state-diagram* representation. In a state-diagram, the *states* are depicted using circles and transitions between the states are drawn as directed edges. A state is identified using state-labels, which are placed inside the circle. In our examples, we have chosen integers to represent state-labels. Similarly, each edge is accompanied by an edge-label which appears above the arrow depicting the edge. The edge-label could be a symbol, or a set of symbols on which the transition occurs. Such a set is called a *character-class*. A *start state* is depicted using a short incoming arrow (with no source), and an *accept state* is shown by marking the boundary of the circle with double lines.

In the beginning, only the start state is active. The symbols of the search string are processed one by one. If a state is active and the symbol to be processed is present in the edge-label of any of its outgoing edge, then the state at the end of that edge is made active for the next symbol. The process continues till either all the states are inactive or one of the accept states becomes active. In the latter case, the string is accepted by the automaton.

Thus, traversing the state-diagram from a start state to an accept state while concatenating the edge labels on the way defines a string accepted by the automaton.

The set of all the strings accepted by an automaton is called its *language*, and the set of all the symbols appearing in these strings is called its *alphabet*. For example, the automaton whose state-diagram is shown in Figure 1 accepts the string *bb* while traversing the states $1, 2$ and 5. Its language is $\{aab, bab, bb\}$, and its alphabet is $\{a, b\}$. The alphabet of all the automata discussed throughout the rest of this chapter is $\{a, b\}$.

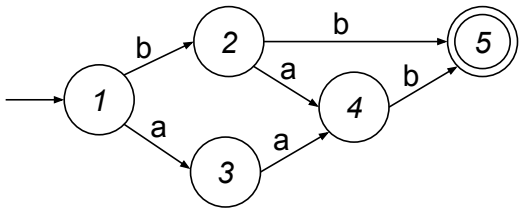


Figure 1: State-diagram of automaton to accept strings from the language $\{aab, bab, bb\}$.

In ANML, an edge transition is programmed as a *State Transition Element* (STE). An STE is represented using a circle with the edge-label placed inside it. States in the classical NFA are captured using connections between the STEs. Each STE representing an incoming edge into a state is connected to all STEs representing the outgoing edges of that state. Every STE representing an outgoing edge of the start state is marked as a *start STE*. A start STE is represented by a circle with a triangle attached to the top. Similarly, STEs representing incoming edges into an accept state are marked as *reporting STEs*. The boundary of a reporting STE is marked with double lines. The schematic representation of the ANML-NFA equivalent to the state-diagram shown in Figure 1, is presented in Figure 2.

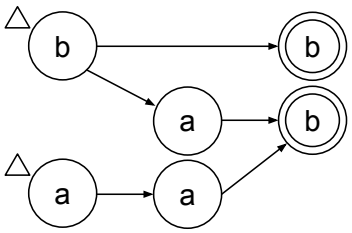


Figure 2: Equivalent ANML-NFA for classical NFA shown in Figure 1.

The processing of a search string by the ANML-NFA can be described as follows. The symbols are processed one by one, one symbol per *symbol-cycle*. The start STE(s) are active on the first symbol-cycle. During a symbol-cycle, active STEs check the symbol against its label, and if a match occurs, activates all STEs on its outgoing edges. If a reporting STE is matched, then the string is accepted by the ANML-NFA and an output is generated.

Besides STEs, ANML-NFA can contain special processing elements called *counter* elements and *boolean* elements. These elements do not process any symbol from the data stream. Neither do they consume a symbol-cycle to complete their processing.

A counter-element is activated when STEs connected to its input line are matched in a symbol-cycle. On being activated, the counter increments the value stored in an internal *count* register by one. If the count value reaches a value stored in a *target* register, then the processing elements connected to its output line are activated for the next symbol-cycle. The target value is a 12-bit number and up to four counter elements can be cascaded together to provide a 48-bit counter.

Boolean elements can be used to emulate multiple input *AND, OR, NOT, NAND, NOR, sum-of-product* and *product-of-sum* gates. If the incoming lines of a boolean element are simultaneously activated in such a way that the boolean logic is satisfied, then the STEs connected to the outgoing lines are activated for the next symbol-cycle. Due to timing restrictions, a boolean element cannot be connected to another boolean element, and a counter element can only be connected to its input line.

These elements can be used for the compaction of automata, as well as defining automata which represent languages which are not regular. This has been discussed in detail in the next chapter.

2.2 Programming Resources

A single AP chip contains 49,152 STEs, 768 counter elements, and 2,304 boolean elements arranged hierarchically into *rows, blocks*, and *half-cores*. The physical routing capability reduces as we move up the hierarchy. All 16 STEs within a single *row* can be connected to each other and to the boolean or counter element in the row. Every fourth row contains a counter element and the other three rows contain a boolean element each. However, elements from different rows need to be connected using *block routing* lines, 24 of which are shared by the 16 rows in a block. The connectivity between STEs from 96 blocks in a half-core is even more limited and is therefore avoided in our automata designs. STEs from different half-cores cannot be connected to each other, thus setting the upper limit on the size of an automaton to be the size of a half-core. The hierarchical layout of processing elements in an AP chip is shown in Figure 3.

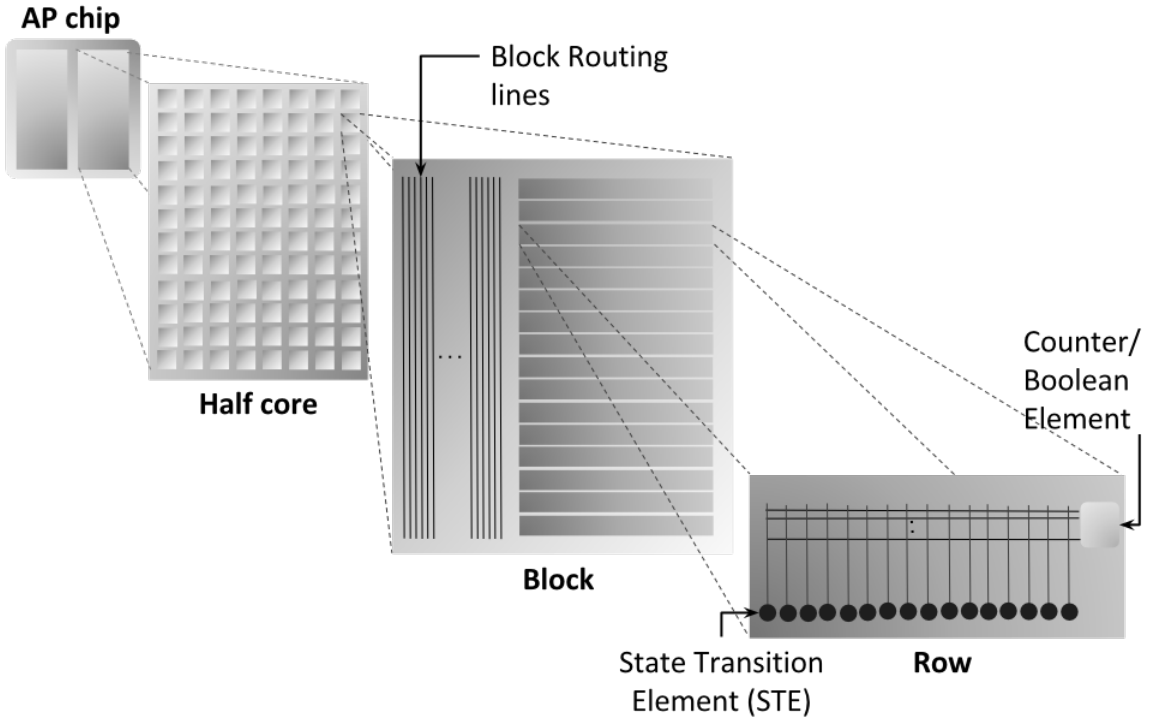


Figure 3: Hierarchical layout of processing elements in an AP chip.

However, an AP chip does not interface with the host-processor directly. Multiple AP

chips are arranged into an Automata Processor board (AP board) which is then connected to the host-processor. Currently, development AP boards come in three form factors.

The smallest AP board contains two AP chips and connects to the host-processor over the Universal Serial Bus (USB) interface. The two AP chips are organized into a single *rank*. The rank contains a high-speed intra-rank bus which allows two separate data flows to be streamed to the two AP chips in parallel, or a common data flow to be broadcast to both the AP chips. In the latter case, the two AP chips are said to constitute a *logical core*. This provides the flexibility of either having a higher processing rate of 2 Gbps or the capability to search for a larger number of patterns against a single data flow at 1 Gbps.

The second AP board contains 32 AP chips and is connected with the host-processor using a high-speed Peripheral Component Interconnect Express (PCIe) interface. The chips are organized into 4 ranks containing 8 chips each. Similar to the previous example, all the chips in the single rank can be organized into a single logical core or into multiple logical cores of 1, 2, 4 or 8 chips. This is also done to provide flexibility. If a large number of patterns need to be programmed, and they do not fit inside a single AP chip, then a logical core with up to 8 AP chips within a rank can be used. However, in this case, the data processing rate falls to 1 Gbps for the entire rank. On the other hand, if all the patterns can be fit inside a single chip, then they can be replicated on all 8 chips on the rank 8 different streams can be processed in parallel giving a processing rate of 8 Gbps. Using the 4 ranks on the board, a large number of patterns can be handled at a cumulative processing rate of between 1 Gbps(one logical core in a single rank) and 32 Gbps (eight logical cores per rank).

The third AP board is the largest of the three and contains 48 AP chips organized into 6 ranks. It is also connected to the host-processor using a PCIe interface. Cumulatively, this AP board contains $2,359,296$ STEs, $36,864$ counter elements and $110,592$ boolean elements. This is sufficient to accommodate thousands of patterns at the same time as we shall see in the next two chapters.

Chapter III

GENERATING EFFICIENT ANML-AUTOMATA

Having understood the processing elements in the AP, let us now look at how to convert a classical NFA expressed as a state-diagram to an *ANML-NFA*, i.e. an NFA defined in ANML. A simple method to do this was introduced in the previous chapter. This method can be extended to handle state-diagrams with even ε-transitions. The ANML-NFA derived in this way may be optimized further, which we shall see in this chapter.

Firstly, the difference in representation brings about redundancies in the ANML-NFA. For example, even if the starting state-diagram is optimal, the resultant ANML-NFA may not be so. We need to identify these redundancies and remove the same.

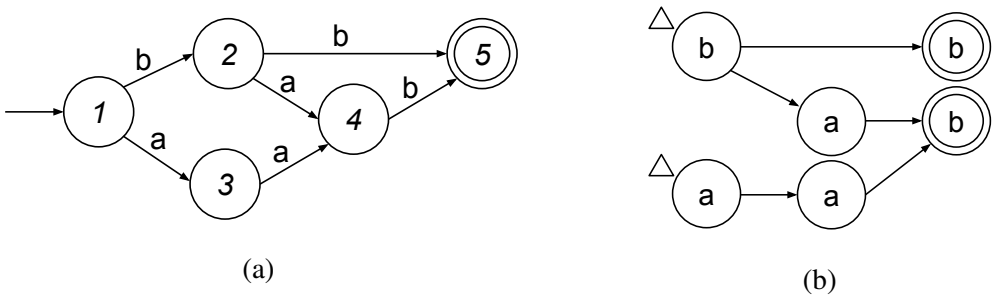
Secondly, The counter and boolean elements have no equivalents in the classical state-diagram representation. This not only provides means to compress the ANML-NFA further, but also allows the expression of languages which are richer than the regular languages represented by classical regular expressions and NFAs.

Finally, the AP architecture has its own set of characteristics like different run-times for different operations, hardware bottlenecks, and hierarchical organizations. These characteristics place some restrictions on design of efficient ANML-NFA. We will conclude this chapter with general guidelines to design automata in adherence to the above characteristics and avoid performance bottlenecks.

3.1 Translating classical NFAs to ANML-NFAs

Let us revisit the simple automaton and translation technique discussed in the previous chapter. The classical and ANML representations have been reproduced in Figure 4. Each state-transition (edge) in the state-diagram is represented using an STE in the ANML-NFA. The label of the edge is programmed as the label of the STE, and shown inside the

circle representing the STE. States in the state-diagram are captured using the connections between the STEs. Each STE representing an incoming edge into a state is connected to all STEs representing the outgoing edges of that state. Every STE representing an outgoing edge of the start state is marked as a start-STE. Similarly, STEs representing incoming edges into an accept state are marked as reporting STEs. The start-STEs are delineated with a triangle at the top of the STE and perimeter of reporting STEs are demarcated using double lines.



(a) (b)

Figure 4: Classical and ANML representation of automaton to accept strings over the alphabet $\{a,b\}$ and from the language $\{aab, bab, bb\}$.

3.1.1 Handling ε-transitions

In a classical NFA, an ε-transition is defined as a spontaneous transition from one state to another without consuming any symbol from the input string. An ε-transition is depicted in a state-diagram by an edge with the ε label. For example, in Figure 5, transitions from state 1 to state 2 and state 2 to state 4 are ε-transitions.

Handling ε-transitions out of a start state was introduced in [25]. Here, we define a general technique to handle any ε-transition using the concept of ε-*closure*. The ε-closure of a state is defined as the set of states, containing itself and all the other states in the NFA which can be reached from that state by following only ε-transitions. For example, the ε-closure of state 1, depicted as $\Upsilon(1)$, is $\{1,2\}$. Similarly, $\Upsilon(2) = \{2\}$ and $\Upsilon(4) = \{4,5\}$.

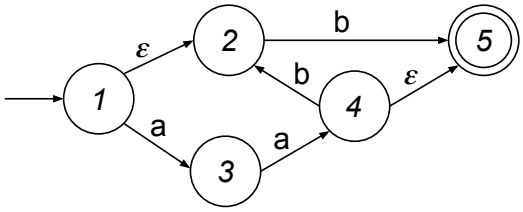


Figure 5: State-diagram of an automaton with ε-transitions.

Our technique to convert classical NFA to ANML-NFA can be extended to handle ε-transitions as follows. First, the ε-closure of every state is computed. Then, each STE representing an incoming edge into a state u should be connected to all STEs representing outgoing edges of a state v, where $v \in \Upsilon(u)$. If u is a start state, then all the STEs representing outgoing edges of v are marked as start STEs, $\forall v \in \Upsilon(u)$. Similarly, if $\exists v \in \Upsilon(u)$, where v is an accept state, then all the STEs representing incoming edges of u are marked as reporting STEs. The equivalent ANML-NFA to the state diagram shown in Figure 5, is shown in Figure 6.

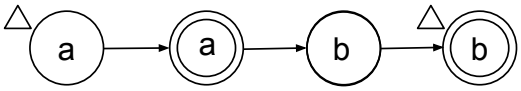


Figure 6: Equivalent ANML-NFA of for classical NFA shown in Figure 5.

3.1.2 Removing Redundancies in ANML-NFA

3.1.2.1 All-input-start STE

The start STE discussed until now is active only during the first symbol of the string. It is called *start-of-data* STE. However, the architecture allows a start STE to be configured as an *all-input-start* STE, which is active on every symbol of the input string. Schematically, such a start-state is depicted using a solid triangle. For example, $STE1$ and $STE2$ in Figure 7b represent all-input-start STEs.

All-input-start STEs can be used to minimize the ANML-NFA by identifying STEs which are active in every symbol-cycle. Consider the ANML-NFA shown in Figure 7a.

First, identify every start-of-data STE which has a loop to itself and whose label contains all the symbols in the alphabet (represented by $*$). Such an STE is active on every symbol-cycle, and is therefore converted to an all-input-start STE. Next, consider all the start-of-data STEs connected to the outgoing edges of these all-input-start STEs. These STEs are also active on every symbol-cycle and are converted to all-input-start STEs. If any of these new all-input-start STEs also have the label $*$, then the process continues in iterations until no new all-input-start STEs are found. AS a final step, all the incoming edges into any all-input-start STE are removed, as they are redundant. The optimized ANML-NFA corresponding to the one shown in Figure 7a, is shown in Figure 7b.

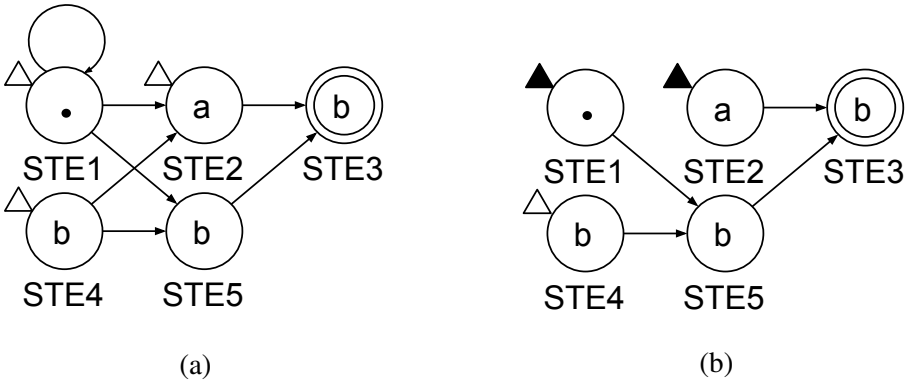
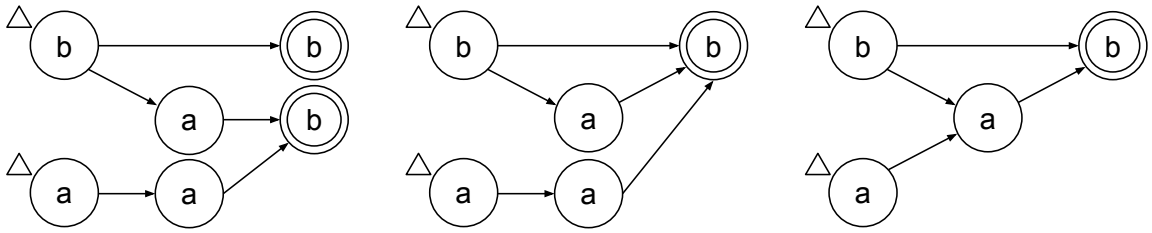


Figure 7: Modifying ANML-NFA using *all-input-start* STE.

3.1.2.2 Minimizing the Number of STEs

The automaton shown in Figure 1 cannot be represented by any state-diagram having less than six edges. However, its equivalent ANML-NFA, shown in Figure 8a, containing six STEs is not optimal! The optimal ANML-NFA contains four STEs as shown in Figure 8c. Amazingly, there is no way to translate the four STEs in this ANML-NFA into a classical state-diagram with four edges. This is because two of the STEs in the optimal ANML-NFA represent two edges each in the classical automaton.

The following steps may be used to identify and remove redundant STEs in an ANML-NFA:



(a) Original (b) Intermediate (c) Optimized

Figure 8: Optimizing ANML-NFA to accept the words from the language $\{aab, bab, bb\}$.

1. Identify all non-reporting STEs with no outgoing edges to any other STE. Such an STE cannot reach any reporting STE and should be removed along with all its incoming edges.

2. Any remaining STEs with no outgoing edges to other STEs must be reporting STEs. Merge all such STEs with the same label into a single reporting STE. For example, both the reporting STEs in Figure 2 can be merged into a single reporting STE as shown in Figure 8b.

3. Starting at a reporting STE, traverse backwards (following incoming edges in the reverse direction) in a breadth first fashion. For any STE, if two or more incoming edges originate from STEs with the same label, that do not have any other outgoing edges, then these edges should be merged along with the originating STEs. For example, in Figure 8c, the two STEs with label *a* connected to the reporting STE are merged into a single STE.

4. Repeat the above step for all reporting STEs with no outgoing edges.

It is important to remember that an all-input-start STE is different from a start-of-data STE and therefore can not be merged even if they have same label and satisfy all the other conditions mentioned in the steps above.

3.2 Counter Elements

A counter element can be used to generate compact automata whenever a pattern contains a repeating sub-pattern. For example, the ANML-NFA shown in Figure 9 checks for substrings wherein the sub-pattern *ab* repeats 50 times consecutively followed by the sub-pattern *aa*, i.e. substrings accepted by the regular expression $(ab)\{50\}aa$. The equivalent ANML-NFA using a counter element is shown in Figure 10.

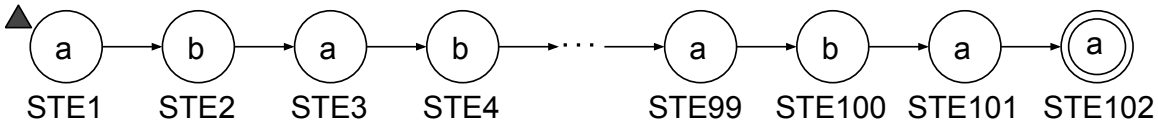


Figure 9: ANML-NFA to identify substrings accepted by the regular expression *(ab)* $\{50\}aa$.

A counter element contains a programmable 12-bit *target-value*, two input lines namely *count-line* and *reset-line*, and an *outgoing-line*. The outgoing-line can be configured to generate a *pulsed* or *latched* output. In our diagrams, a counter element is represented as a rectangle. The target-value is shown in a solid rectangle inside this rectangle. The count-line and the reset-line are shown as triangles on the left boundary, inscribed with the characters *C* and *R* respectively. The outgoing-line on the right is accompanied with the ⎍ symbol or the ⎍ symbol to denote the pulsed-output or the latched-output respectively.

The counter element functions as follows. The programmer sets the target-value for the counter based on the automata design. The counter element contains a *counter-value* (not shown in the ANML-NFA) which stores the current value of the counter at run-time. At the beginning of execution, the counter-value is set to 0. If during a symbol-cycle, the count-line is activated, then the counter-value is incremented by 1. However, if the reset-line is activated, then the counter-value is reset to 0. If none of the input lines are activated, then the counter-value remains unaltered. When the counter-value equals the target-value, the counter element activates its outgoing-line. If the counter element is programmed to

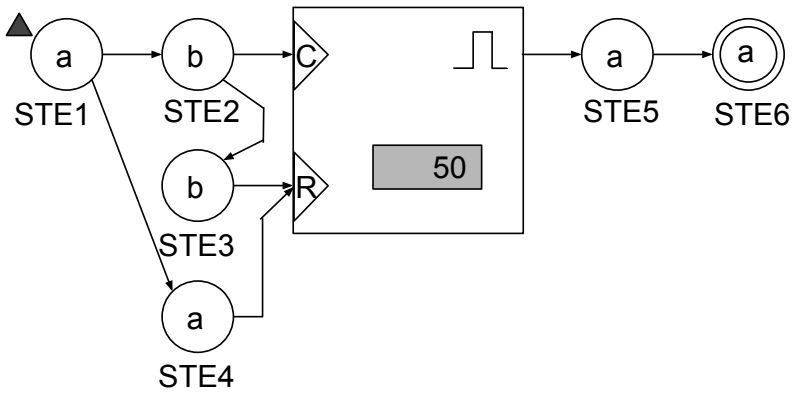


Figure 10: Using a counter element with *pulsed-output* to identify substrings matching the regular expression *(ab){50}aa*.

generate a pulsed-output, the outgoing-line is activated only for the next symbol-cycle. However, if it is programmed to generate a latched-output, the outgoing-line is activated until either the counter element is reset or the end of the data-flow is encountered.

The ANML-NFA shown in Figure 10 searches for substrings which are accepted by the regular expression $(ab)\{50\}aa$. The occurrence of the pattern *ab* in the string increases the count-value by one by activating the count-line. However, the occurrence of the patterns *aa* or *bb* results in the count-value being reset to 0 by activating the reset-line. Therefore, only when the pattern *ab* occurs 50 times consecutively, the counter-value reaches the target-value and the outgoing-line is activated.

Since the counter element is programmed to generate a pulsed-output, $STE5$ attached to its outgoing-line is activated only for the next symbol cycle. This means $STE6$ reports only if the pattern *aa* follows exactly 50 consecutive repetitions of the pattern *ab*, and no more. If it is desired that the pattern *aa* occurs after at least 50 consecutive repetitions of the pattern *ab*, then the counter can be programmed with a latched-output as shown in Figure 11. In this case, $STE5$ is activated as long as the pattern *ab* continues to repeat consecutively beyond the 50^{th} reiteration, i.e. the counter is not reset. Hence, this automaton finds substrings which match the regular expression $(ab)\{50,\}aa$.

If both lower and upper bounds are desired on the number of repetitions, then two

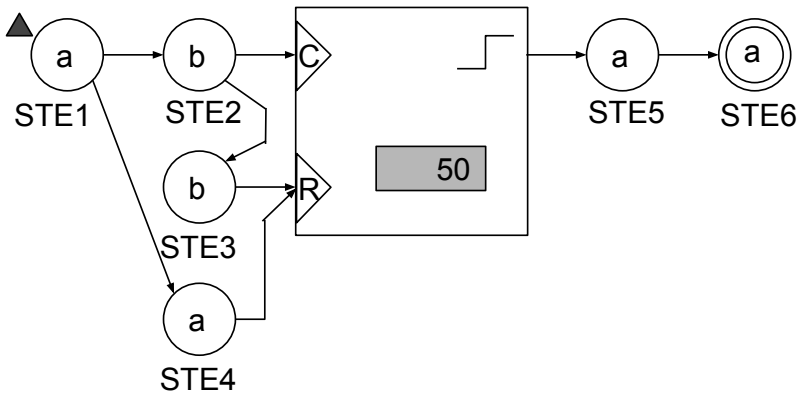


Figure 11: Using a counter element with *latched-output* to identify substrings matching the regular expression *(ab){50, }aa*.

counter elements can be used as shown in Figure 12. The counter element $CTR1$ takes care of the lower bound (of 50), whereas $CTR2$ manages the upper bound (of 100). $CTR1$ activates $STE5$ on every symbol-cycle after the 50^{th} consecutive repetition of the pattern *ab* is found. However, the output of $CTR2$ resets $CTR1$ as soon as the 100^{th} consecutive repetition is encountered, and continues to do so until both $CTR1$ and $CTR2$ are reset or the end of the data-flow. Hence, this ANML-NFA reports substrings which are matched by the regular expression $(ab)\{50, 100\}aa$.

In all the above examples, only back-to-back repetitions allow the counter-value to be incremented to the target-value. However, the counter-elements themselves do not place any such restrictions. For example, if the counter element in Figure 11 is never reset, then $STE6$ would report every occurrence of the pattern *aa* after 50 occurrences of the pattern *ab* have been found. In Section 3.4, we discuss some interesting uses of such a capability.

Similar to an STE, any counter element can also be programmed to be a reporting element. Additionally, it is redundant to have more than one STE with the same label connected to the same input line of a counter-element. Therefore, the methods described in Section 3.1.2.2 to remove all redundant STEs starting with reporting STEs with the same label can also be applied to remove the redundancy in this case.

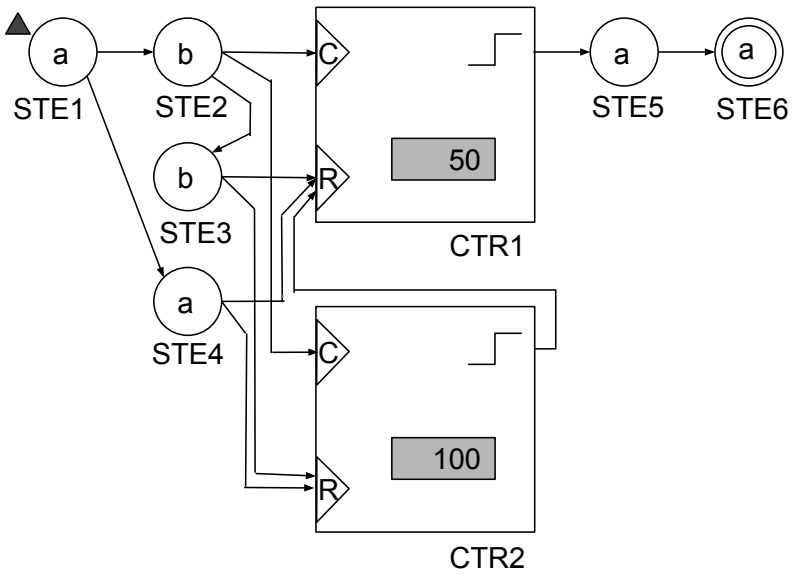


Figure 12: Using two counter elements to identify substrings matching the regular expression *(ab){50,100}aa*.

3.3 Boolean Elements

3.3.1 Group-of-Two (GoT)

All STEs in the AP are physically organized in *Group of Twos* (GoTs) which share an *OR* gate and a 3-input multiplexer with a programmable select signal. A GoT is shown in Figure 13 along with all its routing lines. The programmable routing lines are depicted by solid lines.

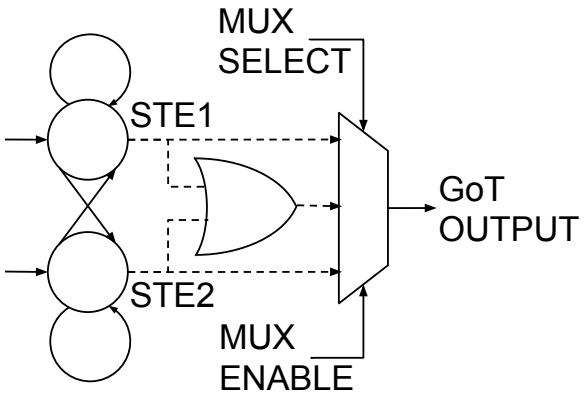
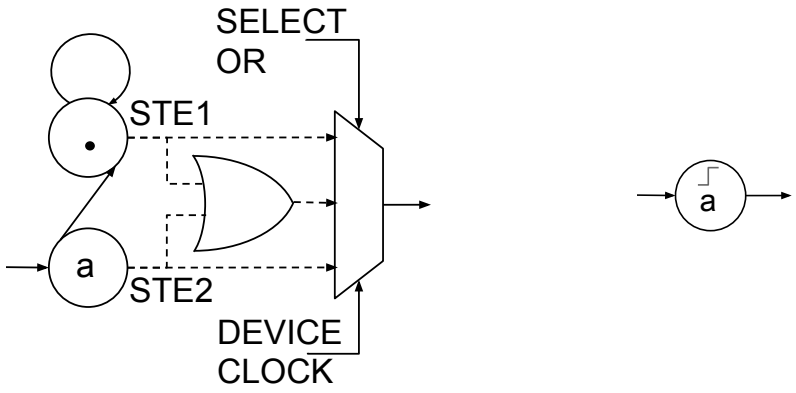


Figure 13: Group of Two (GoT) STEs.

The *OR* gate in the GoT structure allows it to be configured as a *latched-STE*. A latched-STE activates the outgoing line of the GoT on every symbol-cycle subsequent to the one in which $STE2$ is matched. Figure 14a shows a GoT programmed to emulate a latched-STE programmed with the symbol a. For succinctness, henceforth in this paper, a latched-STE is represented using the symbol shown in Figure 14b.

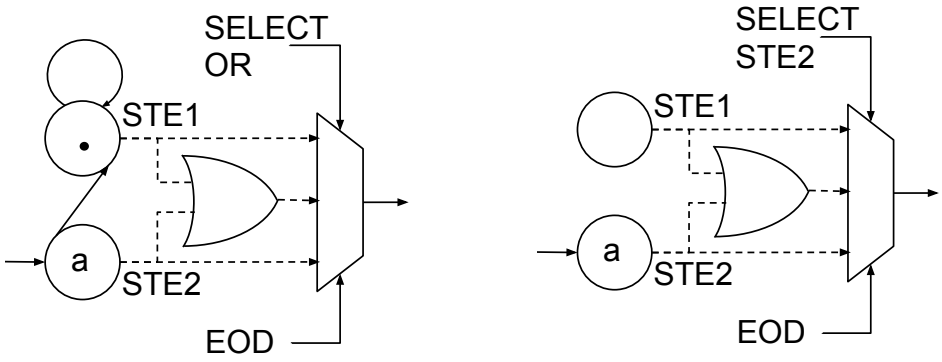


(a) Implementation. (b) Symbol.

Figure 14: *Latched-STE* with label a.

The multiplexer-select and enable lines provide further flexibility. For example, consider the programming of a GoT shown in Figure 15a. By default, the enable line is connected to the device clock of the AP. However, it can be connected to a special *End-Of-Data* (EOD) signal which can be passed at specific symbol-cycles in the data-flow. Therefore, even if $STE2$ is activated and matched on any symbol-cycle, the downstream pattern matching continues only after the symbol-cycle where the EOD signal is asserted. If the matching of the label in $STE2$ has to be anchored at a symbol accompanied with the EOD signal, then the connections can be made as shown in Figure 15b with the multiplexer select line choosing the output from $STE2$.

If $STE2$ in the above examples is a reporting STE, the output is generated only on symbol cycles when the EOD signal is asserted. This decreases the rate of output generation at the cost of losing information on the exact location in the input data where the match



(a) Remember match and activate on EOD signal.

(b) Match only if symbol is accompanied with EOD symbol

Figure 15: Using the *select* and *enable* lines of the GoT for added flexibility.

occurred. This is useful in driving performance as discussed in Section 3.6.2.

The connections in the GoTs described above need not be explicitly defined by the user. All these features: latched/unlatched, anchored/un-anchored, reporting/non-reporting are abstracted as attributes of STEs in ANML. However, if the programmer uses the EOD signals in his/her automata designs, then he/she has to specify the symbol-cycles in which the EOD signal must be asserted, because by default it is not asserted on any symbol-cycle.

Other than the *OR* gates in the GoTs, the AP has dedicated 16-input boolean elements which can perform the *OR*, *AND*, *NOR*, *NAND*, *sum-of-product* and *product-of-sum* boolean operations. The outgoing-line of a boolean element is activated if the input lines are activated in a combination (0 for inactive and 1 for active) which satisfies the boolean operation of the element. Like all other processing elements in AP, boolean elements can also be programmed as reporting elements.

3.4 Handling Look-ahead *Assertions*

Look-ahead is an assertion defined in many modern pattern-recognition libraries such as Perl Compatible Regular Expressions (PCRE) [3] which allows the matching of a pattern in the subsequent part of the string without consuming any symbols. This allows defining

patterns succinctly which otherwise would be very complex to represent using a regular expression or classical NFA.

3.4.1 Example 1

Consider a language of strings which contain all the substrings *aa*, *ab*, *ba* and *bb*. In PCRE, this can be written as `(?=.*aa)(?=.*ab)(?=.*ba)(.*bb)`. It can be interpreted as follows: Starting at the beginning of the string, look-ahead in the string for the first occurrence of the pattern *aa*. If *aa* is found, then look-ahead for the first occurrence of the pattern *ab*. If *ab* is found, then repeat the same for the pattern *ba*. If all the three are found, then start consuming symbols to find the first occurrence of the pattern *bb*.

Figure 16 shows the equivalent ANML-NFA using a boolean *AND* element. It works as follows: $STE1$ and $STE2$ check for the pattern *aa* in the string. Since $STE1$ is an all-input start STE, the search may begin at any offset in the string. Once the pattern is found, $STE2$ which is a latched STE keeps activating the first input line of the boolean *AND* element. Similarly, $STE3$ and $STE4$ look for the pattern *ab* and so on. When all the patterns *aa*, *ab*, *ba* and *bb* have been found, all the input lines to the boolean AND element become active. The AND function is satisfied and the boolean element being a reporting element reports this occurrence.

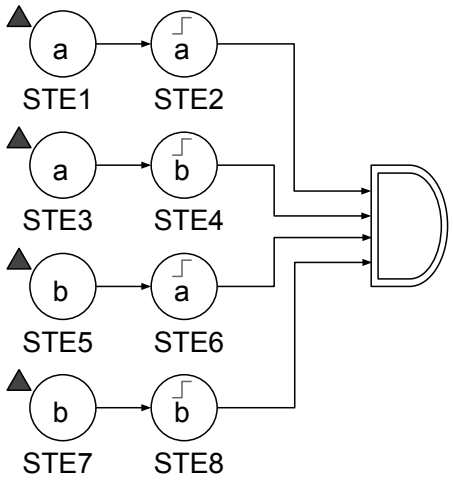


Figure 16: Handling look-ahead assertions using a boolean AND element.

3.4.2 Example 2

Though the above PCRE and ANML-NFA accept the same strings, they differ in the number of symbols consumed before the match is reported. In case of the PCRE, it is the length of the shortest prefix of the string ending with the (last) pattern *bb*. For the ANML-NFA, this number is the length of the shortest prefix containing all the four patterns.

Consider the PCRE which looks for the pattern *a* after the lookahead assertions have been satisfied, i.e. `((?=.*aa)(?=.*ab)(?=.*ba)(.*bb))a`. This can be represented by an equivalent ANML-NFA as shown in Figure 17. Since the search for the pattern *a* should immediately follow the matching of pattern *bb*, these two are concatenated and become the new input to the boolean AND element.

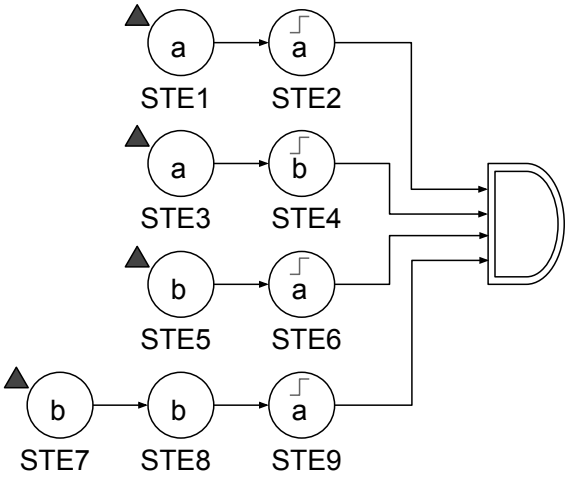


Figure 17: ANML-NFA accepting the same strings as the PCRE *((?=.*aa) (?=.*ab) (?=.*ba) (.*bb))a*.

3.5 Going Beyond Regular Languages

Counter and boolean elements provide the capability to remember information beyond the set of states which are active at any given point of time. The additional expressive power that this provides to ANML-NFA is a matter of ongoing research. For example, ANML-NFA with counter elements can emulate cellular automaton like `Rule 110` which is known

to be Turing-complete. However, such a discussion is outside the purview of this paper. However let us look at an example of an ANML-NFA which defines language that are not regular, i.e. language that cannot be expressed using any classical regular expression or NFA.

3.5.1 Example 3

Consider a language very similar to the languages defined in Examples 1 and 2. A strings in this language contain the symbol *a* after all the patterns *aa*, *ab*, *ba* and *bb* have appeared at least once in the string. This pattern cannot be expressed using look-aheads in PCRE, but can be easily represented by modifying the ANML-NFA shown in Figure 16 to the one shown in Figure 18. Once all the input lines to the boolean element are activated, the *AND* condition is met, and $STE9$ is activated to check if the next symbol is *a*. On all subsequent symbol-cycles also, $STE9$ is activated by the boolean element, as all its input lines originate from latched STEs which have already been matched. Therefore, as soon as the symbol *a* is encountered, a match is reported.

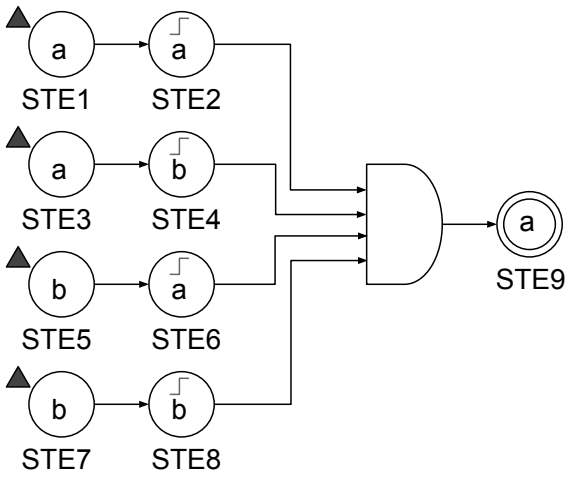


Figure 18: ANML-NFA which accepts strings which contain the symbol *a* after all the patterns *aa*, *ab*, *ba* and *bb* have appeared at least once in the string.

3.5.2 Example 4

Finally, consider the language consisting of strings where the patterns *aa* and *ba* occur at least 50 times each. Again, this language cannot be defined with regular expressions or even extended regular expression libraries like PCRE. However, the simple ANML-NFA for the same is shown in Figure 19.

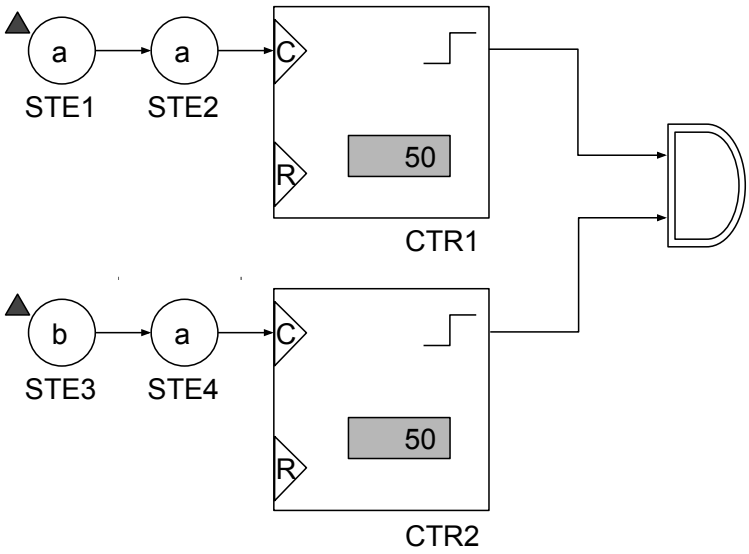


Figure 19: ANML-NFA which accepts strings where the patterns *aa* and *ba* occur at least 50 times each.

3.6 Application Developers Notes

The programming model on the AP is as follows. ANML-NFAs defined by the user are compiled into a machine readable image which is then loaded into the AP. Once the AP has been programmed with the loaded image, the input strings are organized into a data-flow which is then streamed to the AP. Whenever, a pattern is matched an output-vector is stored inside an internal buffer to be read out to the host application running on the CPU, where it can be decoded and suitably handled. In this section, we identify aspects of the above mentioned operations which may lead to performance bottlenecks and describe general guidelines for automata-design which minimize or avoid these bottlenecks.

3.6.1 Minimizing Reconfiguration Time

Compilation of ANML-NFA involves executing complex place-and-route algorithms to map elements of the user-defined ANML-NFA to physical elements on the Automata Processor board (AP board). This may take between seconds to hours based on the complexity and number of ANML-NFA to be compiled. Therefore, whenever possible, this step should be completed before-hand, and the images kept ready to be loaded at run-time. In this manner, the incurred cost at run-time is that of loading them which is in the order of 50 milliseconds.

Compilation of ANML-NFA ahead of time is possible for a variety of applications such as rule-based network intrusion detection or motif-based protein characterization. In these applications, the patterns to be looked for are known well in advance. For other applications, this might not be true. However, in some cases, although the patterns and their corresponding automata are not known ahead of time, all the required automata at run-time are copies of a few *template* ANML-NFA(s), where the copies of the same template differ only in the labels of STEs or target-values of counter elements. Such an application is defined in [62].

If the template ANML-NFA is known ahead of time, then it can be defined as an ANML-*macro*. The macro is compiled into an image which can be replicated easily throughout the AP board without running any place-and-route algorithms. Also, assigning the labels/target-values in all the copies of the macro can be done extremely fast. Therefore, the total time required to program the AP-board is the time required to generate the actual labels plus the time required to load the image onto the AP-board. As discussed earlier, the latter takes about 50 milliseconds. Another advantage of using macros is that they are modular in nature, making them reusable as building blocks for other ANML-NFA.

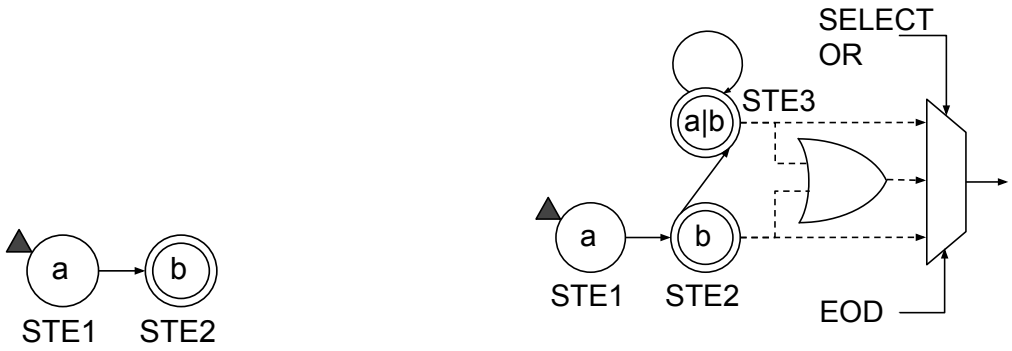
3.6.2 Avoiding Stalls to Handle Output

When one or more reporting STE(s) are matched in a symbol-cycle, an *output-vector* is stored in an internal *output-buffer*. Though storing this output-vector can be completed within the same symbol-cycle, reading this vector out of the buffer to the host application may take between 91 and 291 symbol-cycles. Although the buffer can hold 1024 output-vectors at any given time, if the vectors are generated too frequently, then the output-buffer fills up and the execution has to be stalled till the buffer can be emptied sufficiently. This can considerably slow-down the execution.

The output-vector is a bit-vector, whose length depends on the number of *output-regions* from which the reports are generated in that symbol-cycle rather than the number of reports themselves. Although, the programmer has no control over the mapping of reporting STEs to output-regions, substantial reduction in output-handling time can be affected by batching as many output-generating events into the same symbol-cycle as possible. For example, if the location of a match in the data-flow is not required, then output can be generated only at the end of the data-flow. This can be done by programming the reporting STEs to be latched and generating output only on receiving an EOD signal (as described in Section 3.3), and the EOD signal being asserted only on the last symbol in the data-flow.

Alternatively, if an approximate location of a match in the input string is desired, the modification of an automaton can be done as illustrated in Figure 20. $STE2$ of the original automaton shown in Figure 20a is modified to $STE2$ and $STE3$ of automaton shown in Figure 20b. $STE2$ and $STE3$ are part of a single GoT. A special character which lies outside the alphabet $\{a,b\}$ is required because EOD signal cannot be used to reset a latched STE. Hence, once a pattern is matched, it will be reported in all subsequent intervals. The special character is used to reset the latch. It is streamed at regular intervals in the data-flow. The EOD signal is asserted in the preceding symbol. The special character resets the latch by deactivating $STE2$ and $STE3$ in the modified automaton.

Since the special character is not part of the alphabet, the modified automaton does



(a) Original automaton.

(b) Modified automaton to report match at the end of an interval marked by an EOD signal and a special character.

Figure 20: Using a GoT to regulate output generation.

not have any active STEs at the beginning of the next interval other than the all-input-start $STE1$. Therefore, if an occurrence of the pattern spans across the interval boundary, then such an occurrence is not detected. Thus, the interval boundaries should be chosen at locations where such occurrences cannot occur or could be neglected. Otherwise, sufficient overlap between the end of an interval and the beginning of the next should be provided.

3.6.3 Maximizing Resource Utilization

As discussed in the previous chapter, the physical routing capability decreases as one goes higher up the hierarchy of rows, blocks and half-cores in an AP chip. In the our experience, even a naïve implementation of thousands of small patterns from large databases in the field of network intrusion detection and bioinformatics lead to very high resource utilization. This is because each automaton generally fits inside a single row, where the connectivity is not a issue. Even when the automaton spans multiple-rows, automaton generated from regular expressions generally have a linear chain-like shape which can be partitioned into rows with very few connections between elements from different rows.

However, efficiently mapping large automata containing hundreds of processing elements and interconnection lines, such as the ones discussed in [62] require more careful thought. As a rule of thumb, it is better to design automata which can be broken into small

clusters of highly connected STEs with very few inter-cluster connecting lines. Also, since the number of processing elements is very high, it may be desirable to design automata which duplicate processing elements to decrease the demands on routing. This may actually provide higher resource utilization by decongesting the interconnection bottleneck.

3.7 Summary

In this chapter, we have described how to convert any NFA to ANML-NFA. We have also discussed several optimization techniques to minimize the size of the ANML-NFA and avoid performance bottlenecks. Finally, we have shown how the boolean and counter elements can be used to describe ANML-NFA which may be difficult or impossible to define using regular expressions or classical state-diagrams.

Chapter IV

HIGH PERFORMANCE PATTERN MATCHING

By virtue of being able to execute a large set of NFAs on a single input data stream, the Automata Processor is naturally poised to provide hardware acceleration to applications which need support for finding all occurrences of thousands of complex string patterns in a data-stream. Examples of such applications include network intrusion detection and characterization of protein sequences using motifs.

The acceleration of these applications using existing accelerators like GPUs, special-purpose ASICs, and FPGAs have been widely studied. Some of the primary challenges to overcome towards achieving this acceleration include (1) broadcasting the same data-stream to all the string pattern-matching operations in parallel (2) executing NFA without state-space explosion and (3) supporting the search for thousands of patterns in parallel.

Among the reported solutions, GPU-based solutions enjoy better generality and clock-speeds over other accelerators. However, they struggle to handle execution divergence in its executing threads, i.e. if any of the threads makes a conditional jump which is different from the others, then all the executing threads are preempted, and a new set of threads are loaded. This is a serious impediment when threads are programmed to search for different patterns. Besides, state-space explosion in the case of handling NFAs continues to be a major hurdle. On the other hand, custom-made ASIC designs are either too specific or limited by their memory capacity and bandwidth requirements.

The best results thus far have been reported by solutions exploiting the reconfigurability and parallelism of FPGAs. Through the concurrent execution of multiple Non-deterministic Finite Automata (NFA) in hardware, significant speed-up is obtained and state-space explosion is avoided. However, even the largest FPGAs cannot fit beyond a few

hundred NFAs at a time. Therefore, applications involving thousands of patterns have to partition the same into subsets and search for their occurrences iteratively. This adds to the complexity and loss of performance.

We have implemented two applications from two different fields to study and demonstrate their acceleration using the AP. The automata developed for these applications illustrate the various design considerations to bear in mind to derive maximum performance benefits from the AP. Besides this, we have estimated the run-time of the applications using accurately known features such as on-board capacity, automata load time, data streaming rate and output handling throughput.

The first application, called Fast-SNAP (for Fast-SNort using AP), a rule-based Network Intrusion Detection (NID). These rules are obtained from the popular Snort database [4]. We developed techniques to transform a rule into one or more PCREs. However, some of these PCREs cannot be compiled by the AP compiler. In such cases, the rules are converted into ANML-NFA.

The second application is called PROTOMATA (for PROTein autOMATA). It looks for the presence of sequence-motifs in protein sequences. These motifs have been derived from the PROSITE database [66] and are originally defined in a proprietary syntax. Although, translating them into equivalent PCRE is relatively straight-forward, executing these PCRE would encounter performance bottlenecks for some use-cases of this application. Therefore, the motifs are instead represented as ANML-NFA, which include special additional parts to avoid these bottlenecks. PROTOMATA is described in detail in Section 4.2.

4.1 Fast-SNAP

In this section, we briefly discuss the state-of-the-art of Snort-based NID systems, followed by a brief introduction to Snort rules, and finally a detailed description of the Fast-SNAP application.

4.1.1 Background

Snort [61] describes signatures of anomalous activity in networks as string patterns and searches for occurrences of any of these patterns in network packets to detect an intrusion. The signatures are maintained in a database in the form of *rules*, one each for every known signature. The ruleset is updated as and when new signatures are discovered. The number of rules has seen a steady rise with the increase in the frequency and complexity of cyber-attacks. At the time of writing this paper, the ruleset contains $5,310$ active rules. In order to deal with increased network traffic and cyber-attacks, accelerated solutions using GPUs and FPGAs have received considerable attention in literature.

The first GPU-based solution [70] used DFA-based pattern-matching. This solution was found to be suboptimal as some patterns led to state-space explosion when converted to DFAs. Cascarano et al. [15] proposed the first NFA-based engine using GPUs which involved maintaining a global NFA transition table along with vectors for active and future states. They reported a maximum throughput of about 1.5 Gbps for *Snort534* ruleset from [12]. Zu et al. [79] noted that this approach suffered from the problem of serialization of threads because of execution divergence. They addressed the problem by identifying *compatible states*, i.e. states that cannot be active simultaneously, and then using these compatible states to create *virtual-NFAs* from the original NFAs . Using the virtual-NFAs, they reported a maximum throughput of nearly 13 Gbps for synthetic pattern sets generated with the workload generator from [9].

FPGA-based solutions rely on configuring the processor to concurrently execute multiple NFAs in hardware [53, 11, 55, 74, 72]. Yang et al. [74] used a modified version of McNaughton-Yamada algorithm to convert PCRE-based regexes to modular NFAs with multi-character transition labels. This allowed them to reach a maximum throughput of 10.3 Gbps. Mitra et al. [53] reported an interface throughput of 12.9 Gbps on the *SGI RASC RC 100* blade connected to *SGI ALTIX* supercomputing system, by transforming

PCRE op-codes generated by the Snort rules compiler to Very High Speed Integrated Circuit (VHSIC) Hardware Description Language code. However, the capacity of even the largest FPGAs is not enough to accommodate large rulesets [11, 55, 74], requiring them to be partitioned and programmed iteratively as multiple subsets.

The Snort rules database is updated as and when new signatures are discovered by a community of developers in the field. The number of rules has seen a steady rise with the increase in the frequency and complexity of cyber-attacks. At the time of writing this paper, the database contains $5,310$ active rules. Even the accelerated solutions discussed above are finding it difficult to scale with this ever-increasing rulesets and network bandwidth demands.

4.1.2 Snort Rules

Snort rules are written in a lightweight description language. Each rule contains a *header* section and an *options* section. The header section specifies the protocol, source, and destination of the network packet for which the rule is active, and the type of action to be taken if the rule is matched. The options section consists of one or more *keywords* belonging to one of the following categories: *general*, *non-payload detection*, *payload detection*, and *post-detection*. Detailed documentation of the Snort syntax can be found online [4].

General keywords provide common information about the rule such as an *sid*, which specifies the unique integer id of the rule. Non-payload detection keywords describe anomalous values for various fields in the header section of a network packet. On the other hand, payload detection keywords identify patterns to look for only in the data section of a network packet. Finally, the post-detection keywords specify actions to be taken if an occurrence of the signature is detected.

For example, consider the following sample rule:

```
alert tcp any any -> any 80 ( sid:42; content:"foo"; content:"bar";
distance:10; pcre:"/foo[0-9]{10}bar"; )
```

The header section of the above rule states that this rule is active for *tcp* packets going to port 80 and it raises an *alert* if all the patterns are matched in a packet. The options section of the rule lists four keywords, separated by semicolons. The first keyword specifies that the sid of the rule is 42. The following three keywords are of the payload detection type, on which we will focus for the rest of this section.

Most of the patterns in the SNORT rules are defined using payload detection keywords. These patterns either specify exact-strings (represented by the keyword `content`) or PCRE (represented by the keyword `pcre`). Our sample rule contains two exact-string patterns (namely `foo` and `bar`) and one PCRE pattern (namely `/foo[0-9]{10}bar`).

In addition to the keywords to specify patterns, a rule may contain *modifier* keywords. A modifier is associated with the previous pattern in the rule, and adds extra constraints on the matching of the pattern. For example, our sample rule contains the modifier `distance` which has an argument of 10. This modifier mandates that the occurrence of previous pattern (namely `bar`) and the one before that (namely `foo`) is separated by exactly 10 characters. Such modifiers placing distance constraints between the occurrence of on one or more patterns in a rule are henceforth referred to as *distance-modifiers*. Other modifiers restrict the search for occurrences of the previous pattern to a specific part of the payload. We call this kind of modifiers as *location-modifiers*. A rule is *triggered* if all the patterns defined in the rule are matched within the constraints specified by the modifiers in the input data-stream, which in the case of payload detection keywords, is the data-section of the network packet.

Notice that, in the sample rule, there is some redundancy in the exact-string patterns combined with their distance constraints and the PCRE pattern. More specifically, the latter accepts a subset of the strings accepted by the former. This is useful for existing solutions to avoid the execution of the computationally expensive PCRE-engine on the entire input

data set. Instead, the search is broken down into two stages. In the first stage, the data is *filtered* out using fast exact-string matching algorithms. Only, the packets suspected of having malicious content undergo Deep Packet Inspection (DPI) using the PCRE-engine in the second-stage. However, this convention is not universally followed by all rules in the SNORT database.

4.1.3 Methodology

4.1.3.1 Configuration Stage

About 83% of the active rules in SNORT can be efficiently implemented using the AP. The remaining 17% contain the keywords `byte_test`, `byte_jump`, or `byte_extract`. These rules extract various parameters for the pattern matching operations from specific `bytes` in the input data-stream. The implementation of these rules on the AP would would require reprogramming the device based on the input data. Therefore, the checking for these rules should be carried out using existing methods.

For each rule to be implemented in AP, we create PCRE or ANML-NFA using three steps. In the first step, we extract the non-payload detection and payload detection keywords from the rules and partition them into different *buckets* based on the location-modifiers. Within the same bucket, keywords from the same rule are combined into a single PCRE or ANML-NFA using steps 2 and 3. In step 2, the keywords from the same rule which are related to each other through distance constraints are combined into a single PCRE. Then in step 3, if more than one PCRE is created for a rule, then they are expressed as a single ANML-NFA using a boolean *AND* gate. These steps are described in detail below using an extended version of our sample rule.

```
alert tcp any any -> any 80 ( sid:42; content:"foo"; content:"bar";
distance:10; pcre:"/foo[0-9]{10}bar"; content:"kludge"; http_header;
content:"cluft"; http_header; content:"baz"; http_header;
content:"qux"; http_header; content:"abc"; http_uri; )
```

Step 1. Handling Location-Modifiers Searching for the occurrences of a pattern can be restricted to a particular section of the payload through the use of the location-modifiers. Additionally, location-modifiers can specify whether the pattern should be matched in the raw data or the normalized data.

In the first step, separate *buckets* are created corresponding to each location-modifier defined in SNORT. Then, for each rule, patterns qualified by different location-modifiers are placed in their respective buckets along with the `sid` of the rule. This allows us to program patterns from different buckets into different logical cores, and stream only the data relevant to the location-modifier to that logical core. For our sample rule, the patterns `foo`, `bar`, and `/foo[0-9]{10}bar` are placed in the *general* bucket; `kludge`, `cluft`, `baz`, and `qux` in the *http_header* bucket; and `abc` in the *http_uri* bucket.

Step 2. Handling Distance-Modifiers Distance-modifiers specify constraints on the location of the occurrence of a pattern in the data stream relative to an anchor. This anchor could either be the beginning of the stream or the end of the occurrence of the previous pattern. Keywords *offset* and *depth* are used to specify minimum and maximum distances from the beginning of the stream; whereas, keywords *distance* and *within* define minimum and maximum distances relative to the occurrence of the previous pattern.

In the second step, patterns within each bucket are considered separately. Patterns belonging to the same rule which are related through distance-constraints are combined into a single PCRE using *repetition quantifiers*. For example, patterns `foo` and `bar` in our sample rule are linked by a distance-modifier and belong to the general bucket. Hence, they

are combined to the PCRE: `foo.{10}.*bar`.

Step 3. Generating Final Regex or ANML-NFA After combining patterns into PCRE in the second step, multiple PCRE for a rule may be left in a bucket. All such PCREs should match in the data stream (in any order) for a rule to be triggered.

In the third and final step, if there is more than one regex for a rule in a bucket, then they are converted into ANML-NFA using a boolean *AND* element and latched-STEs as described in Section 3.4 of Chapter 3. For example, there are four patterns in the *http_header* bucket: `kludge`, `cluft`, `baz`, and `qux`. The ANML-NFA for the same is shown in Fig. 21. Notice that the classical NFA for this automaton is fairly complex and large because it needs to capture all the combinatorial ways in which the substrings `kludge`, `cluft`, `baz`, and `qux` may be ordered in the data-flow.

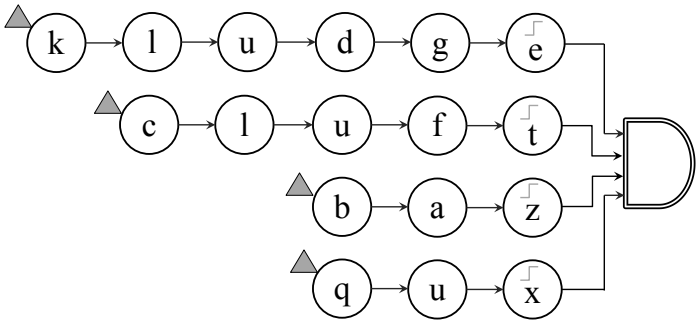


Figure 21: ANML-NFA corresponding to regex with 3 lookaheads: *(?=.*kludge) (?=.*cluft) (?=.*baz) .*(?:qux).*

Note: In the beginning, for the ease of implementation, we considered combining all the PCRE pertaining to a single rule from the same bucket into a unified PCRE using lookahead assertions. The definition of *lookahead* assertions may be found in the PCRE specification [3]. For example, the patterns in the *http_header* bucket can be expressed as a PCRE with three lookaheads: `(?=.*kludge)(?=.*cluft)(?=.*baz).*(?:qux)`. However, the AP compiler does not accept this PCRE as it has more than two lookahead assertions. This is because the ANML-NFA corresponding to PCRE with lookahead assertions

is very large. Instead, the ANML-NFA described above is considerably smaller.

At the end of these steps, in every bucket, there is at most one pattern (regex or ANML-NFA) per rule. Automata from each bucket can be compiled into a single image which can be loaded into a separate logical core at run-time. However, some buckets may contain very few automata or correspond to segments in the network packet which are very short. Automata from such buckets are combined and a single composite image is generated to be loaded into the same logical core.

4.1.3.2 Execution Stage

Since the automata are already compiled, the images are directly loaded into the AP board and the processing of the network packets begins instantaneously. The host application breaks a network packet into different segments and generates the data-flow corresponding to each logical core on the AP board. The `sid` of a rule is reported whenever the corresponding automata detects an occurrence in a network packet. In case, a rule is programmed as multiple automata, pertaining to keywords with different location modifiers, then the host application triggers the necessary action(s) specified in the original SNORT rule, only if, all the constituent automata generate a report within the same network packet.

Even after loading the automata for all the SNORT rules, significant portions of a AP board remain unused. Therefore, the logical cores may be replicated and data from different network packets may be streamed to different logical cores in parallel. In this way, a very large number of automata is executed in parallel on the AP board and the implementation of a very high throughput DPI engine is realized.

4.2 Protomata

PROTOMATA scans for all occurrences of protein *motifs* in protein sequences. A motif is described as a small conserved region in the protein sequence which plays a biologically meaningful role in the behavior of the protein, such as defining a binding or catalytic site,

enzymatic activity, etc. PROTOMATA first downloads the list of all known protein motifs from the PROSITE database. Then, for each motif, an NFA-based *pattern-macro* is created which can identify the occurrence of the motif in any protein sequence. Finally, additional macros are added to this pattern-macro to provide various user functionality and performance benefits. Once these NFAs have been programmed into the processor, protein sequences can be streamed, and all occurrences of the motifs can be checked in parallel. In this section, we briefly describe the PROSITE pattern-motifs, the state-of-art of finding the occurrence of these motifs in protein sequences, and the PROTOMATA application in detail.

4.2.1 Background

PROSITE is a large annotated database of known protein motifs. The motifs are described as either *pattern-motifs* or *profile-motifs*. A pattern-motif is expressed using *PROSITE pattern notation* described in Section 4.2.1.1. On the other hand, profile-motifs use a weight-matrix-based method to calculate similarity. PROTOMATA only scans for occurrences of pattern-motifs, and henceforth in this chapter, the words 'pattern-motifs' and 'motifs' are used interchangeably. Currently, PROSITE has 1308 pattern-motifs.

ScanProsite [21] is an online tool provided by PROSITE to scan protein sequences for occurrences of motifs present in the database. The following modes of operations are supported:

- *Use case 1*: Protein sequences(s) can be submitted to be scanned against all the motifs in the PROSITE database.
- *Use case 2*: Motif(s) can be submitted to be scanned against a protein database (UniProtKB, PDB, or user-defined).
- *Use case 3*: Motif(s) and protein sequence(s) can be submitted to be scanned against each other.

A Perl-based version of the tool called *ps_scan* can be downloaded for execution on a local machine. Similar programs were developed by academic groups [31, 67, 46, 64, 35, 71, 22, 34, 32] and commercial companies [68] in the early 1990s. However, we could not trace these implementations and had to limit our comparative studies to *ps_scan* only.

4.2.1.1 PROSITE pattern notation

Each motif consists of a sequence of *pattern elements* separated by a concatenation symbol '$-$'. The pattern elements are defined as follows:

- A single character represents a single amino acid in accordance to the international standard [41].
- A list of characters within brackets (*character class*) denotes any one of the amino acids in the list. For example, $[ST]$ denotes either the amino acid *Serine* (represented by *S*) or the amino acid *Threonine* (represented by *T*).
- A list of characters within curly brackets (*complementary class*) denotes any amino acids but the ones listed in the curly brackets. For example, $\{ST\}$ denotes any amino acid that is neither *Serine* nor *Threonine*.
- The lower case letter *x* denotes any amino acid.
- The character $<$ at the beginning of the motif denotes that the match is anchored to the *N*-terminal (beginning) of the protein. For example, $< S$ denotes the amino acid *Serine* at the beginning of the sequence, and $[< S]$ denotes beginning of sequence or the amino acid *Serine*.
- The character $>$ at the end of the motif denotes that the match is restricted to the *C*-terminal (end) of a protein sequence. For example, $T >$ denotes the amino acid *Threonine* at the end of the sequence, and $[> T]$ denotes end of sequence or the amino acid *Threonine*.

- A single number within parenthesis denotes repetition of a pattern element. Two numbers within parenthesis denotes repetition within the range. For example, $S(3)$ represents the repetition of the amino acid *Serine* exactly 3 times, and $T(3,5)$ indicates the amino acid *Threonine* is repeated anywhere between 3 and 5 times.

- End of a motif is denoted by the period character '.' .

For example, Fig. 22 shows the occurrence of the PROSITE motif *PS*00430 in the *Lissencephaly-1 homolog (D3BUN1)* protein sequence.

4.2.2 Methodology

In order to generate the required automata, the database of motifs is downloaded periodically (or on user request) and all the pattern-motifs are extracted from the same. For each motif in this database, two ANML-NFAs are generated. The design of these NFAs allows enabling the search for this motif at run-time by including its id in a *preamble-sequence* streamed at the beginning of the data-flow. The first automaton called the *Locate-occurrence* automaton is used when the location of each occurrence of an enabled motif has to be reported. The second automaton called the *Global-match* automaton is used when the user provides multiple protein sequences and only motifs which occur in all the sequences need to be reported. This automaton generates a single match report at the end of the streaming of all the sequences, thereby reducing the frequency of output-generation

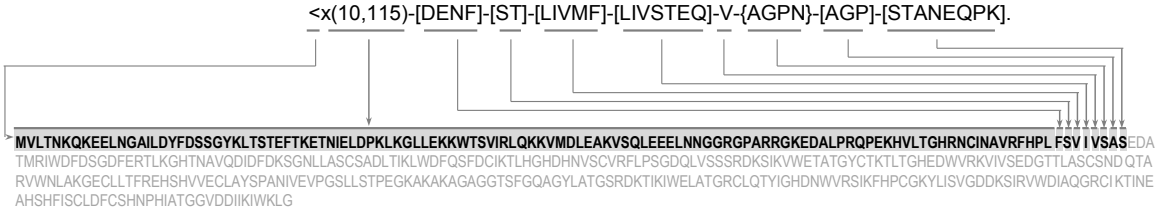


Figure 22: Occurrence of the PROSITE motif *PS*00430: *TonB-dependent receptor proteins signature 1* ($< x(10,115) - [DENF] - [ST] - [LIVMF] - [LIVSTEQ] - V - \{AGPN\} - [AGP] - [STANEQPK]$.) in the *Lissencephaly-1 homolog (D3BUN1)* protein sequence.

and increasing the overall performance. If the location of the occurrence of these common motif(s) within each sequence is also desired, then a second pass is used wherein the Locate-occurrence automata are programmed and only the search for the common motifs is enabled. We now describe both these designs in detail.

4.2.2.1 Automata Design

For the simplicity of expression, the STE labels in the diagrams are represented as follows:

- An upper-case letter represents an amino acid. An STE-label with one or more upper-case letters represents the character class containing the ASCII equivalent of the letters in the upper and lower cases. For example, the label *A* represents the character class $[41_{16}, 61_{16}]$ and label *AGV* represents the character class $[41_{16}, 47_{16}, 56_{16}, 61_{16}, 67_{16}, 76_{16}]$.
- $\sum$ represents the character class containing the ASCII equivalent of all letters representing amino acids in upper and lower cases.
- . represents the entire 8-bit symbol-set, i.e. $[00_{16} - ff_{16}]$.
- The letter *p* followed by a numeral represents a parameterized label.
- All other labels are 2-digit hexadecimal numbers.

Each motif is assigned a unique *PROTOMATA-id* which is a 2-byte (4-digit hexadecimal) number. This id is different from the PROSITE-id and a simple one-to-one mapping is maintained on the host application to provide the necessary interface between the user input (using the PROSITE-ids) and the working of PROTOMATA (using the PROTOMATA-ids).

A preamble-sequence is streamed at the beginning of the data-flow which contains the PROTOMATA-ids (both bytes) of only those motifs whose search has to be enabled. For

example, if the search for all the motifs has to be enabled, then the preamble-sequence contains the PROTOMATA-ids of all the motifs. On the other hand, if the search for *frequently-occurring motifs* has to be disabled, then the PROTOMATA-ids of only the rest of the motifs are included in the preamble-sequence. Thus the preamble-sequence is changed based on user input. However, preamble-sequences for standard choices are pre-calculated and stored. The end of the preamble-sequence is earmarked by the ff_{16} character. In order to avoid any confusion between this end-marker and the first (most significant) byte of a PROTOMATA id in the preamble-sequence, the range of the latter is restricted from 0_{16} to fe_{16}, i.e. the PROTOMATA-ids lie between 0_{16} and $feff_{16}$.

4.2.2.2 ANML Macro

If multiple ANML-NFAs use a common sub-automaton where the instances differ only in the labels of the STEs, then such a sub-automaton can be defined as a *macro*. The labels of the STEs which vary across the instances are parameterized to be defined at compile-time or before loading. STEs within macros can be marked as start and/or reporting STEs, and the processing elements inside the macro are connected to processing elements outside the macro through input and output terminals. Additionally, macros may be nested inside other macros. The use of these features of ANML macros to define automata in a modular manner is described in Section 4.2.2.1.

Enable Macro The Enable macro is part of both the Locate-occurrence and the Global-match automata. It processes the preamble-sequence and activates the subsequent processing elements only if the PROTOMATA-id of the associated motif is present in the preamble-sequence. This obviates the need to reprogram the AP chip every time the user enables the search for a different set of motifs.

The Enable macro is shown in Fig. 23. STE 1 and STE 3 is active for the first symbol in the preamble-sequence. STE 1 activates STE 2 for the second symbol-cycle. STE 1 and STE 3 are in turn activated by STE 2 for the third symbol-cycle. This process continues

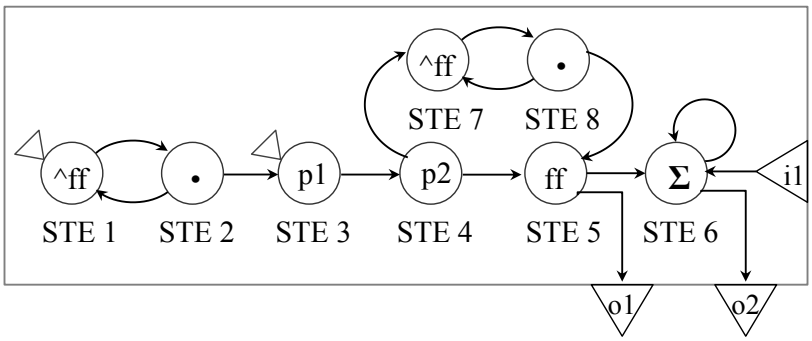


Figure 23: The *Enable* macro.

till the end of the preamble, i.e. STE 1 and STE 3 are active on every odd symbol-cycle and STE 2 is active on every odd symbol-cycle. STE 3 matches the first byte of every PROTOMATA-id in the preamble. If a match is found, then STE 3 activates STE 4 to check the second byte of the id. If both the bytes are matched then STE 5 is activated to indicate that the PROTOMATA-id was detected in the preamble-sequence. In order to remember this till the end of the preamble-sequence, STE 7 and 8 activate STE 5 on every even symbol-cycle. STE 5 is connected to the outgoing port which is used to enable the subsequent processing elements in the automaton.

The output port $o1$ is used to enable the search of the motif only at the first symbol of the protein sequence, whereas the port $o2$ is used to enable the search at every subsequent symbol. This feature allows the handling of motifs anchored to the beginning of the sequence (only $o1$ is used), and motifs which can occur anywhere in sequence (both $o1$ and $o2$ are used) with the same Enable macro. The utility of the input port $i1$ will be clear when we describe the *Repeat* macro. Notice that all instances of this macro for different motifs only differ in the labels of STE 3 and STE 4. Hence their labels are parameterized.

Pattern Macros Two pattern-macros called the *Report-on-match* macro and *Continue-on-match* macro are created for each motif. The first macro is used in the Locate-occurrence automaton shown in Fig. 24 and the second macro is used in the Global-match automaton shown in Fig. 25. Both these macros correspond to the same motif *PS*00430 shown in

Fig. 22, i.e.: $< x(10,115) - [DENF] - [ST] - [LIVMF] - [LIVSTEQ] - V - \{AGPN\} - [AGP] - [STANEQPK]$.

The Locate-occurrence automaton creates an output report as soon as the occurrence of a motif is detected in the data-flow (thereby marking the location of the occurrence). Therefore, STE 8 of the macro corresponding to the last pattern element in the motif is programmed as a reporting STE. On the other hand, in the Global-match automaton, STE 8 of the Continue-on-match macro is used to enable the Repeat macro if the occurrence of the motif is found in the protein sequence. The significance of this along with using two separate output ports $o1$ and $o2$ emanating from the STE 8 will become evident when we discuss Global-match automaton.

Locate-occurrence Automaton The PROTOMATA-id of the above mentioned motif is $18a_{16}$. Notice that, the parameters $p1$ and $p2$ of the Enable macro in Fig. 24 have been set to 1_{16} and $8a_{16}$ respectively. Furthermore, even though the motif is anchored to the beginning of the protein sequence, the output port $o2$ of the *enable* macro is connected to the input port $i1$ of the pattern macro. This is because the first pattern element $< x(10,115)$ needs to process up to the first 115 amino acids appearing in the protein sequence. Also note that the automaton is compilable even though the input port $i1$ of the Enable macro is left unconnected.

Global-match automaton In some use-cases, the desired output identifies only those motifs which are present in all the input protein sequences. A simple way to handle these use-case involves finding the motifs which occur in each individual protein sequence using the AP, and then identifying the common motifs using the host processor. The problem with this method is the rate of output generation on the AP may be quite high. Instead, the Global-match automaton generates a single output event at the end of streaming of the data-flow, identifying only the common motifs.

In order to do this, the data-flow is constructed as follows: Multiple sequences are

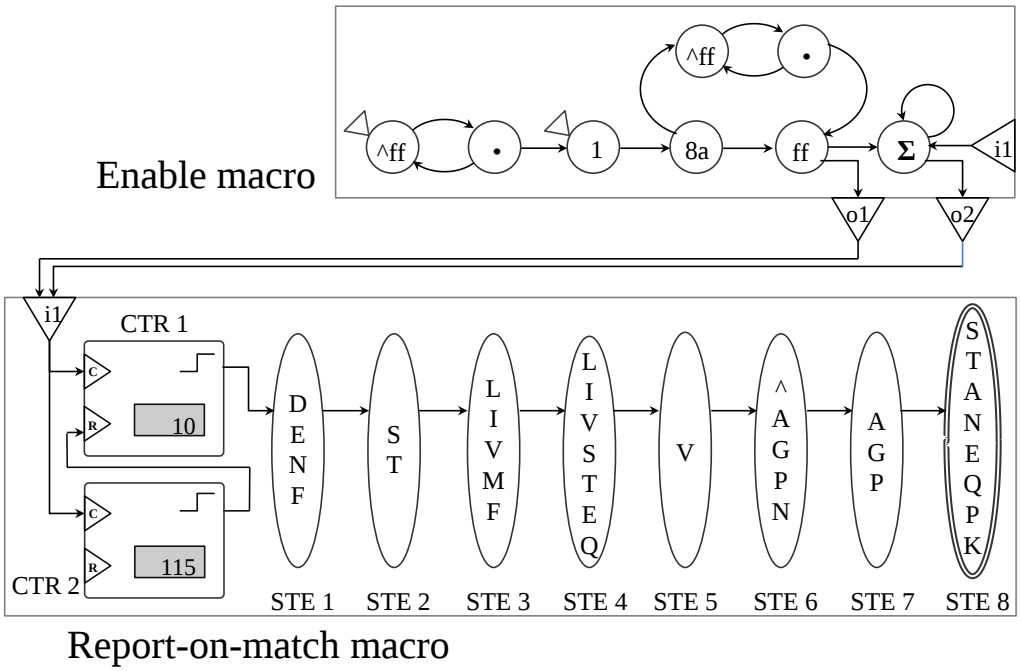


Figure 24: The *Locate-occurrence* automaton.

separated by a $0a_{16}$ character (ASCII equivalent of the *newline* character) and end of the data-flow is marked by the 0_{16} character. A motif common to all the protein sequences should be matched in all the sequences serially. This is done with the help of the Repeat macro shown in Fig. 25. STE 9 and STE 10 in this macro are activated once a motif is matched in a sequence. STE 9 then keeps itself and STE 10 activated till the $0a_{16}$ character marking the end of the sequence is encountered.

On matching the $0a_{16}$ character , STE 10 activates STE 11 and the Continue-on-match macro to check for the occurrence of the motif starting at the first character of the next protein sequence. If the motif need not be anchored to the beginning of the protein sequence, then STE 11 also activates the Enable macro which in turn activates the Continue-on-match macro on every subsequent character of the next protein sequence. At the end of streaming of all the protein sequences, STE 12 of the Repeat macro is active only if the motif has occurred in all the protein sequences and an output report is generated on processing the 0_{16} character, otherwise not. Notice that the connection from the output port $o2$ of the

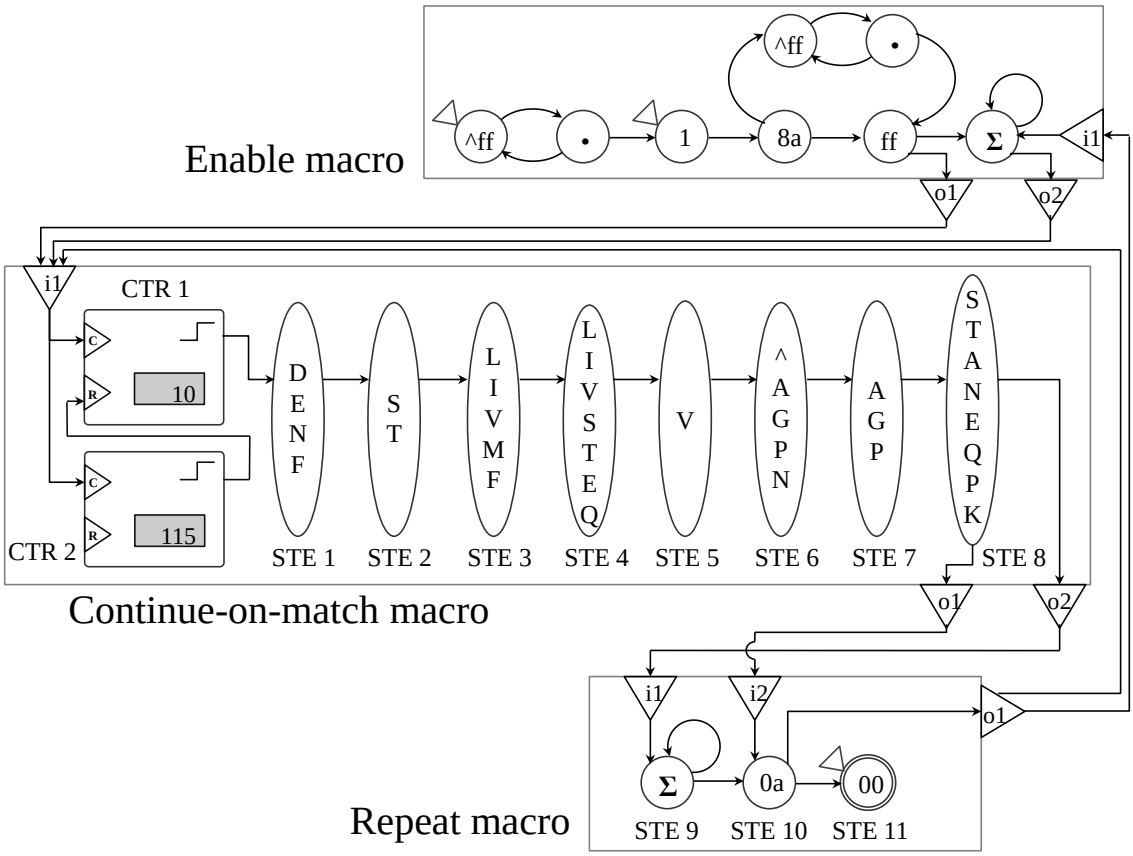


Figure 25: The *Global-match* automaton.

Continue-on-match macro to the input terminal $i1$ of the Repeat macro allows the motif to be matched anywhere within the protein sequence. If the motif needs to be anchored to the end of the protein sequence then only output port $o1$ of the Continue-on-match macro is connected to the input terminal $i1$ of the Repeat macro.

4.3 Performance Evaluation

For both these applications, the PCRE regex and ANML-NFA can be defined and compiled before-hand. Therefore, although the compile-times of both these applications have been reported here, they do not figure in the run-time calculations. All the CPU-based operations (compilation and execution of host-application) are executed on a quad-core *Intel(R) Core(TM) i5-3570* CPU, running at 3.4 GHz with 8 GB of main memory.

The correctness of the ANML-NFA and their ability to find all the matching occurrences have been verified using a simulator which comes with the SDK. However, the space and time constraints restrict the simulations to be limited to a small number of patterns at a given time. Therefore, the run-times for real use-cases are estimated by using accurately known input streaming time, output-handling handling time and the possible process-stalling that it may generate. These times can be calculated precisely because the length of the input and the occurrences of patterns in the input data is known.

The computation on the host application comprises of fetching the input data, organizing the same into a data-flow, stream the data-flow to the AP board, and take actions based on the occurrence of patterns as reported by the AP. This is not expected to be the bottleneck for the applications discussed in this chapter and can be easily hidden by an asynchronous multi-threaded pipeline.

4.3.1 Estimation model

The number of symbol-cycles (t) required to read out an output-vector from the output-buffer can be calculated as $t = 40 \times r + 32$, where r is the number of *output regions* in which at least one reporting STE is matched in that output event. The AP chip contains a total of 6 output regions. Therefore $t = 72$ and $t = 272$ correspond to the best case ($r = 1$) and worst case ($r = 6$) scenarios. Unfortunately, the placement of the automata within different output-regions of a AP chip is currently outside the control of the programmer. Hence, for our estimation, we have always assumed the worst case, i.e. $t = 40 \times min(p, 6) + 32$ where p is the number of reporting STEs matched in the output event.

4.3.2 Snort

Owing to delays in receiving necessary software, the implementation of Fast-SNAP is an ongoing process. Here, we briefly outline the preliminary results.

4.3.2.1 Configuration overhead

The current ruleset from the Snort database can be downloaded from the Snort website. The conversion of the active rules to the equivalent PCRE regex or ANML-NFA takes about 3 minutes. Compiling these regexes and ANML-NFA takes about 2 hours.

4.3.2.2 Run-time estimation

We assign different logical cores for rules related to the header and payload sections of a data packet. The host application running on the CPU handles the packet decomposition and schedules the streaming of the payload section to a logical core handling the payload rules, and the header section to a logical core handling the header rules. Since the processing of any section is only dependent on the length of the section, and length of a section is typically fixed in a network packet, this scheduling is fairly simple, and can be pipelined with the streaming of the data to the AP, thus obviating the overhead of this operation.

All the regexes and ANML-NFA for checking signatures of intrusion related to the body section of the network packet can be fit inside a logical core of size 4, whereas all the rules for the header sections require a logical core of size 2. Therefore, with an AP board containing 6 ranks of 8 chips each, we can support a data rate 8 Gbps. This is done by employing the first 4 ranks to contain two logical cores each for checking the body section of the 8 network packets in parallel. The remaining 2 ranks can be employed to contain the 8 logical cores required to support the scanning of 8 header sections in parallel.

However, better throughput can be derived by taking into account the asymmetry in the lengths of header and payload sections of the data packet for better load balancing. As the header section is typically much smaller than the payload section, the logical core handling the same would be idle most of the time. We attain better load balancing by making multiple copies of the payload section handling logical core for every header section handling logical core on the AP board. While, the header section from multiple packets are handled serially by the header section handling logical core, their payload sections are handled in parallel

by the multiple payload section handling logical cores. Using this scheme, a throughput of 11 Gbps can be supported by making 11 copies of the payload handling logical cores of 4 chips each, while employing the remaining 4 chips on the AP board to support 2 copies of the header section handling logical cores.

At the time of writing this document, a more fine-grained partitioning of the rules related to the payload section is being undertaken. This will allow the partitions to be programmed into smaller logical cores than of size 4, so that higher parallelism, and hence higher throughput can be derived using a single AP-board.

4.3.3 PROTOMATA

4.3.3.1 Configuration overhead

A file named *prosite.dat* can be downloaded from the PROSITE website which contains all the pattern-motifs in the database. The conversion of these motifs from the PROSITE pattern notation to the ANML-NFAs described in Section 4.2.2.1 takes about 1.6 seconds and compilation of these ANML-NFA takes about 20 minutes. All the Locate-occurrence automata for the motifs can be programmed using half the resources on a single AP chip. Similarly, all the Global-match automata can be fit into a single AP chip.

4.3.3.2 Run-time estimation

The run-times of the *ps_scan* application and PROTOMATA are compared in Table 1 using all the motifs from PROSITE and various proteomes (all proteins from a single organism) from the UniProtKB database [50]. The Swiss-Prot and the TrEMBL sections list the manually annotated sequences and the computationally analysed sequences from the database respectively. The number of protein sequences from the proteomes and their combined lengths are expressed in columns 3 and 4 respectively. The two search settings are whether 'greediness' (reporting the longest possible match starting at a location) is enabled or not; and if the search for 'frequently occurring motifs' is enabled or not. The latter has a huge impact on the number of motif occurrences found in the protein sequences, as captured in

Table 1: Comparison of run-times from *ps_scan* and PROTOMATA.

Database	Organism(s)	#Protein sequences	#Amino acids	Settings: Greediness	Settings: Frequently occurring motifs	#Motif occurrences	ps_scan: run-time (in ms)	PROTOMATA: Streaming time (in ms)	PROTOMATA: Output handling time (in ms)	PROTOMATA: Overall run-time (in ms)	Speed up (approx.)
UniProtKB/ Swiss-Prot	E.coli	4305	1,360,331	enabled	enabled	65,826	213,064	11	52	102	2,088
					disabled	1,644	209,168	11	2	61	3,428
				disabled	enabled	65,826	216,152	11	52	102	2,119
					disabled	1,644	211,922	11	2	61	3,474
	human	20,183	11,336,473	enabled	enabled	630,396	1,238,029	89	493	543	2,279
					disabled	24,332	1,217,581	89	20	139	8,759
				disabled	enabled	630,476	1,220,367	89	493	543	2,279
					disabled	24,385	1,196,148	89	20	139	8,605
	all	479,406	174,899,570	enabled	enabled	9,761,277	23,413,614	1,367	7,626	7,676	3,050
					disabled	755,365	23,632,751	1,367	591	1,417	16,678
				disabled	enabled	9,767,490	24,039,496	1,367	7,631	7,681	3,129
					disabled	761,442	23,623,676	1,367	595	1,417	16,671
UniProtKB/ TrEMBL	E.coli	4,333	1,372,277	enabled	enabled	66,552	216,273	11	52	102	2,120
					disabled	1,655	211,773	11	2	61	3,471
				disabled	enabled	66,552	219,984	11	52	102	2,156
					disabled	1,655	213,158	11	2	61	3,494
	human	67,084	22,252,781	enabled	enabled	1,226,823	3,290,032	174	959	1009	3,260
					disabled	38,812	3,225,670	174	31	224	14,400
				disabled	enabled	1,226,906	3,265,080	174	959	1009	3,235
					disabled	38,865	3,205,392	174	31	224	14,309
	all	18,394,018	6,583,760,868	enabled	enabled	1,681,423,194	980,991,184	51,436	1,313,612	1,313,662	746
					disabled	1,327,068,910	962,727,380	51,436	1,036,773	1,036,823	928
				disabled	enabled	1,690,088,977	962,368,859	51,436	1,320,383	1,320,433	728
					disabled	1,322,376,307	942,714,285	51,436	1,033,107	1,033,157	912

the next column.

The run-times of *ps_scan* and PROTOMATA are listed next. For PROTOMATA, the 'streaming time' is the estimated time required to stream all the input protein sequences in the absence of any stalling by the output-handling bottleneck. The 'output handling time' lists the time required to read out all the output-vectors from the output-buffer. In our tests none of the motif occurrences happened on the same symbol-cycle. Therefore, the time taken to read out the vector should be around 72 symbol-cycles. However, for

our calculations, we have assumed this number to be 100 to take care of any unforeseen overheads in the handling of the pipeline. The 'overall run-time' has been arrived at by taking a maximum of the input and output-handling times and adding 50 milliseconds to account for the load-time.

The speed-up greatly depends on the size of the input data, and whether the search for the frequently occurring motifs is enabled or not. If the input data is not large enough, the run-time is dominated by the load-time as in the case of the *E.coli* proteome. If the search for frequently occurring motifs is enabled, the number of motif occurrences and hence the rate of output generation is greatly enhanced which slows down the overall processing rate. By using this setting, the speed-up varies between 2088 times and 3260 times. If the search for the frequently occurring motifs is disabled (general practice in the field), then expected speed-up goes up to between 3428 times and 16668 times.

The above results are calculated considering a single logical core executing on the AP board, i.e a maximum processing-rate of 1 Gbps. For the use-case demonstrated above, 48 logical cores can be executed in parallel on the AP board, because the associated automata can be fit inside a single AP chip. Therefore, if the input size is large enough it can be broken into 48 parts and streamed to 48 logical cores in parallel. However, if the input size is not large enough, the speed-up is not be linear, as the processing of the individual parts may be dominated by the load time. For example, consider the case of all proteomes in the $UniProtKB/Swiss-Prot$ database being scanned in a greedy fashion with the search for occurrences of frequently-occurring motifs disabled. The streaming time and the output handling time with 48 logical cores working in parallel would be $136748 \approx 29$ milliseconds and $59148 \approx 13$ milliseconds respectively. Thus the overall run-time is expected to be 79 milliseconds, i.e. dominated by the 50 milliseconds of load-time. Even then, there is an almost 18 times further speed-up using the 48 logical cores. For larger input sizes like all proteomes in the $UniProtKB/TrEMBL$ database, similar calculations show a more linear speed-up (almost 46 times), which is approximately a 0.7 million times speed-up over

ps_scan.

4.4 Summary

The design of AP inherently lends itself well to the acceleration of software which check for occurrences of thousands of pattern in an input data stream. Using this capability, two applications called Fast-SNAP and PROTOMATA were developed. Fast-SNAP conducts deep packet inspection by checking for signatures of network intrusion in network data-packets. On the other hand, PROTOMATA searches for patterns called *motifs* in protein sequences which are linked to biologically meaningful operations or characteristics. Both these applications show significant speed-up over contemporary methods. Besides, the techniques and methods described in this chapter are both generic and modular and may be used by other applications needing similar processing capabilities.

Chapter V

FINDING DE-NOVO PATTERNS IN BIOLOGICAL SEQUENCES

In the last chapter, we looked at the use of AP for its design-goal, i.e. to accelerate applications which check for occurrences of thousands of string-patterns in parallel. In this chapter, we look at its use in a more non-intuitive manner. In fact, this application was the first published paper [62] on accelerating a complex application using the AP. It laid the foundations for understanding how to solve graph problems using the Automata Processor.

The application solves an important problem of finding similarities between biological sequences defined in more detail in Section 5.1. We designed an exact algorithm for solving this using NFA programmed into the AP. Our algorithm, termed MOTOMATA for *MOTif autOMATA*, not only has lower execution times but can also handle more challenging instances of the problem which are hitherto unsolved. Macros used in the algorithm is described in Section 5.2. An overview of the algorithm is provided in Section 5.3. The algorithm is divided into two distinct stages. The first stage is the most time-consuming and is accelerated using the AP. This is detailed in Section 5.4. Section 5.5 describes a CPU-implementation of second stage of the algorithm. The expected times of the first stage and the actual run-times of the second stage is presented in Section 5.6. These run-times for various sizes of the problem are a few order of magnitudes faster than those reported by prominent software based methods.

5.1 Background

Large repositories of genetic data have been produced through numerous sequencing projects, a trend that was significantly accelerated during the last decade. An important part of the analysis of this data consists of finding patterns in DNA, RNA or protein sequences. Discovery of these patterns called *motifs* helps in the identification of transcription factor

binding sites, transcriptional regulatory elements and their consequences on gene functions, variants causing human diseases, and therapeutic drug targets.

The problem of motif search has been classified into the following three types: *Planted (l,d) Motif search Problem* (PMP), *Simple Motif search Problem* (SMP), and *Edit-distance-based Motif search Problem* (EMP). Of these three types, PMP has been studied in the greatest detail and is described as follows. Given n sequences of length m each, find a motif M of length l which occurs in all the sequences with up to d mismatches. For example, given the input sequences $AGTCTCTCGAG$, $TTAGACGGTCA$, and $GATCAGTTCAC$, and $l = 4$, $d = 1$, the motif $CTCA$ occurs in all the three sequences. Note that the motif itself need not be present in its exact form in any of the sequences, which is the case in this example. The generic form of the PMP is defined as the *Quorum Planted Motif search Problem* (qPMP), where the motif M is present in at least q of the n sequences. PMP (and therefore qPMP) has been shown to be NP-complete [28].

Although the first motif was discovered in 1970 [45], this problem is far from being satisfactorily solved. *Approximate* algorithms [7, 48, 44, 59, 36] use statistical analysis to reach for solutions faster, but are not guaranteed to identify all motifs [57], sometimes even significant ones. On the other hand, *exact* algorithms [57, 60, 19, 24, 8, 23, 16, 58, 26, 14, 27, 52, 56] can identify all the motifs, but take large amount of time and memory to do so. Since all exact algorithms return the same answers, they are evaluated based on their execution time. For this evaluation, 20 sequences of length 600 are randomly generated by using characters from the alphabet. Similarly, a motif M of length l is generated. For each sequence, an occurrence of M with up to d random mismatches is *planted* at a random location in the sequence. Run-times are only reported for instances of (l,d) which are known to be *challenging*. For these instances, apart from the planted motif, additional motifs are expected to be found in the sequences. Examples of such instances are $(19,7)$, $(21,8)$, $(23,9)$, and $(25,10)$. At the time of this writing, the shortest execution time reported for the $(26,11)$ instance on a 48 core machine is 46.9 hours, and the $(28,12)$ instance remains

unsolved to the best of our knowledge.

5.2 Design of Building Blocks for Motif Search on Automata Processor

The key to solving problems on the AP is to cast the algorithms in terms of NFA. Once again, the design philosophy is to develop NFA with fixed topologies which can be pre-compiled as macros. This gives us the advantages of simplicity, modularity and speed. Below, we describe a building block automaton that is used throughout our application. It is designed to represent a sequence of length l and accept input strings of length l which differ from it in at most d positions.

5.2.1 Bounded Mismatch Identification Automaton

For sequence $S = s_1 s_2 s_3 ... s_i ... s_l$, our automaton design consists of $2d+1$ rows of STEs arranged in l columns. Column i corresponds to symbol s_i: the label of STEs in that column recognizes s_i if the row number is odd, and recognizes anything other than s_i (described as $\hat{s}_i$) if the row number is even. The STEs in odd rows are connected in a linear chain to reflect continued matching of symbols. Each mismatch causes skipping of two rows.

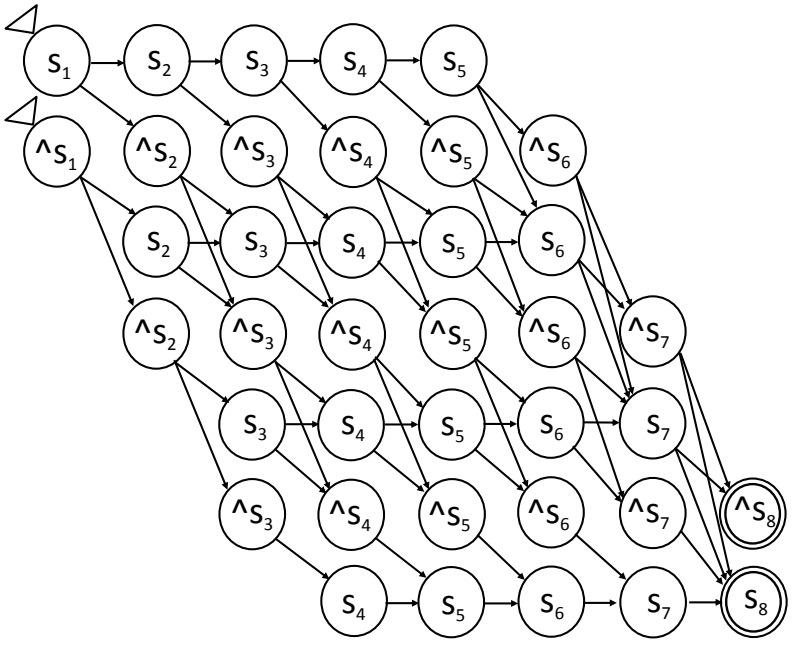


Figure 26: An automaton to accept strings with at most 3 mismatches from $s_1 s_2 \ldots s_8$.

The automaton has $l-d$ STEs in every odd numbered row and $l-d+1$ STEs in every even numbered row, thereby requiring a total of $(2d+1)l-2d^2$ STEs. An STE is present in column i and row j only if $j \leq min(2i, 2d+1)$ and $j \geq 2d-2(l-i)$. Intuitively, the last d symbols are recognized in successively lower rows with accept states restricted to last two rows. An example of the automaton for $l=8$ and $d=3$ is shown in Figure 26.

For succinctness, we refer to the bounded mismatch identification automaton as the (l,d) match automaton. If this automaton is used as a building block within a more complex automaton, the acceptance states of the (l,d) match automaton will cease to be so, but instead are connected to other STEs as needed. Similarly, we may wish to use (l,d) match automaton to detect patterns at the beginning of an input string or internal to an input string. For succinctness, we refer to these four cases using shorthand diagrams illustrated in Figure 27.

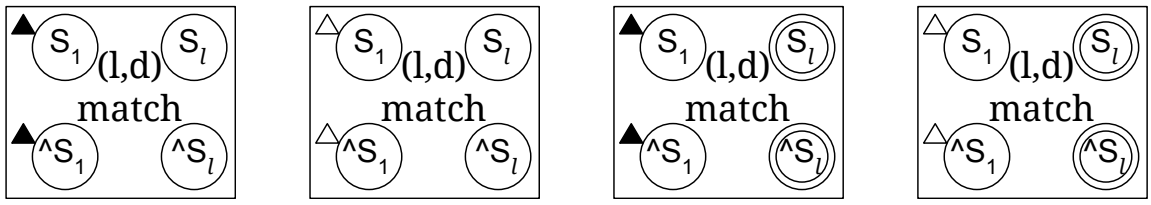


Figure 27: Shorthand representations of the (l,d) match automaton with (a) *all-input-start* STEs and no reporting STEs, (b) *start-of-data* STEs and no reporting STEs, (c) *all-input-start* STEs and reporting STEs, and (d) *start-of-data* STEs and reporting STEs.

5.3 Overview of Proposed Algorithm

Let $S^1, S^2, \ldots, S^n$ denote the n input sequences, where $S^i = s^i_1 s^i_2 \ldots s^i_m$. For convenience of presentation we assume all sequences are of length m, though this assumption is not made in the algorithm. Let S^i_j denote the substring of S^i of length l starting from position j $(1 \leq j \leq m-l+1)$, i.e. $S^i_j = s^i_j s^i_{j+1} \ldots s^i_{j+l-1}$. Define a multipartite sequence graph G on l-mers in the input sequences as follows: Node u^i_j in the graph represents S^i_j. An

edge $(u^i_j, u^{i'}_{j'})$ is drawn if and only if $i \neq i'$ and the Hamming distance between S^i_j and $S^{i'}_{j'}$ is $\leq 2d$. Graph G is therefore a multipartite graph with n partitions $P^1, P^2, \ldots, P^n$, where $P^i = \left\{ u^i_j \mid 1 \leq j \leq m-l+1 \right\}$ contains nodes corresponding to unique l-mers from S^i.

Observation 1. *Let M be an (l,d) motif occurring in the n sequences as $S^1_{j_1}, S^2_{j_2}, \ldots, S^n_{j_n}$, respectively. Then, the nodes $u^1_{j_1}, u^2_{j_2}, \ldots, u^n_{j_n}$ in G form a clique.*

Proof. Consider any two nodes $u^i_{j_i}$ and $u^k_{j_k}$. Since M is an (l,d) motif, the Hamming distance between M and $S^i_{j_i}$, and between M and $S^k_{j_k}$, is at most d. It follows that the Hamming distance between $S^i_{j_i}$ and $S^k_{j_k}$ is at most $2d$. Hence, an edge exists between nodes $u^i_{j_i}$ and $u^k_{j_k}$ in G. □

Note that the converse need not be true; i.e, there may be cliques of size n in G without a corresponding (l,d) motif in the sequences. This can happen, for example, if a potential candidate motif is at a distance of more than d from each of two l-mers $S^i_{j_i}$ and $S^k_{j_k}$, but these two l-mers agree at some positions where they differ from the candidate, leading to a shorter distance ($\leq 2d$) between them.

We exploit this observation in our MOTOMATA algorithm as follows. In the first stage, we identify all cliques of size n in the graph G. In the second stage, we search all possible candidate motifs for each clique to identify one or more motif sequences, if they exist.

Although identifying cliques greatly limits the search space, finding cliques is quite difficult. In the case of PMP, finding a clique of size n is tantamount to finding a maximal clique in the graph G, which is *NP*-hard. Knowing the size of the maximal clique and the multipartite nature of the graph does simplify the problem somewhat. However, for the *challenge problems*, the ratio of number of edges in the graph G to the number of edges retained in a clique is $20,000$:1 [57]. This leads to large run-times for microprocessor based algorithms.

In the second stage, every clique found in the first stage is targeted for possible motif discovery. Consider a clique and an l-mer sequence T corresponding to one of its nodes. If

a motif is present, it must be within a Hamming distance of d from T. Therefore, a search tree is created by arranging all sequences of length l within a Hamming distance of d from T. Each root-to-leaf path in the tree corresponds to a possible motif. Each such path is checked to see if it is at a Hamming distance of at most d from all the l-mers corresponding to the other nodes in the clique.

We developed the following algorithm for tree traversal. First, the tree is decomposed into layers, each layer containing subtrees rooted at a specific depth. The tree is built in such a way that the number of distinct subtrees in a layer are quite limited. Also notice that for all paths in subtrees from a layer, the substring to be matched from the sequence represented by a node in the clique remains the same. These observations are used to quickly find the number of mismatches between any path in the subtrees and the corresponding substrings of the l-mers from the cliques. Next the tree is traversed using a depth-first branch-and-bound technique, but at the granularity of the subtrees instead of individual nodes.

5.4 Finding Cliques in Multipartite Sequence Graph

We first present an algorithm to find cliques of size n, denoted *n-cliques* from here onwards, required for solving *PMP*. This can be easily adapted later to identify q-cliques ($q \leq n$) needed for the *qPMP* problem. A clique is represented by the set of nodes contained in it.

5.4.1 Overview of the Proposed Algorithm

We seek an exact solution to the clique finding problem. However, algorithmic techniques for pruning the search space without losing valid solutions, and economizing number of STEs required, are crucial to solving problems of practical interest. In addition, our algorithmic techniques should be amenable to AP implementation, which can only store automata and process streaming data.

We propose an iterative clique finding algorithm that incrementally builds cliques of increasingly larger size, starting from one. Initially, all nodes in G are considered 1-cliques. In iteration i, we extend i-cliques to form $(i+1)$-cliques.

5.4.2 Pruning the Clique Search Space

Storing all i-cliques requires significant storage and processing capabilities. Thus, we seek to eliminate as many i-cliques as possible from further consideration, before proceeding to the next iteration. We define node u to be a *neighbor* of clique C, if either 1) $u \in C$, or 2) $u \notin C$ and u is connected to every node in clique C. Recall that an i-clique consists of i nodes each of which is from a distinct partition of G.

Observation 2. *Let C be an i-clique. If $\exists$ partition P such that no $u \in P$ is a neighbor of C, then C is not a subclique of any n-clique.*

Proof. An n-clique must contain exactly one node from each of the n partitions. If clique C does not already contain a node from partition P, and if it cannot be extended to include any node from partition P, then it is not a subclique of any n-clique. □

This observation is used to weed out i-cliques that do not have a neighbor in each of the partitions. Note that the i partitions which contributed to the i-clique can also be included as part of the test, with no change in results. By definition, a node already in the clique is considered a neighbor to the clique. While these tests seem wasteful, they eliminate the need to keep track of partitions associated with each clique, and treat all partitions uniformly with respect to any clique of any size. This simplicity greatly aids in mapping the tests onto the AP.

5.4.3 Clique Management and Pruning the Node Search Space

An i-clique contains a node each from i distinct partitions. Multiple cliques, the nodes of which are derived from the same set of partitions, are grouped into *buckets*. Thus, there are $\binom{n}{i}$ buckets for storing i-cliques. Each bucket is given a unique n-bit *bucket id*, with each bit indicating the presence or absence of a particular partition in its cliques. We extend the notion of a *neighbor* to buckets. A node u is a neighbor of bucket B if u is a neighbor of at least one clique $C \in B$.

As with clique pruning, it is also possible to identify nodes that can never be part of an n-clique, and thus remove them from future consideration, including, in subsequent building of cliques.

Observation 3. *Let C be an n-clique in G. In iteration i, each $u \in C$ must be a neighbor of all buckets containing i-cliques.*

Proof. Consider a bucket B containing i-cliques. Without loss of generality, let $P^1, P^2, P^3, \ldots, P^i$ denote the partitions from which nodes in a clique in B are drawn. Then, $\exists$ a sub-clique D of C, containing only those nodes of C which are from any of $P^1, P^2, P^3, \ldots, P^i$. It is clear that D is a clique in B, and each $u \in C$ is a neighbor of D. □

This observation can be used in reverse to prune the node search space. Buckets containing i-cliques are in place at the beginning of iteration i. In this iteration, if we find a node that is not a neighbor of one of the buckets containing i-cliques, then that node cannot be a part of any n-clique. Not only can that node be removed, all cliques that currently contain this node can also be removed. Organizing cliques into buckets is needed to identify such nodes, and their removal in turn paves the way for pruning cliques that contain any such nodes. Pseudocode describing the clique identification algorithm is described in Algorithm 1.

5.4.4 Clique Finding using Automata Processor

The most time consuming part in the algorithm is to test each surviving node in the graph in each iteration to see if it is a neighbor of all buckets. A node u that fails this test is removed along with all cliques containing u. A node u that passes the test survives for the next round. In addition, the testing process finds all cliques that u is a neighbor to. This information is then used to extend those cliques to include u, so that cliques for the next iteration are formed.

Cliques are maintained on the CPU. Node testing is carried out on the AP, by implementing nodes as automata and buckets of cliques streamed as input. To do so, recall that

Algorithm 1: Algorithm for finding n-cliques in graph G.

Input : G, l, d

Output: All n-cliques in G

nodeBasedNCliquesSearch (G, l, d)

for $i = 1$ *to* $n-1$ **do**

  for *every node* $u \in P^j \; \forall j : 1 \leq j \leq n$ **do**

    if *u is a neighbor of all buckets in this iteration* **then**

      Create cliques of size $i+1$ by adding u to all cliques that u is a neighbor of.

      Calculate bucket id for new cliques by setting the j^{th} least significant bit to 1.

    end

    else

      Remove u and all cliques containing u.

    end

  end

  if *any bucket becomes empty* **then**

    Stop and report no cliques.

  end

end

return all cliques from last bucket

a node represents an l-mer from one of the input sequences. To test if a node u is neighbor to a clique C, we need to check if u is connected to each $v \in C$. By definition, u and v are connected if the Hamming distance between their corresponding l-mers is $\leq 2d$. Thus, we can use a $(l, 2d)$ match automaton for the l-mer represented by u and stream v. To perform such tests on all nodes in C, we simply stream the concatenation of l-mers of each of its constituent nodes. Thus, an i-clique translates to a stream of length il, and multiple successive tests can be conducted by linking the output of the recognizing automaton to its input. The topology of the automata for all nodes is identical, but labels inside the STEs vary. We purposefully achieve this for efficient execution on the AP.

5.4.5 Automata Processor Implementation Details

To delimit boundaries between nodes, cliques, and buckets when streaming input, we use special characters $\{X, Y, Z, B\}$ not part of the sequence alphabet. These delimiters help to purge the $(l, 2d)$ match automaton and remember various state information. Let X^{2d} denote a sequence of $2d$ consecutive X characters. We use Y, $X^{2d}Z$, $X^{2d}B$ and E to mark node, clique, bucket, and data flow boundaries, respectively.

Logically, one continuous data flow is generated by the CPU for each iteration, and streamed twice: Once to identify which nodes survive, and the second time to create cliques for the next iteration using these nodes. In iteration i, the data flow consists of a sequence of all buckets containing i-cliques with appropriate node/clique/bucket/data flow delimiters. The order of buckets, of cliques within a bucket, and of nodes within a clique, is irrelevant. Despite this logical view, actual data need not be copied in each iteration to make up the data flow. All the l-mers and the delimiters are stored once in the physical memory. Logical chunks permitted in the definition of the data flow to the AP are flexibly used to point to the actual locations of the l-mers and delimiters to produce the desired logical data flow.

To identify if node u is a neighbor of all buckets in the data flow, we designed the automaton shown in Figure 28. The $(l, 2d)$-match building block accepts l-mers within a

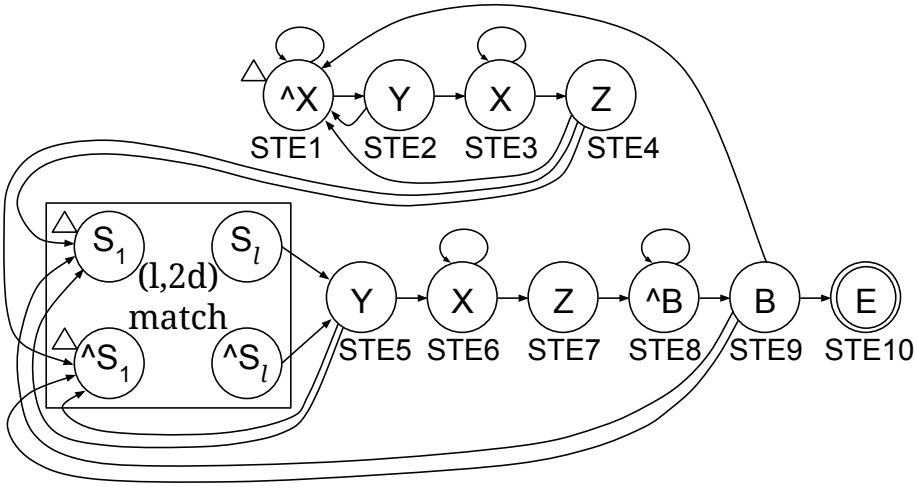


Figure 28: Automaton to check if node is a neighbor of every bucket.

Hamming distance of $2d$ from the l-mer represented by u. The STEs 5, 7, and 9 are reached when a neighboring node, clique, and bucket, respectively are found in the data flow. STEs 6 and 8 are parking states used to wait for the transition between two consecutive cliques, and from an accepted clique to the end of its current bucket, respectively. The X^{2d} part of delimiter sequences is needed to flush the $(l, 2d)$ match automaton in preparation for next l-mer match. Note that u survives for next iteration if there is at least one neighboring clique in each bucket. Thus, the clique neighborhood test should be activated for each clique, even if the prior clique neighborhood test is failed. To achieve this, STEs 1 through 4 are used to bypass cliques not neighboring u. STE 4 is reached at the end of every clique, and triggers clique neighborhood test machinery for the next clique.

If STE 9 is active at the penultimate character of the data-flow, then node u is a neighbor of all buckets in the data-flow. The delimiter at the end of the data-flow generates an output recognizing u to be retained for the next round.

Nodes that survive the neighborhood test as indicated by their automata reaching the respective acceptance states, are reported to the CPU. We then program the AP using the automaton shown in Figure 29 for every surviving node. The purpose of this automaton is to identify each neighboring clique of a node so that the clique can be extended to include

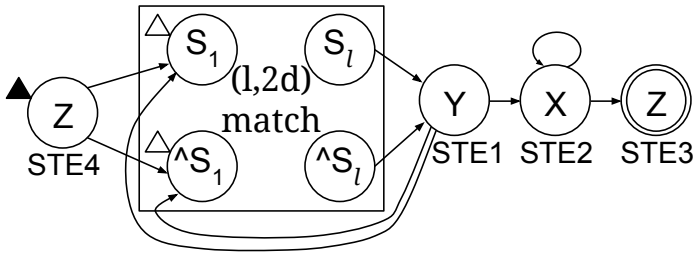


Figure 29: Automata to identify neighboring cliques of a node.

the node by the CPU. Not surprisingly, this automaton is very similar to that in Figure 28. STE 3 triggers output with identification of each neighboring clique. The offset into the data flow where the clique is found is used by the CPU to identify the clique.

5.4.6 qPMP solution

The qPMP problem takes a factor of $\binom{n}{q}$ larger run time than the PMP problem on a conventional CPU. The qPMP solution on the AP is very similar to the PMP solution, requiring only minor changes. The algorithm is run for only $q-1$ iterations instead of $n-1$ iterations. On line 4 of the *nodeBasedNCliquesSearch* function, instead of requiring u to be a neighbor of *all* buckets of size i, u is retained if it is a neighbor of *any* bucket of size i. This is equivalent to checking if u is a neighbor of any clique in the data flow, the corresponding automaton for which has already been described in Figure 29.

5.5 Searching for Motifs Consistent With Cliques

Let C be an n-clique of nodes $\{u^1_{j_1}, u^2_{j_2}, \ldots, u^n_{j_n}\}$ identified in stage one of the algorithm. Let $T^i = S^i_{j_i}$ be the l-mer corresponding to node $u^i_{j_i}$, i.e., the l-mer starting from position j_i in input sequence S^i. By writing $T^i = t^i_1 t^i_2 \ldots t^i_l$, we can simplify the notation by dropping the starting position of T^i in S^i.

If a motif is present, then it must be at a Hamming distance of $\leq d$ from each of the sequences in $\{T^1, T^2, T^3 \ldots T^n\}$. Therefore, if we enumerate all sequences of length l at a Hamming distance of $\leq d$ from T^1, the search for a motif can be confined to this set of sequences. Each such sequence can be tested by checking if it is within a Hamming

distance of $\leq d$ from all the other sequences in $\{T^2, T^3, \ldots, T^n\}$, and if so, declared a motif.

5.5.1 Arranging Candidate Motifs as Search Trees

To enumerate potential motifs, a search tree is constructed based on T^1. The tree has $l+1$ levels, where the root node is at level 0. Each node in the tree is assigned a single character label. We define the *string label* of a node v as the concatenation of labels in the nodes on the path from root to v. Let $w_v \leq d$ denote the Hamming distance between the string label of v at level i and the sequence $t_1^1 t_2^1 \ldots t_i^1$. If $w_v < d$, then the leftmost child of v has the label t_{i+1}^1 and a weight of w_v. All other children of v represent a possible substitution for t_{i+1}^1 and are arranged in lexicographic order from left to right, and assigned a weight of $w_v + 1$. If $w_v = d$, then v has only one child with the label of t_{i+1}^1 and a weight of w_v. An example search tree for $S = actgtc$ allowing a maximum of 1 mismatch is shown in Figure 30. The label of each node is shown inside the circle, whereas the Hamming distance of its string label is shown adjacent to the circle. If a motif is present, then it must be the string label of a leaf in this tree.

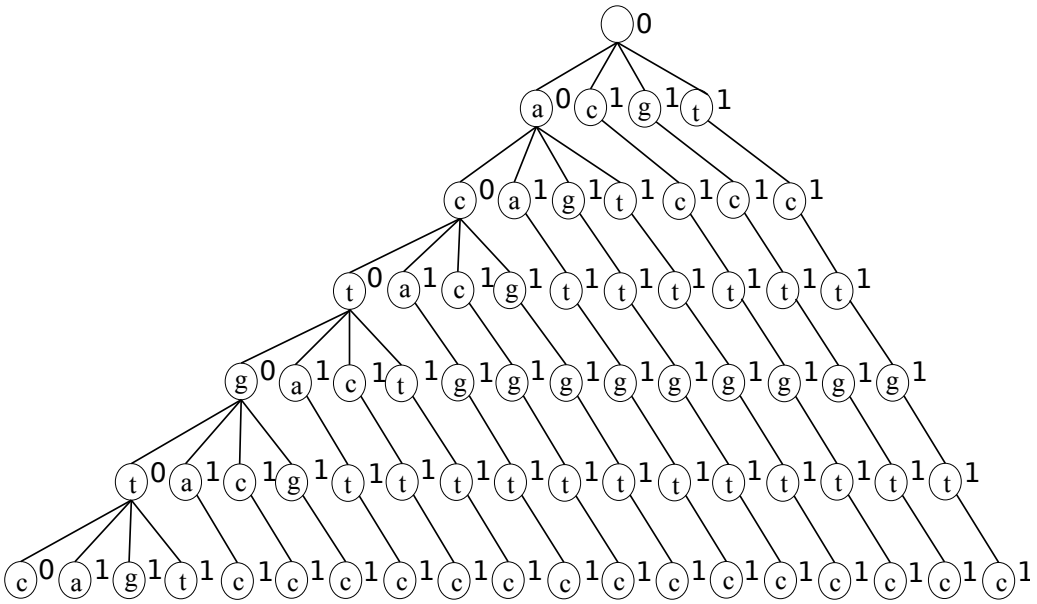


Figure 30: Search tree for $S = actgtc$ with up to one mismatch.

Let p_i denote the number of nodes at level i of the search tree for the genomic alphabet $= \{a, c, g, t\}$.

$$p_i = \begin{cases} \sum_{j=0}^{i} 3^j . \binom{i}{j} & \text{if } 1 \leq i \leq d; \\ \sum_{j=0}^{d} 3^j . \binom{i}{j} & \text{if } d < i \leq l; \end{cases}$$

Each term in the summation is the number of nodes at level i with a weight of j. For large values of l and d, the search tree can be extremely large. However, two key observations greatly help in scanning this tree quickly.

Observation 4. *For any two nodes u and v appearing at the same level in a search tree with $w_u = w_v$, the subtrees rooted at u and v are identical if we disregard the labels of u and v. Since w can only take values between* 0 *and d, the number of distinct subtrees at any level of the tree is bounded by $d+1$. This is illustrated in Figure 31.*

5.5.2 Handling Large Search Trees

For large values of l and d, the search tree is very large, and a naive branch-and-bound algorithm which continues scanning the subtree rooted at any node v at level i only if its string label has $\leq d$ mismatches with i-length prefix of all the sequences in $\{T^2, T^3, \ldots, T^n\}$, is too slow. Instead, our algorithm provides a much faster solution by dividing the search tree into numerous subtrees, and combining the computations for a large number of identical subtrees.

We partition the levels of the search tree into λ *layers* with each layer containing a consecutive set of levels, with the exception that a level at the boundary of two consecutive layers is included in both. Let $R = \{r_1, r_2, \ldots, r_\lambda\}$ denote the number of levels in the λ layers. If $r = \lceil l/\lambda \rceil$, then $r_1 = r_2 = \cdots = r_{\lambda-1} = r$ and $r_\lambda = l - (\lambda - 1) \times r + 1$. Each layer is partitioned into subtrees of nodes from the top level in the layer. Due to Observation 4, the number of distinct subtrees in any layer is $min((i-1)r, d) + 1$. This is illustrated in Figure 32 where the search tree is partitioned into three layers containing three levels each. Notice that the root nodes of the subtrees are not assigned any character label.

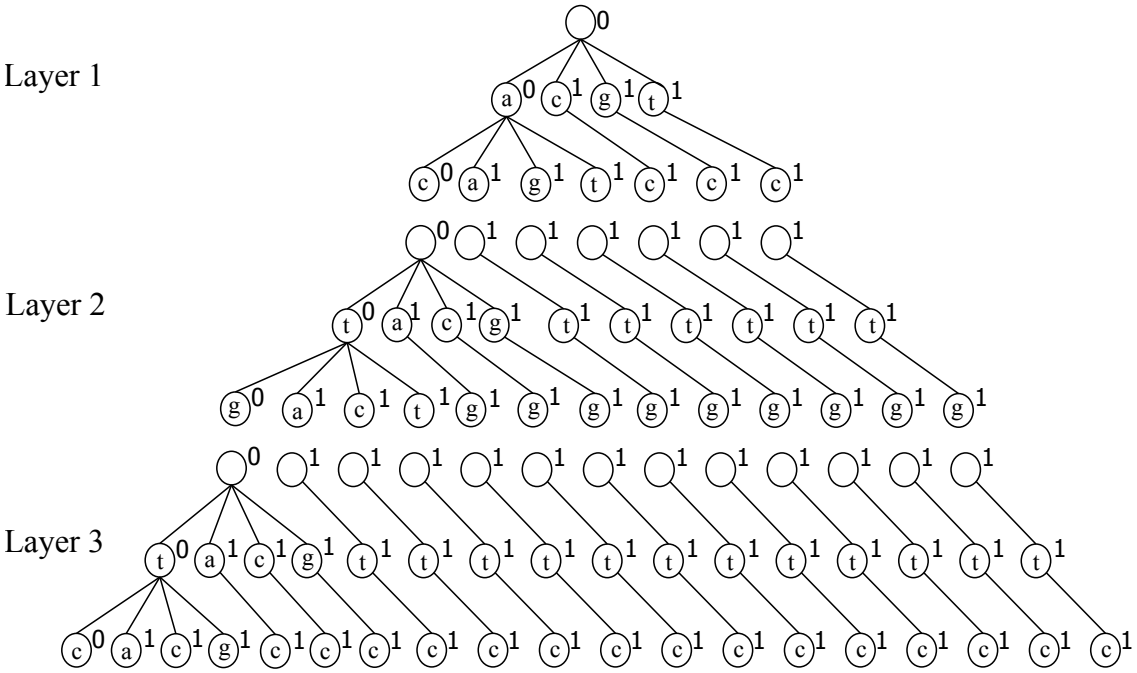


Figure 31: Search tree shown in Figure 30 partitioned into multiple layers.

For each layer, we define the *substring label* of a leaf node v as the concatenation of labels in the nodes on the path from the root to v in the subtree to which v belongs. For example, the substring label of the leftmost leaf node in Layer 1, 2 and 3 of Figure 32 are ac, tg and tc respectively. Note that each candidate motif sequence is the concatenation of λ substring labels, one each from layers 1 to λ. These constituent substring labels are henceforth denoted as $\mu_1, \mu_2, \ldots, \mu_\lambda$. Similarly, every sequence T^i is partitioned into λ subsequences $\tau_1^i, \tau_2^i, \ldots, \tau_\lambda^i$, where the length of μ_j and τ_j^i is equal, $\forall_j 1 \leq j \leq n$.

Observation 5. *If the Hamming distance between a candidate motif and T^i is at most d, then $\sum_{k=1}^{j} H(\mu_k, \tau_k^i) \leq d$ ($\forall_j 1 \leq j \leq \lambda$), where $H(\mu_k, \tau_k^i)$ represents the Hamming distance between μ_k and τ_k^i.*

5.5.3 Overview of the Proposed Algorithm

The identification of actual motifs from the list of candidate motifs represented by string labels of leaf nodes in the search tree is divided into two phases. In phase 1, we prune the search space by identifying candidate motifs which have at most d mismatches from at least

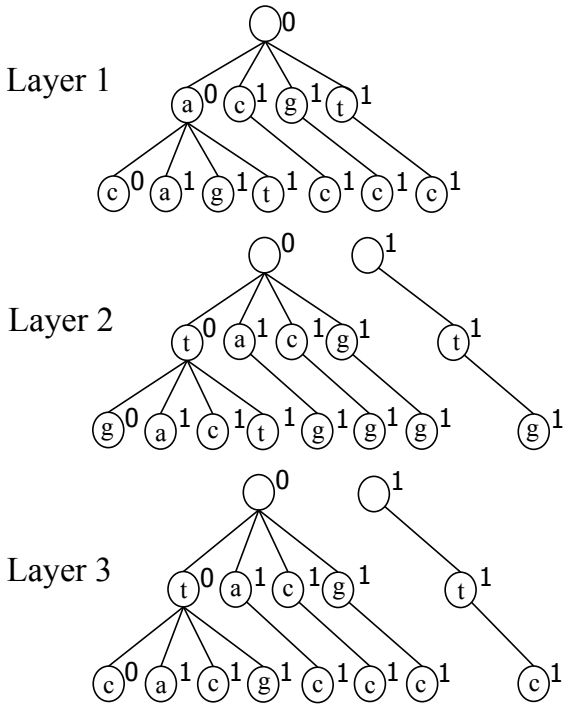


Figure 32: Unique subtrees in each layer of the search tree.

one sequence T^i, where $2 \leq i \leq n$. We call them *eligible* candidate motifs with respect to T^i. This phase uses a branch-and-bound technique based on Observation 5: Evaluation of a candidate motif at layer $j+1$ continues only if $\sum_{k=1}^{j} H(\mu_k, \tau_k^i) \leq d$, and only those candidate motifs for which the above condition is met at layer λ are termed as eligible candidate motifs. For each layer j, further acceleration is obtained by combining the operations for substring labels from the layer which have the same number of mismatches with τ_j^i. Phase 1 has been described in detail in Section 5.5.4.

In Phase 2, each selected candidate from phase 1 is checked to see if it is eligible with respect to all sequences T^i, $\forall_i 2 \leq i \leq n$, and if so, declared as an actual motif. Since the subtree rooted at any two nodes u and v from the same level in the search tree is identical when $w_u = w_v$, we use the information gleaned from traversing the subtree rooted at node u to decide if we need to traverse the subtree at node v. This branch and bound technique allows us to significantly accelerate Phase 2 which has been described in detail in Section 5.5.5.

Sections 5.5.4 and 5.5.5 below fully describe the details of how Phase 1 and Phase 2 are designed and implemented. Note that the algorithms used for these phases are executed on CPU alone. The reader interested only in a high-level overview of the algorithms can safely skip to Section 5.6.

5.5.4 Filtering candidate motifs With Less Than d Mismatches With At Least One Sequence

Instead of storing the large number of candidate motifs directly, we express them as concatenation of substring labels from different layers. By organizing these substring labels into elements of a matrix, and then defining rules on how to concatenate a substring label from one element with a substring label from another, we can represent all the candidate motifs. For every value of i between 2 and n, our method organizes substring labels from each layer j which have a common Hamming distance from τ_j^1 and a common Hamming distance from τ_j^i. This allows the algorithm to consume significantly lesser run-time and memory by combining the operations for all substrings grouped into the same element.

5.5.4.1 Data structures required

- **Local Mismatch Matrix (χ)**: is a 4-dimensional matrix with axes i, j, δ and δ^i, where $2 \leq i \leq n$, $1 \leq j \leq \lambda$, and $0 \leq \delta, \delta^i \leq r$. Element $\chi[i][j][\delta][\delta^i]$ from layer j is denoted by $\chi_\delta^{\delta^i}[j]$, and contains the substring labels of candidate motifs from layer j, which have δ mismatches with τ_j^1 and δ^i mismatches with τ_j^i. $\chi_\delta^{\delta^i}[j]$ is assigned an integer $id = r \times \delta + \delta^i$.

- **Global Mismatch Matrix (ψ)**: is a 4-dimensional matrix with axes i, j, ρ and ρ^i, where $2 \leq i \leq n$, $1 \leq j \leq \lambda$, and $0 \leq \rho, \rho^i \leq d$. Element $\psi[i][j][\rho][\rho^i]$ from layer j is denoted by $\psi_\rho^{\rho^i}[j]$, and represents the $(j \times r)$-length prefix of all candidate motifs which have ρ mismatches with the $(j \times r)$-length prefix of T^1 and ρ^i mismatches with the $(j \times r)$-length prefix of T^i. Instead of storing all these prefixes directly, it only stores the ids of the elements from layer j of the Local Mismatch matrix. If

the id of $\chi_{\delta}^{\delta^i}[j]$ is listed in $\psi_{\rho}^{\rho^i}[j]$, then all substring labels in $\chi_{\delta}^{\delta^i}[j]$ occur as the j^{th} substring of candidate motifs whose $(j \times r)$-length prefix has ρ mismatches with the $(j \times r)$-length prefix of T^1 and ρ^i mismatches with the $(j \times r)$-length prefix of T^i. The entries for layer $(j+1)$ of the Global Mismatch Matrix can be generated using the entries from layer j of the Global Mismatch Matrix and layer $(j+1)$ of the Local Mismatch Matrix. This has been shown in Figure 33.

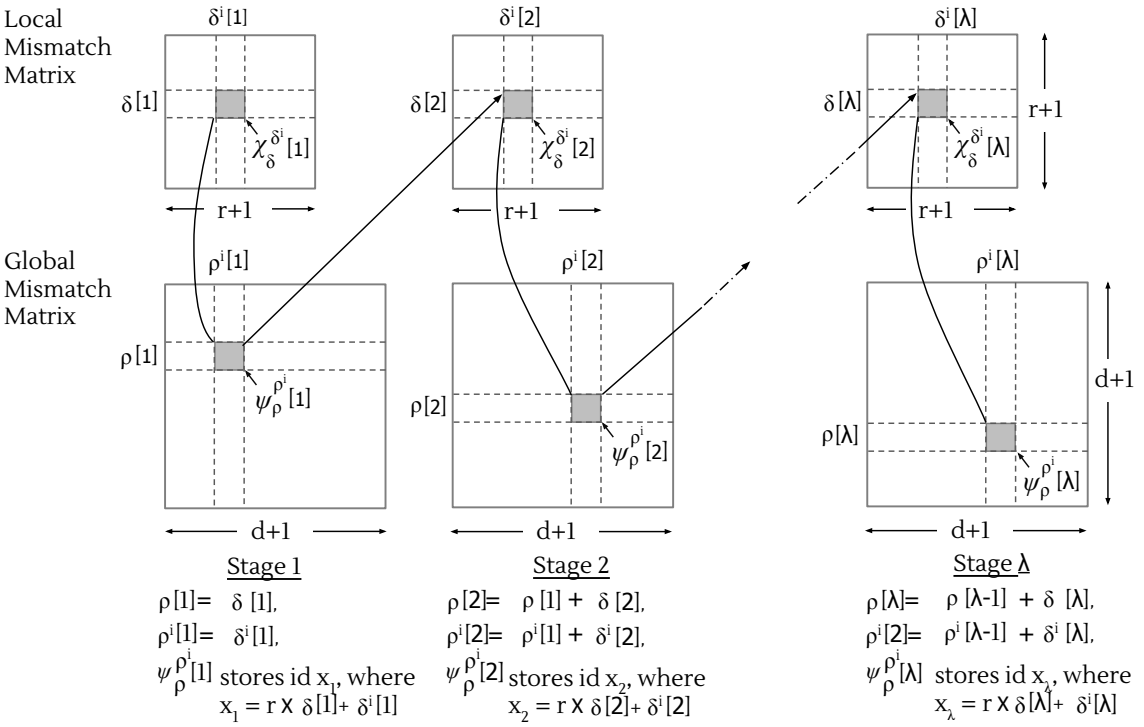


Figure 33: Representation of candidate motif sequences with respect to T^i using the *Local* and *Global Mismatch Matrices*. Elements of the *Local Mismatch Matrix* are used to group all substring labels from layer j having a common Hamming distance from τ_j^1 and a common Hamming distance from τ_j^i. Using this Local Mismatch Matrix, the elements of *Global Mismatch Matrix* represent $(j \times r)$-length prefix of all candidate motifs which have a common Hamming distance from the $(j \times r)$-length prefix of T^1 and a common Hamming distance from the $(j \times r)$-length prefix of T^i.

- **Pruned Substring Matrix (φ)**: is a 4-dimensional matrix with axes i, j, ρ, and ρ^i, where $2 \leq i \leq n$, $1 \leq j \leq \lambda$, and $0 \leq \rho, \rho^i \leq d$. Element $\varphi[i][j][\rho][\rho^i]$ from layer j is denoted by $\varphi_{\rho}^{\rho^i}[j]$ and represents the $(j \times r)$-length prefix of only eligible candidate motifs with respect to sequence T^i, which have ρ mismatches with the $(j \times r)$-length prefix of T^1 and ρ^i mismatches with the $(j \times r)$-length prefix of T^i. $\varphi_{\rho}^{\rho^i}[j]$ lists the ids of elements from layer $j+1$ of the Local Mismatch matrix whose entries constitute the $(j+1)^{th}$ substring of eligible candidate motifs with respect to T^i.

5.5.4.2 Accelerated Algorithm using Mismatch Matrices

Filling up the Local Mismatch Matrix All unique substring labels from each layer are enumerated and organized based on their Hamming distance from τ^1 and τ^i, $\forall_i 2 \leq i \leq n$ as shown in Algorithm 2.

Filling up the Global Mismatch Matrix Element $\psi_{\rho}^{\rho^i}[j]$ denotes the sequence formed by concatenating substring labels $\mu_1, \mu_2, \ldots, \mu_j$ from layers 1 to j such that $\sum_{k=1}^{j} H(\mu_k, \tau_k^1) = \rho$ and $\sum_{k=1}^{j} H(\mu_k, \tau_k^i) = \rho^i$. Notice that if $\delta[k] = H(\mu_k, \tau_k^1)$ and $\delta^i[k] = H(\mu_k, \tau_k^i)$, then μ_k is enlisted in $\chi_{\delta}^{\delta^i}[k]$, $\forall_k 1 \leq k \leq j$. Additionally, if μ_k is replaced by any other substring from $\chi_{\delta}^{\delta^i}[k]$, then the resultant string also has ρ mismatches with the $(j \times r)$-length prefix of T^1 and ρ^i mismatches with $(j \times r)$-length prefix of T^i. Therefore, we do not need to list the substrings individually. Instead, we can work with the ids of the elements from the Local Mismatch Matrix. Besides, $\psi_{\rho-\delta[j]}^{\rho^i-\delta^i[j]}[j-1]$ already represents the $((j-1) \times r)$-length prefix of candidate motifs which have $\rho - \delta[j]$ mismatches with the $((j-1) \times r)$-length prefix of T^1 and $\rho^i - \delta^i[j]$ mismatches with the $((j-1) \times r)$-length prefix of T^i. Thus, for $\psi_{\rho}^{\rho^i}[j]$ we only need to list ids of $\chi_{\delta}^{\delta^i}[j]$ which are non-empty, and $0 \leq \delta[j] \leq min(r_j, \rho)$ and $0 \leq \delta^i[j] \leq min(r_j, \rho^i)$. This provides a very compact representation of the search tree.

Figure 33 illustrates the use of Algorithm 3 to fill up the Global Mismatch Matrix. For any eligible candidate motif with respect to T^i, if its substrings from layers 1 to λ have $\delta[1], \delta[2], \ldots, \delta[j], \ldots, \delta[\lambda]$ mismatches with $\tau_1^1, \tau_2^1, \ldots, \tau_j^1, \ldots, \tau_\lambda^1$, and $\delta^i[1], \delta^i[2], \ldots$

Algorithm 2: Organizing Substring Labels from Each Layer into the Local Mismatch Matrix.

Input : $C = \{T^1, T^2, T^3 ... T^n\}$,

$R = \{r_1, r_2, \ldots, r_\lambda\}, \lambda, r, n$

Output: χ

ComputeLocalMismatchMatrix (C, R, λ, r, n)

for $j = 1$ *to* λ **do**

  for $i = 2$ *to* n **do**

    for *all substring labels from layer* j **do**

      Calculate $\delta =$ Hamming dist. from τ_j^1

      Calculate $\delta^i =$ Hamming dist. from τ_j^i

      Append substring label to the list stored at $\chi_\delta^{\delta^i}[j]$

    end

  end

end

$, \delta^i[j], \ldots, \delta^i[\lambda]$ mismatches with $\tau_1^i, \tau_2^i, \ldots, \tau_j^i, \ldots, \tau_\lambda^i$; then the ids of $\chi_{\delta[1]}^{\delta^i[1]}[1], \chi_{\delta[2]}^{\delta^i[2]}[2], \ldots,$ $\ldots \chi_{\delta[j]}^{\delta^i[j]}[j], \ldots, \chi_{\delta[\lambda]}^{\delta^i[\lambda]}[\lambda]$ are listed in the lists stored at $\psi_{\rho[1]}^{\rho^i[1]}[1], \psi_{\rho[2]}^{\rho^i[2]}[2], \ldots, \psi_{\rho[j]}^{\rho^i[j]}[j], \ldots$ $\ldots \psi_{\rho[\lambda]}^{\rho^i[\lambda]}[\lambda]$ respectively, where $\rho[j] = \sum_{k=1}^{j} \delta[j] \leq d$ and $\rho^i[j] = \sum_{k=1}^{j} \delta^i[j] \leq d$.

Filling up the Global Mismatch matrix with respect to each sequence T^i $(\forall_i 2 \leq i \leq n)$ starts from the first layer. If $\chi_{\delta[1]}^{\delta^i[1]}[1]$ is non-empty, then the list at $\psi_{\delta[1]}^{\delta^i[1]}[1]$ stores the id of $\chi_{\delta[1]}^{\delta^i[1]}[1]$ only, signifying that all r-length prefixes of candidate motifs with $\delta[1]$ and $\delta^i[1]$ mismatches with τ_1^1 and τ_1^i respectively can only be substring labels listed in $\chi_{\delta[1]}^{\delta^i[1]}[1]$. Now for layer 2, the id of $\chi_{\delta[2]}^{\delta^i[2]}[2]$ is added to the list $\psi_{\rho[2]}^{\rho^i[2]}[2]$, only if $\psi_{\rho[1]}^{\rho^i[1]}[1]$ and $\chi_{\delta[2]}^{\delta^i[2]}[2]$ are both non-empty; and $\rho[2] = \rho[1] + \delta[2] \leq d$ and $\rho^i[2] = \rho^i[1] + \delta^i[2] \leq d$. The process is continued until no sequences can be extended using substring labels from the next layer (no true motifs exist), or lists for all elements in layer λ have been generated.

Algorithm 3: Generating Compact Representation of Candidate Motifs using the Global Mismatch Matrix.

Input : $\chi, R = \{r_1, r_2, \ldots, r_\lambda\}, \lambda, r, n$

Output: ψ

ComputeGlobalMismatchMatrix (χ, R, λ, r, n)

for $i = 2$ *to* n **do**

  Mark all $\psi_\rho^{\rho^i}[j]$ as empty, $\forall_j 1 \leq j \leq \lambda$, and $\forall_{\rho,\rho^i} 0 \leq \rho, \rho^i \leq d$

  Add number $x = \delta \times r + \delta^i$ to list $\psi_\delta^{\delta^i}[1]$, where $\chi_\delta^{\delta^i}[1]$ is non-empty, and $\forall_{\delta,\delta^i} 0 \leq \delta, \delta^i \leq r_1$

  for $j = 2$ *to* λ **do**

    for $\rho = 0$ *to* d **do**

      for $\rho^i = 0$ *to* d **do**

        for $\delta = 0$ *to* $min(\rho, r_j)$ **do**

          for $\delta^i = 0$ *to* $min(\rho^i, r_j)$ **do**

            if $\psi_{\rho-\delta}^{\rho^i-\delta^i}[j-1]$ *and* $\chi_\delta^{\delta^i}[j]$ *are non-empty* **then**

              Append id of $\chi_\delta^{\delta^i}[j]$ to the list stored at $\psi_\rho^{\rho^i}[j]$

            end

          end

        end

      end

    end

  end

end

Filling the Pruned Substring Matrix For any sequence T^i, the Global Mismatch Matrix captures the prefix of all candidate motifs up to layer j where the number of mismatches with the $(j \times r)$-length prefix of T^i is $\leq d$. If $j = \lambda$, then the prefix belongs to an eligible

candidate motif with respect to T^i, otherwise not. Therefore, we want to retain only the former and remove all instances of the latter. The Pruned Substring Matrix is used to store the result of this operation.

Algorithm 4 uses the characteristic of an eligible candidate motif that it has an entry in the last layer of the Global Mismatch Matrix. More specifically, the id of the element of the Local Mismatch Matrix to which its last substring belongs to, is listed in $\psi_\rho^{\rho^i}[\lambda]$, where ρ and ρ^i denote the number of mismatches of the candidate motif with T^1 and T^i respectively. We start filling up the Pruned Substring Matrix from the last layer. All elements in this layer are marked as empty except for $\varphi_\rho^{\rho^i}[\lambda]$, where $\psi_\rho^{\rho^i}[\lambda]$ is non-empty, $\forall_{\rho,\rho^i} 0 \leq \rho, \rho^i \leq d$. Then, we work backwards filling in the entries for layer $\lambda - 1$, and so on. We append the id of $\chi_\delta^{\delta^i}[\lambda]$ to the list stored at element $\varphi_{\rho-\delta}^{\rho^i-\delta^i}[\lambda-1]$, if $\psi_{\rho-\delta}^{\rho^i-\delta^i}[\lambda-1]$, $\chi_\delta^{\delta^i}[\lambda]$ and $\varphi_\rho^{\rho^i}[\lambda]$ are non-empty; and $\rho \geq \delta$ and $\rho^i \geq \delta^i$. We continue this back-tracking process layer-by-layer till we have generated entries for all elements in layer 1.

For any eligible candidate motif with respect to T^i, if its substrings from layers 1 to λ have $\delta[1], \delta[2], \ldots, \delta[j], \ldots, \delta[\lambda]$ mismatches with $\tau_1^1, \tau_2^1, \ldots, \tau_j^1, \ldots, \tau_\lambda^1$, and $\delta^i[1], \delta^i[2]$, $\ldots, \delta^i[j], \ldots, \delta^i[\lambda]$ mismatches with $\tau_1^i, \tau_2^i, \ldots, \tau_j^i, \ldots, \tau_\lambda^i$; then its first substring is listed in $\chi_{\delta[1]}^{\delta^i[1]}[1]$, and the ids of $\chi_{\delta[2]}^{\delta^i[2]}[2], \ldots, \chi_{\delta[j]}^{\delta^i[j]}[j], \ldots, \chi_{\delta[\lambda]}^{\delta^i[\lambda]}[\lambda]$ are listed in $\varphi_{\rho[1]}^{\rho^i[1]}[1], \ldots,$ $\varphi_{\rho[j-1]}^{\rho^i[j-1]}[j-1], \ldots \varphi_{\rho[\lambda]}^{\rho^i[\lambda-1]}[\lambda-1]$, where $\rho[j] = \sum_{k=1}^{j} \delta[j] \leq d$ and $\rho^i[j] = \sum_{k=1}^{j} \delta^i[j] \leq d$.

5.5.5 Phase 2: Identifying Actual Motifs from Eligible Candidates

5.5.5.1 Data Structures Required

- **Individual Substring Matrix (Φ)**: is a 3-dimensional matrix with axes i, j and ρ, where $2 \leq i \leq n$, $1 \leq j \leq \lambda$, and $0 \leq \rho \leq d$. Element $\Phi[i][j][\rho]$ from layer j is denoted as $\Phi_\rho^i[j]$ and lists all unique j^{th} substrings of eligible candidate motifs with respect to a sequence T^i, whose $(j \times r)$-length prefix has ρ mismatches with $(j \times r)$-length prefix of T^1.

- **Shared Substring Matrix (Γ)**: is a 2-dimensional matrix with axes j and ρ, where

Algorithm 4: Storing Only Eligible candidate motifs With Respect to Each Individual Sequence $T^i \; \forall_i 2 \leq i \leq n$ in the Pruned Substring Matrix.

Input : $\chi, \psi, R = \{r_1, r_2, \ldots, r_\lambda\}, \lambda, r, n$

Output: φ

ComputePrunedSubstringMatrix(χ, R, λ, r, n)

for $i = 2$ *to* n **do**

  Initialize all $\varphi_\rho^{\rho^i}[j]$ to empty lists, $\forall_j 1 \leq j < \lambda$, and $\forall_{\rho,\rho^i} 0 \leq \rho, \rho^i \leq d$

  Mark $\varphi_\rho^{\rho^i}[\lambda]$ as non-empty if $\psi_\rho^{\rho^i}[\lambda]$is non-empty, $\forall_{\rho,\rho^i} 0 \leq \rho, \rho^i \leq d$

  for $j = (\lambda - 1)$ *to* 1 **do**

    for $\rho = 0$ *to* d **do**

      for $\rho^i = 0$ *to* d **do**

        for $\delta = 0$ *to* $min(\rho, r_{j+1})$ **do**

          for $\delta^i = 0$ *to* $min(\rho^i, r_{j+1})$ **do**

            if $\psi_{\rho-\delta}^{\rho^i-\delta^i}[j]$, $\chi_\delta^{\delta^i}[j+1]$ *and* $\varphi_\rho^{\rho^i}[j+1]$ *are non-empty* **then**

              Append id of $\chi_\delta^{\delta^i}[j+1]$ to list stored at $\varphi_{\rho-\delta}^{\rho^i-\delta^i}[j]$

            end

          end

        end

      end

    end

  end

end

$1 \leq j \leq \lambda$ and $0 \leq \rho \leq d$. Element $\Gamma[j][\rho]$ from layer j is denoted as $\Gamma_\rho[j]$ and lists all common substrings present in $\Phi^i_\rho[j]$ $\forall_i 2 \leq i \leq n$. Each entry in this list is represented as $\Gamma_\rho[j][\kappa]$, where $1 \leq \kappa \leq |\Gamma_\rho[j]|$.

- **Successor Matrix (ζ)**: is a 2-dimensional matrix associated with every entry $\Gamma_\rho[j][\kappa]$ in the Shared Substring Matrix. Element $\zeta[i][\rho^i]$ of this matrix is denoted by $\Gamma_\rho[j][\kappa].\zeta^{\rho^i}$ where $1 \leq i \leq \lambda$ and $0 \leq \rho^i \leq d$ .It lists all values of κ', where $\Gamma_\rho[j][\kappa]$ and $\Gamma_{\rho+\delta}[j+1][\kappa']$ are the j^{th} and $(j+1)^{th}$ substrings of eligible candidate motif(s) with respect to T^i, whose $(j \times r)$-length prefix has ρ mismatches with the $(j \times r)$-length prefix of T^1 and ρ^i mismatches with the $(j \times r)$-length prefix of T^i.

- **Tolerance Array(Δ)**: has n entries where the i^{th} entry, denoted as Δ_i stores the number of mismatches between equal-length prefixes of an eligible candidate motif and the sequence T^i.

- **Minimum Tolerance Array(ϑ)**: is an array of length n associated with every element of the Shared Substring Matrix, and depicted as $\Gamma_\rho[j][\kappa].\vartheta$. Entries of this array store the values of corresponding entries of the Tolerance array from a previous traversal, when choosing $\Gamma_\rho[j][\kappa]$ as the j^{th} substring failed to yield an actual motif.

5.5.5.2 Filling Up the Data Structures

The Individual Substring Matrix is filled up as follows. For $j = 1$ and $\forall_i 2 \leq i \leq n$ element $\Phi^i_\delta[j]$ lists all substrings in $\chi^{\delta^i}_\delta[j]$, where $\varphi^{\delta^i}_\delta[j]$ is non-empty. For $2 \leq j \leq \lambda$, $\Phi^i_\rho[j]$ lists all substrings from elements $\chi^{\delta^i}_\delta[j]$ whose ids are listed in $\varphi^{\rho^i}_\rho[j-1]$, $\forall_{\rho,\rho^i} 0 \leq \rho, \rho^i \leq d$, $\forall_{\delta,\delta^i} 0 \leq \delta, \delta^i \leq r$ and $\forall_i 2 \leq i \leq n$.

Then, entries of the element $\Gamma_\rho[j]$ of the Shared Substring Matrix are computed by listing only the substrings which are common to $\Phi^i_\rho[j]$, $\forall_i 2 \leq i \leq n$. Finally, entries for element $\Gamma_\rho[j][\kappa].\zeta^{\rho^i}$ are calculated by finding all values of κ' where the substring label $\Gamma_{\rho+\delta}[j+1][\kappa']$ is listed in element $\chi^{\delta^i}_{\delta'}[j+1]$, the id of $\chi^{\delta^i}_{\delta'}[j+1]$ is stored in $\varphi^{\rho^i}_\rho[j]$, and

number of mismatches between $\Gamma_\rho[j][\kappa]$ and τ_j^1 is δ. Finally, all entries in $\Gamma_\rho[j][\kappa].\vartheta_i$ are initialized to d before starting the following branch and bound algorithm.

5.5.5.3 Branch and Bound Algorithm

If M is an actual motif sequence, then its substrings $\mu_1, \mu_2, \ldots \mu_\lambda$ appear in layers 1 to λ of the Shared Substring Matrix, such that μ_{j+1} appears in the Successor Matrix of μ_j with respect to every sequence T^i, $\forall_i 2 \leq i \leq n$ and $\forall_j 1 \leq j \leq \lambda - 1$. Algorithm 5 searches for these motif(s) in a recursive manner starting from layer 1 of the Shared Substring Matrix. For each substring in $\Gamma_\rho[1]$, we find all common successor substrings in $\Gamma_\rho[2]$ with respect to all sequences T^i $\forall_i 2 \leq i \leq n$. In turn, for each of these common successors, we find their common successors from $\Gamma_\rho[3]$, and so on. This continues till there are no common successors at a layer $j < \lambda$, or we have found common successor(s) at layer $j = \lambda$, i.e. actual motif(s).

First, we initialize all entries of the Tolerance Array by setting $\Delta_i = 0$, $\forall_i 1 \leq i \leq n$. As we choose a substring from layer j, we add the number of mismatches between that substring and τ_i to Δ_i. The values of Δ_i are nothing but coordinates on the ρ_i axis of the Successor Matrix and help in identifying the candidate successor substrings from layer $j+1$ with respect to sequence T^i.

If the choice of a substring from layer j fails to generate an actual motif, with a set of values for entries in the Tolerance Array Δ, then these values are stored in the Minimum Tolerance Array associated with that substring. If the recursion returns to this substring during a future traversal, then the values stored in this Minimum Tolerance Array are used to bound the search by checking against the current values stored in the Tolerance Array.

Algorithm 5: Find all actual motifs from eligible candidate motifs

Input : $\Gamma, L, \Delta, prefix, j, n, \lambda$

Output: SUCCESS, FAILURE

RecursiveMotifSearch$(\Gamma, L, \Delta, prefix, j, n, \lambda)$ $returnFlag = FAILURE$

if $\Delta_i \leq \Gamma_\rho[j][\kappa].\vartheta_i$, $\forall_i 1 \leq i \leq n$ **then**

  if $j = \lambda$ **then**

    if $\Delta_i \leq d$, $\forall_i 1 \leq i \leq n$ **then**

      Output $prefix$ as actual motif, and set $returnFlag = SUCCESS$

    end

  end

  else

    $L = FindCommonSuccessors(\rho, \kappa, j, \Gamma, \Delta)$

    for $\iota = 1$ *to* $|L|$ **do**

      $\Delta'_i = \Delta_i + H(L_\iota, \tau^i_{j+1})$ $\forall_i 1 \leq i \leq n$

      $\rho' = H(\Gamma_\rho[j][\kappa], \tau^1_j)$

      Append $\Gamma'_\rho[j+1][L_\iota]$ to $prefix$ to generate $prefix'$

      $flag = RecursiveMotifSearch(\rho', L_\iota, j+1, \Gamma, \Delta', prefix', n, \lambda)$

      if $flag = SUCCESS$ **then**
        $returnFlag = SUCCESS$

      end

    end

  end

  if $returnFlag = FAILURE$ **then**

    $\Gamma_\rho[j][\kappa].\vartheta_i = \Delta_i$, $\forall_i 1 \leq i \leq n$

  end

end

Return returnFlag

Algorithm 5: (...contd) Find all actual motifs from eligible candidate motifs

Input : $\Gamma, L, \Delta, prefix, n, \lambda$

Output: All actual motifs occurring in $S^1, S^2, \ldots S^n$

StartMotifSearch(Φ, Γ, d, n)

Initialize L as an empty list

for $\rho = 1$ *to* d **do**

  if $|\Gamma_\rho[1]| > 0$ **then**

    for $\kappa = 1$ *to* $|\Gamma_\rho[1]|$ **do**

      $\Delta_i = H(\Gamma_\rho[1][\kappa], \tau_1^i)\ \forall_i 1 \leq i \leq n$

      $prefix = \Gamma_\rho[1][\kappa]$

      Append tuple (ρ, κ) to L

      $RecursiveMotifSearch(\Gamma, L, \Delta, prefix,\ 1, n, \lambda)$

    end

  end

end

Input : $\rho, \kappa, j, \Gamma, \Delta$

Output: List of entries in $\Gamma_\rho[j+1]$

FindCommonSuccessors$(\rho, \kappa, j, \Gamma, \Delta)$

Initialize empty list L

Append κ' to L, where $\kappa' \in \Gamma_\rho[j][\kappa].\zeta^{\Delta^i}\ \forall_i 2 \leq i \leq n$

Return L

5.6 Performance Estimation

The AP is expected to be released soon. Hardware design has progressed to the point of initial prototyping and accurate estimates for fundamental operations and streaming bandwidth are available. As the chip is currently unavailable, it is not possible to report experimental results of actual runs conducted on the chip. Despite this limitation, the techniques presented in this paper are valuable to the community for early exploitation of the AP. In addition, we can provide accurate and compelling estimates of run-time as described below.

As mentioned earlier, the software ecosystem for the processor is already in place including a Software Development Kit, and two programming methodologies – 1) regular expressions expressed in Perl Compatible Regular Expression (PCRE), or 2) automata expressed using the language ANML exclusively developed for this purpose, and associate compilers. No cycle level simulator is currently available, but a simulator that visualizes the execution of small size automata is available. Even if a cycle level simulator was available, CPU-based simulations are not feasible for the large problem instances that we have demonstrated to be solvable on the AP.

First, we verified the programmability of the solution on the chip using the already developed programming language ANML, and ensured proper compilation. We further verified accurate execution of components such as the Bounded Mismatch Identification automaton, as they are within the size allowed by the simulator. We then estimated runtimes based on well established costs for operations like STE label assignment time, automata load time, streaming rate, and output handling time. We also make pessimistic assumptions when exact estimates are unavailable or data dependent operations exist in our code.

Our analysis is presented for an AP board attached to a single CPU through a PCIe slot. We choose 20 randomly generated DNA sequences of length 600, followed by planting of a randomly generated motif. We present results for two of the largest challenging instances in published literature – $(25, 10)$ and $(26, 11)$ solved on a 48-core system

by PMS8 [56] – and demonstrate numerous larger instance that could not be solved yet – $(28,12)$, $(30,13)$, $(32,14)$, $(34,15)$, $(35,16)$, $(36,16)$, $(37,16)$, $(37,17)$, $(38,17)$, $(39,17)$, $(39,18)$, and $(40,17)$.

5.6.1 Established run-time Features of the Automata Processor

The AP board has 6 ranks, each having 8 processors. When time on an AP (or rank) is specified, all 48 processors (or 6 ranks) can operate in parallel, hence the same time estimate applies for the entire board.

Assignment of labels to all STEs of precompiled automata which span an entire rank of AP and loading the same takes at most 50 milliseconds. Once the automata is loaded, 6 data flows can be streamed to 6 ranks on the AP board in parallel at the rate of 1 Gbps per data-flow, for a cumulative data-processing rate of 6 Gbps.

Matching of any reporting STE triggers an output of the state of all reporting STEs. It takes one clock cycle to record output in an internal buffer but up to 240 AP clock cycles to transfer the output out of the buffer. During this time, the AP must stall. To mitigate this, a buffer that can hold 1024 output vectors is provided on the AP. In the ideal case, this expanded buffer space and infrequent output generation can completely eliminate the stall. Here, we assume the worst case and charge 240 clock cycles every time an output event is triggered.

Other considerations include the memory required to store the image to be loaded onto the AP and STE utilization efficiency when automata is mapped onto the physical processor chip. The size of an image spanning an entire rank of the AP is about 25 MB.

5.6.2 Execution Time for Finding n-cliques

The maximum number of nodes in the graph G is $20 * 600$, realized when no sequence contains an l-mer repeat. The size of the automaton for each node is $(2d+1)l - 2d^2$. All of these automata do not fit simultaneously into a single AP board. Hence, the board is reprogrammed in iterations with different subsets of automata. For each iteration, the same

Table 2: Estimated run-times using a single AP board for Stage 1: Finding n-cliques, and actual run-times using a CPU implementation for Stage 2: Searching for motifs consistent with cliques.

	Stage 1		**Stage 2**			
(l,d)	Number of nodes per iteration	Clique finding time minutes	Maximum time in minutes	Minimum time in minutes	Average time in minutes	Median time in minutes
$(25,10)$	5737	12.26	0.0037	0.0003	0.00010	0.0008
$(26,11)$	5243	13.96	0.0138	0.0006	0.00285	0.0021
$(28,12)$	4537	17.34	0.0631	0.0011	0.01099	0.0068
$(30,13)$	3965	21.21	0.1672	0.0035	0.04564	0.0328
$(32,14)$	3495	25.63	0.5766	0.0134	0.13029	0.0775
$(34,15)$	3104	30.62	6.0824	0.0411	0.86926	0.4541
$(35,16)$	2917	33.57	41.4650	0.1417	9.14443	5.4058
$(36,16)$	2776	36.22	16.3369	0.1023	3.49089	1.9229
$(37,16)$	2647	38.96	33.6622	0.3289	3.06252	1.7130
$(37,17)$	2618	39.50	116.8098	1.0056	37.58410	30.3302
$(38,17)$	2497	42.46	112.6668	1.1899	18.44524	7.8159
$(39,17)$	2386	45.51	52.0210	0.2940	7.37592	3.7162
$(39,18)$	2362	46.08	1031.6840	6.1580	180.89054	121.6833
$(40,17)$	2285	48.67	22.9254	0.3317	4.03950	2.3128

data-flow is streamed. The execution time can be estimated by computing the number of times the AP board has to be reprogrammed to fit all the automata. Due to hardware constraints, the AP board cannot fully utilize all of its physical STEs. The run-time estimates assuming a 80% STE utilization efficiency are shown in the second column of Stage 1 in Table 2.

For our calculation, we make the unrealistic worst case assumption that all nodes survive all iterations. We divide the number of nodes with the maximum number whose collective automata are programmable on a board, and multiply the run-time by this factor. The number of buckets and the size of the cliques are fixed for every iteration. The number of cliques per bucket however is based on the input sequences. We estimate this number to be 500 per iteration, larger than what published algorithms consider solving [56]. Output is generated only once at the end of the data stream while identifying nodes that will be retained for the next iteration, whereas it is generated at the end of each clique when forming cliques for the next iteration. Knowing clique sizes for each iteration and the value of l, we can precisely estimate the stalls expected due to output. On the CPU side, we have to update bucket ids and just assign pointers to create a logical data flow for the next iteration. This can be easily completed while the data flow is being streamed. The time to find 20-cliques for different (l,d) pairs is given in Table 2. In comparison, the best known exact algorithm requires 15.5 hours to solve instances of the $(25,10)$ problem on a single core machine, and 21 minutes and 46.9 hours to solve instances of the $(25,10)$ and $(26,11)$ problems on a 48-core machine [56].

5.6.3 Execution Time for Motif Discovery from Cliques

In order to evaluate the execution time of identifying motifs consistent with n-cliques, we simulate the input by creating random n-cliques. For testing an instance of (l,d), first a motif M of length l is created by randomly choosing letters from the alphabet $\{a,c,g,t\}$. Then, each sequence corresponding to a node in the clique is randomly generated having

length l and between $d-3$ and d mismatches with M. The location of the mismatches and the characters at those locations are also randomly chosen. The algorithm is presented with the n sequences, l, d and $r = 4$. At the end of the execution, we test whether M is one of the reported motifs and if all other reported motifs are indeed actual motifs.

We ran the test on a single quad-core *Intel(R) Core(TM) i5-3570* processor, running at 3.4 GHz and with 8 GB of main memory. The algorithm is embarrassingly parallel and OpenMP was used to run the code in parallel on all the cores. For each value of (l, d), we ran the tests on 50 different cliques. As expected the execution time varied widely based on the input clique. The maximum, minimum, average and median execution times in minutes are presented in Table 2.

5.7 Summary

The AP can be programmed to identify thousands of patterns present in a data stream in parallel. We used this capability to develop an exact algorithm for solving the Planted Motif Search problem and the Quorum Planted Motif Search Problem. Along with demonstrating the capability to solve large instances of this problem quickly, when compared to the state of the art, this application showed that one can cast graph problems into string pattern-matching problems which can then be accelerated using the AP. This laid the foundation for solving other graph problems using the AP which we shall see in the upcoming chapters.

Chapter VI

SOLVING GRAPH PROBLEMS - I

In the last chapter, we looked at the first non-intuitive use of the AP, to solve a a graph problem. In this chapter, and the next, we continue to look at similar problems using unweighted graphs.

First, we look at problems where the input itself is not a graph, but is transformed into one by the algorithm. The planted motif search problem discussed in the last chapter falls under this category. The input is a bunch of sequences, subsequences of which are represented as nodes in a graph. A condition is provided for the existence of an edge between two nodes, but the edges themselves are not provided. The objective is to find a set of nodes which are all connected to one another, i.e. a clique. In general, we can classify these problems discovery of edges in a *specialized* graph, such that all the connectivity constraints of an output graph structure are satisfied. We will discuss the speciality of the graph momentarily.

In the next chapter, we look at problems where the edges are already part of the input. Therefore, edge discovery is not part of the problem. In fact, the automata are designed on the basis of the edges in the graph, and are called *edge-automata*. The design of the automata is such that primary challenge of programming any generic graph quickly and efficiently into the AP is overcome. Using these automata, some fundamental algorithmic techniques have been developed to organize the edges (and hence the nodes) in the input graph into output graph structures such as search-trees, acyclic paths, etc.

6.1 Background

In the last chapter, we solved the problem of finding the largest clique in a specialized input graph. The speciality of the graph is that its nodes are distributed into disjoint *partitions*,

wherein no two nodes from the same partition are connected to each other. In an input graph with n partitions, the problem is to find all cliques of size n. If such a clique is present, then it must be a maximal clique in the graph. This is because each node in any clique in the graph must be drawn from a separate partition in the graph, and hence the size of the largest clique can only grow till n. In this chapter, we discuss another problem which reduces to the same problem.

6.1.1 Boolean Satisfiability (SAT) Problem

The boolean satisfiability (SAT) problem is the first known NP-complete problem. In fact, it lies at the very basis of the definition [18, 43] of this class of problems. It finds use in a wide variety of real world problems [51] such as digital circuit design, term-rewrite systems, model-checking, artificial intelligence, haplotype inference, etc.

This well-known decision problem deals with determining whether a boolean formula made up of boolean variables $x_1, x_2, \ldots x_m$ is *satisfiable* (evaluates to $TRUE$) for any assignment of values to the variables. In its *Conjunctive Normal Form* (CNF), the formula is structured as *conjunction* (*AND*, denoted as $\wedge$) of *clauses*. Each clause is demarcated using a pair of parentheses and is made up of *disjunction* (OR, denoted by $\vee$) of variables. Each variable x_i can be present as itself, called a *positive literal*, or its *negation* (NOT, denoted as $\neg$). In the latter case, $\neg x_i$ is called a *negative literal*. For example, the boolean formula $(x \vee y) \wedge (\neg x \vee \neg y) \wedge (x \vee \neg y)$, is satisfiable by assigning the values of $TRUE$ and $FALSE$ to variables x and y respectively, as $(TRUE \vee FALSE) \wedge (\neg TRUE \vee \neg FALSE) \wedge (TRUE \vee \neg FALSE)$ evaluates to $TRUE$.

The SAT problem can be solved by reducing it to the maximal clique finding problem. For example, the boolean formula discussed above can be transformed into the graph shown in Figure 34. Every clause in the boolean formula is expressed as a partition in the graph. Every literal in the clause is denoted as a node in the partition. An undirected edge is drawn between two nodes only if they belong to two different partitions and the conjunction of

the literals they represent, is satisfiable. For example, an edge is drawn between nodes representing x and $\neg y$ because $x \wedge \neg y$ is satisfied with the assignment of $x = TRUE$ and $y = FALSE$. On the other hand, no edge is drawn between nodes representing x and $\neg x$ because $x \wedge \neg x$ is not satisfiable for any value of x.

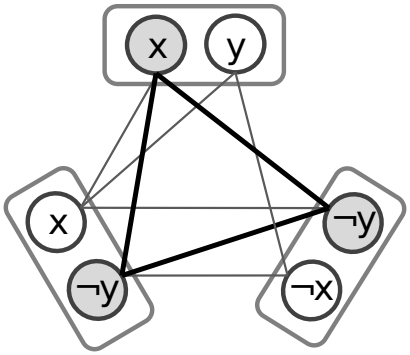


Figure 34: Boolean formula $(x \vee y) \wedge (\neg x \vee \neg y) \wedge (x \vee \neg y)$ reduced to maximal clique finding problem.

If the number of clauses in the formula is n, then it can be proven that if the graph has an n-clique, then the boolean formula is satisfiable. This is because, the nodes in a clique represent literals from different clauses which can be mutually satisfied. Since a clique of size n must draw one node from each partition of the graph corresponding to one literal from each clause in the formula, mutual satisfiability of all the literals leads to the mutual satisfiability of all the clauses, and hence the entire formula.

Each clique may contain either the positive or the negative literal of a variable. If the positive literal is present, then the variable should be assigned the value of $TRUE$ in the solution. Otherwise, it should be assigned the value of $FALSE$. If neither the positive nor the negative literal of the variable is present in the clique, then both values can be assigned to the variable in the solution. For example, for the graph shown in Figure 34, nodes and edges of an n-clique are outlined in dark circles and bold lines respectively. Since the clique contains the literals x and $\neg y$, the assignment of the values $TRUE$ and $FALSE$ to the variables x and y respectively provides a solution to the formula.

Many solvers exist for the SAT problem. However, the performance of all of them degrades to exponential time in the worst cases. For example, in 2011, at the annual *International SAT Solver Competition*, the smallest instance of the problem which could not be solved by any solver within the stipulated time (of 5000 seconds) contained only 141 variables, 292 clauses, and 876 literals [42]. On the other hand, the biggest instance which was solved by at least one solver contained 10 million variables, 32 million clauses and 76 million literals. Our solution does not compete with these solutions, rather compliments them to solve small-sized but hard to compute instances.

Our method finds all the n-cliques in the graph simultaneously. Therefore, it naturally solves the $UNSAT$ problem, which is an NP-hard problem that answers the question whether a boolean formula is not satisfiable for any possible assignment of values to its variables. This is because, corresponding to every possible assignment for which the formula is satisfied, there is an n-clique in the graph. This can be proven as follows. For every satisfiable assignment, at least one literal from each clause is satisfied. Also, none of these literals are complementary to one another. Therefore, the nodes corresponding to the literals in the graph are mutually connected to one another forming an n-clique. Thus, if our method does not find any n-cliques in the graph, then the formula does not have any satisfiable assignment.

6.1.2 Modified Requirement Specification

We have already solved the problem of finding n-cliques in an n-partitioned graph in the MOTOMATA application. However that solution does not provide a viable solver for the SAT problem because of the following differences.

First of all, in MOTOMATA nodes represented strings, but in the SAT problem, we have to define string labels for all literals in the boolean formula. This is described in Section 6.2.2.

Second, the criteria for the existence of an edge is different in both the applications. In

MOTOMATA, an edge represents two subsequences which have up to a fixed Hamming distance between them. On the other hand, an edge in the SAT problem represents two literals which are not *complementary*. Two literals are called complementary if they are the positive and negative literals of the same variable. Therefore, the automata must also be redesigned. Although, the overall design principles are the same, the automata for the SAT problem are significantly smaller. This is described in Section 6.2.2.

Finally, the number of partitions in the graph and the number of nodes in partition are significantly different for both the problems. In the planted motif search problem which MOTOMATA solves, the number of partitions and the number of nodes per partition are 20 and 600 respectively for the benchmark application. In the SAT problem, the number of partitions is expected to be much higher, while the number of nodes per partition is expected to be much smaller.

In MOTOMATA, the nodes are programmed as automata, and the cliques are streamed. In each iteration, many nodes are pruned away, and the remaining nodes are combined with the cliques to increment their size by one for the next iteration. Starting with cliques of size one (nodes themselves), this method requires $n-1$ iterations to find all the n-cliques in the graph.

At iteration i, the cliques from the same set of i partitions are organized into a bucket. Therefore, there are $\binom{n}{i}$ buckets in iteration i. Cumulatively, the total number of buckets streamed in the $n-1$ iterations is $\binom{n}{1}+\binom{n}{2}+\ldots+\binom{n}{n-1}=2^n-\binom{n}{0}-\binom{n}{n}=2^n-2$. Since we assume that the average number of cliques per bucket over all the iterations is 500 (far above all competing solutions), the number of cliques to be streamed over the entire process is $O(2^n)$. This makes the run-time of MOTOMATA exponential in n. Although, this is manageable for the planted motif search problem where $n=20$, this is not a viable solution for the SAT problem where the value of n is typically much larger.

6.1.3 Overview of Changes

Our method solves instances of the SAT problem where the number of variables is at most 256. The number of literals and clauses in the formula can be much larger. In the new solution, we limit the number of iterations required to reach the solution to $\lceil log\ n \rceil$ by designing compact *clique-automata* which can be used to combine cliques with cliques, thus doubling their size in every iteration. This was not possible in the case of MOTOMATA as the design of such compact clique-automata was not feasible for that application.

The other modification is the reduction in the number of cliques handled in every iteration. In MOTOMATA, in iteration i, i-cliques from all possible combination of i partitions in the graph were streamed. This allowed us to prune the nodes in the graph more aggressively and provide a solution to the more generic quorum motif search problem. However, this method has to handle a large number of cliques. We limit this in our new algorithm by dividing the partitions into disjoint sets. Then, cliques from one partition is combined with the cliques from only one other partition leading to significantly less processing time.

6.2 Methodology

Let G denote the graph generated by transforming the boolean formula and $P_1, P_2, \ldots, P_n$ denote the partitions in G. Two cliques C_1 and C_2 are called *neighboring* if every node in C_1 is adjacent to every node in C_2. The nodes in two neighboring cliques can be simply combined to generate a larger clique. We have the following two observations.

Observation 6. *If an n-clique exists in G, then $P_1, P_2, \ldots, P_{\lceil n/2 \rceil}$ contains a $\lceil n/2 \rceil$-clique and $P_{\lceil n/2 \rceil+1}, P_{\lceil n/2 \rceil+2}, \ldots, P_n$ contains a $\lfloor n/2 \rfloor$-clique.*

Proof. If C is an n-clique in G, then it contains a node from each partition in G. The nodes in C from $P_1, P_2, \ldots, P_{\lceil n/2 \rceil}$ are all mutually adjacent to each other, and hence form a $\lceil n/2 \rceil$-clique. Similarly, the nodes in C from $P_{\lceil n/2 \rceil+1}, P_{\lceil n/2 \rceil+2}, \ldots, P_n$ are mutually adjacent to each other and form a $\lfloor n/2 \rfloor$-clique. □

Observation 7. *If C_1 is a subclique of an n-clique C made of nodes drawn from partitions $P_1, P_2, \ldots, P_{\lceil n/2 \rceil}$, and C_2 is a subclique of C made of nodes drawn from partitions $P_{\lceil n/2 \rceil+1}, P_{\lceil n/2 \rceil+2}, \ldots, P_n$, then C_1 and C_2 are neighboring cliques.*

Proof. For every node $u_1 \in C_1$ and $u_2 \in C_2$, $\{u_1, u_2\} \in C$. Therefore, u_1 must connected to u_2, and C_1 and C_2 must be neighboring cliques. □

6.2.1 Overview of the Algorithm

Using the above observations, Algorithm 6 works by calling the function *RecursiveSAT* with all the partitions in G. It recursively divides the problem into two parts: finding all $\lceil n/2 \rceil$-cliques in $P_1, P_2, \ldots, P_{\lceil n/2 \rceil}$ and all $\lfloor n/2 \rfloor$-cliques in $P_{\lceil n/2 \rceil+1}, P_{\lceil n/2 \rceil+2}, \ldots, P_n$. Cliques of size n are then generated by combining cliques from the first part which are neighboring to the cliques from the second part. The recursion stops, when the function is called with a single partition, in which case, all nodes in the partition are returned as 1-cliques. At any stage, if no $\lceil n/2 \rceil$-cliques are present in the first part, or no $\lfloor n/2 \rfloor$-cliques are present in the second part, the functions outputs that there are no n-cliques in the graph and quits.

The identification of cliques from $P_1, P_2, \ldots, P_{\lceil n/2 \rceil}$ which can be combined with cliques from $P_{\lceil n/2 \rceil+1}, P_{\lceil n/2 \rceil+2}, \ldots, P_n$ is accelerated using the AP. In order to compute this, each clique from $P_1, P_2, \ldots, P_{\lceil n/2 \rceil}$ is programmed as a clique automaton, and all cliques from $P_{\lceil n/2 \rceil+1}$, $P_{\lceil n/2 \rceil+2}, \ldots, P_n$ are streamed. Since the size of the clique-automaton is quite small, a large number of automata can be executed in parallel to identify cliques from $P_1, P_2, \ldots, P_{\lceil n/2 \rceil}$ which are neighboring to a clique being streamed.

6.2.2 Automata Design

In order to generate the string representation of the graph, each node in the graph is assigned a string label which can then be parsed using automata. This is done on the basis of the literal that the node represents. Each variable in the formula is assigned an unique integer id between 0 and $m-1$. Each literal in the formula is then represented by a two-byte string

Algorithm 6: Recursive algorithm to find n-cliques in graph G.

Input : $P_1, P_2, \ldots, P_k$

Output: List of all k-cliques in $P_1, P_2, \ldots, P_k$

RecursiveSAT $(P_1, P_2, \ldots, P_k)$

if $k = 1$ **then**

    Return all nodes in P_1 as 1-cliques

end

else

    Calculate $k' = \lceil k/2 \rceil$

    {Identify all $\lceil k/2 \rceil$-clique in the first part.}

    L_1 = RecursiveSAT $(P_1, P_2, \ldots, P_{k'})$

    {Identify all $\lfloor k/2 \rfloor$-clique in the second part.}

    L_2 = RecursiveSAT $(P_{k'+1}, P_{k'+2}, \ldots, P_k)$

    if L_1 *or* L_2 *is empty* **then**

        Stop and report no cliques.

    end

    else

        L_3 = List k-cliques generated by combining cliques $C_1 \in L_1$ and $C_2 \in L_2$,

        where C_1 and C_2 are neighboring cliques

        Return L_3

    end

end

label. The first byte in the string label of a positive literal is set to 0, whereas in the case of a negative literal, it is set to 1. The second byte is set to the id of the corresponding variable.

The data-flow is created for every recursive call using the cliques in list L_2. The data-flow contains a sequence of cliques in the list separated by a one-byte delimiter set to 255. The string labels of the nodes within a clique are simply concatenated one after the other.

A simple clique automaton is shown in Figure 35. The literals corresponding to nodes in a clique C are used to generate the labels of $STE2$ and $STE4$. For short, we write $x_i \in C$, if C contains a node corresponding to literal x_i. Similarly, we write $\neg x_j \in C$, if node corresponding to literal $\neg x_j$ is in C. X and Y are two lists of variable ids such that $X = \{i; \forall x_i \in C\}$ and $Y = \{j; \forall \neg x_j \in C\}$.

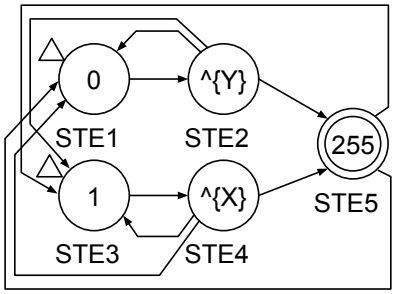


Figure 35: Clique-automaton.

$STE1$ and $STE2$ are used to match the string label of a node corresponding to any positive literal in clique being streamed, namely C_s. $STE2$ activates $STE1$ and $STE3$ if the literal is not complementary to any negative literal $\neg x_j \in C$. Similarly, $STE3$ and $STE4$ are used to match the string label of a node corresponding to any negative literal in C_s. $STE4$ activates $STE1$ and $STE3$ if the literal is not complementary to any positive literal $x_i \in C$.

As long as no string labels in C_s are encountered which belongs to a literal complementary to any literal in C, $STE1$ and $STE3$ are activated by either $STE2$ or $STE4$ to check the next string label. If the string labels of all the nodes in C_s are non-complementary to all of the literals in C, then $STE5$ matches the delimiter streamed at the end of streaming C_s, and an output is generated. This signifies that C and C_s are neigboring cliques and should be combined to generate a larger clique.

The combined clique is maintained on the CPU. The automata for the combined clique differs from that of C in only the labels of $STE2$ and $STE4$. The new labels of $STE2$ and $STE4$ can be generated by adding the variable ids of negative literals in C_s to Y, and that of the positive literals in C_s to X.

6.2.3 Macro Design

The automaton for all cliques in the algorithm differ only in the labels of $STE2$ and $STE4$. Therefore, such an automaton is amenable to be defined as a macro. However, whenever a macro is pre-compiled it is mapped to complete *logical-row(s)* of an AP-chip. That is, at the time of loading this macro, only a mapping of the logical-rows in the pre-compiled image to the physical rows in the AP-chip has to be completed. Therefore, macro for a single clique requiring only five STEs is compiled and mapped to 16 physical STEs.

A more efficient macro design encapsulates automata for three cliques into a single macro as shown in Figure 36. Parameters $p2$, $p6$ and $p10$ correspond to list Y of the first, second and third cliques respectively. Similarly, parameters $p4$, $p8$ and $p12$ capture list X of the first, second and third cliques respectively. Parameters $p1$, $p5$ and $p9$ are set to 0, whereas $p3$, $p7$ and $p11$ are set to 1.

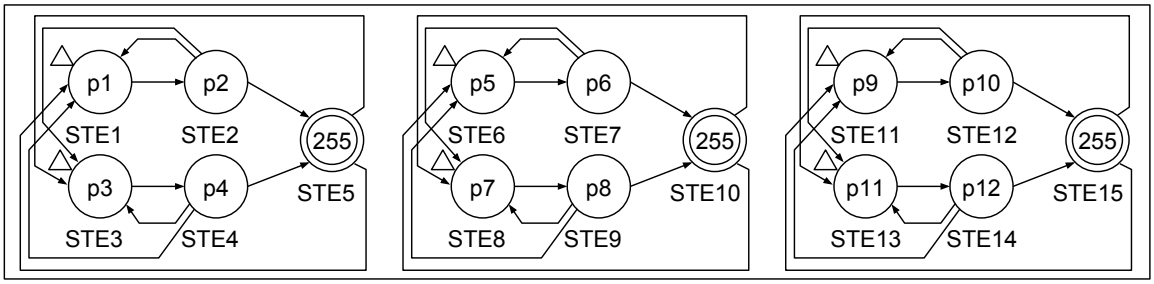


Figure 36: Programming three cliques as a macro.

If the number of cliques is not a multiple of the three, then one of the macros does not have three cliques left to be programmed. Without loss of generality, let us consider the case where there are only two cliques left to be programmed. In such a case $p9$ and $p10$ are set to 255. Therefore, STEs for the third automaton are deactivated after processing the

very first symbol of the data-flow.

6.3 Performance Analysis

The actual run-time of the application depends on the instance of the problem being solved. As this run-time cannot be simulated easily, we have with-held the analysis of the actual run-times till the processor becomes available in a few months time. However, in this section we provide the motivation for carrying out this exercise whenever the hardware becomes ready.

The number of times the AP board needs to be reprogrammed is based on the number of recursive calls made by the algorithm which is $\approx n$. Note that if the number of cliques to be programmed is larger than what can be fit inside the AP board at any given time, the cliques have to be divided into groups and programmed iteratively. This would lead to an increase in the total number of times the AP board needs to be programmed. None the less, programming of the AP board merely requires relabelling the STEs of the pre-compiled macros and loading them into the AP chips.

In every recursive function call, every literal in a clique being streamed is checked against every literal in all cliques programmed into the AP board in parallel. At any given time, each row in an AP chip can fit three cliques, giving rise to $9,216$ cliques per chip, and $442,368$ cliques per AP board (with 48 chips). If the programmed cliques are drawn from k partitions, the speed-up reaches $442,368 \times k$ times. This number is very significant when k equals $\lceil n/2 \rceil$, $\lceil n/4 \rceil$, etc. If this acceleration is summed over all recursive calls in the algorithm, the speed-up can be written as $442,368 \times (\lceil n/2 \rceil + 2 \times \lceil n/4 \rceil + 4 \times \lceil n/8 \rceil \cdots + 1) \approx 442,368 \times (n/2) log\ n$ times.

Although the process of identifying cliques to be merged on the AP, and the process of combining those cliques on the CPU, can be pipelined, the latter is expected to become a severe bottleneck in the face of the significant acceleration provided by the former. Therefore, ways to parallelize the CPU operations needs to be investigated further.

Some fine tuning can also be done for function calls when the number of cliques to be programmed into the AP is not significant. This can happen at the deepest levels of recursion, where the value of k is very small, i.e. k-cliques are drawn from a very small number of partitions. For example, at the deepest level, $k = 1$ and 1-cliques (nodes) from one partition are checked against 1-cliques (nodes) from another. Since the expected number of nodes per partition is typically not high, the cost of programming the AP with 1-cliques may be higher than simply carrying out the operations on the CPU itself. Therefore, a small cost-benefit analysis may be carried out before programming the AP with cliques especially for function calls where the value of k is very small.

6.4 Summary

SAT is a well-studied problem which is used in various fields of computing. Although, a wide variety of such algorithms and applications are available to solve this *NP*-complete problem, there are relatively small instances of the problem which remain unsolved. We have proposed a method to accelerate the search for solutions to such instances of the problem using the AP.

The SAT problem can be reduced to the problem of finding an n-clique in an n-partitioned graph, a problem that we had already solved in MOTOMATA. However, the differences in the characteristics of the input make the MOTOMATA solution non-viable as a SAT solver. In our new method, we have significantly reduced the number of iterations required to reach the solution, and the number of cliques streamed per iteration. This is done by neatly dividing the partitions in the graph into non-overlapping parts using recursion and then considering cliques from only these parts. The cliques from one part are combined with cliques of only one other part to generate larger cliques which are twice the original size. All these improvements have the potential to deliver significant acceleration using the AP, whenever the hardware becomes available.

Chapter VII

SOLVING GRAPH PROBLEMS - II

In the last chapter, we looked at providing a SAT solver which reduces the problem to finding a clique in the graph using the AP. However, the transformation of the input boolean formula to the graph gives a fixed structure to the graph i.e. the nodes in the graph are organized into partitions and the string representation of nodes in the graph can be derived from the input. More importantly, the edges of the graph are not available as part of input, but discovered over the course of building the solution. A similar problem was solved for MOTOMATA in Chapter 5. In both these problems, the discovery of edges in the graph to satisfy the connectivity constraints of the output graph structure, namely a clique, serves as the primary computational challenge.

In contrast, this chapter deals with problems in any generic unweighted graphs. The nodes and edges in this graph are part of the input. The primary computational challenge is to arrange the nodes and edges in the input graph into specific graph derived output structures like a set of acyclic paths and shortest path trees. Using the former technique, we provide algorithms for identifying all Hamiltonian paths and cycles in a graph. Using the latter technique, we provide algorithms for finding single-source shortest paths and connected components in a graph. The algorithmic techniques we present are foundational for providing streaming solutions to a wide variety of graph problems using the AP, especially when exploiting data-locality is difficult.

The following transformations are imperative for solving graph problems on the Automata Processor. First, the nodes and edges in the graph must be represented using strings which can be streamed to the co-processor. Second, the graph problem must be converted into a string-pattern matching problem which can be evaluated using automata over the

streaming data. We achieve these by defining fundamental techniques to (1) Represent nodes and edges of a generic graph using strings, and (2) Create efficient automata which can be used to solve a variety of graph problems.

7.1 Preliminaries

7.1.1 Representing A Generic Graph Using Strings

Let $G = (V, E)$ be the input graph where $|V| = n$. We assign each node a unique integer id between 0 and $n-1$. The string label of a node is obtained by converting its integer id to its base-256 representation. We then treat this as a string over the alphabet $\{0 \ldots 255\}$. This is because the AP can only process strings over an alphabet size of 256, containing symbols 0 through 255. A directed edge is represented as the concatenation of the string labels of its source and destination nodes. An undirected edge is represented using two directed edges in opposite directions. A path is represented by the concatenation of the string labels of the nodes along the path. Figure 37 shows the string labels of nodes and edges in an example directed graph with four nodes. The path $0 \rightarrow 2 \rightarrow 3 \rightarrow 1$ is represented by the string 0 2 3 1.

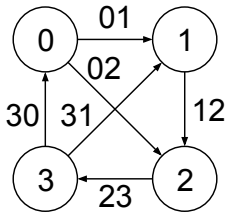


Figure 37: String labels of nodes and edges in a graph.

7.1.2 Automata Design

There is a gap between the elegant theoretical model supported by the AP and what can be supported efficiently in practice. To address this gap, we should design automata which is architecture-aware, and easier/faster to compile and execute. Such automata designs are generally built using basic building blocks. One advantage of developing algorithmic

building blocks and optimizing automata is to alleviate the need for future users to invest in this difficult exercise. Care should be taken in creating automata designs that exhibit as much generality as possible.

7.1.2.1 General Guidelines

1. **Architecture-aware Automata**: In principle, the AP can support any ANML-NFA as long as the number of STEs fit within half of the resources of a single AP chip. In practice, however, the routing network is not an all-to-all connection between all the physical STEs on the chip. Instead, the STEs in the AP are arranged hierarchically into rows, blocks, half-cores and cores, with increasingly limited routing capacity between them. This has been discussed in detail in Section 3.6.3 of chapter 3. When mapping STEs and connections in the ANML-NFA to physical STEs and routing lines on the chip, the routing constraints have to be satisfied. This can cause some of the physical STEs to be 'left-out' for lack of ability to support the required connectivity. Therefore, usage of large and highly-connected automata generally causes poor STE utilization. However, if a large automaton consists of small regions of highly interconnected STEs which can fit inside a single row of physical STEs, but sparsely connected to the rest of the automaton, then such an automaton can map efficiently onto the physical hardware. This will become more clear when we discuss our automata designs in the following sections.

2. **Modular Design Using Macros**: Part or whole of an ANML automata can be designed using macros. As discussed earlier, this has dual advantage. Firstly, the macros can be used as modular parts in future automata. Secondly, the labels of the STEs in a macro can be parameterized and the macro can be pre-built to be loaded at run-time with minimum overhead. Compilation of an ANML-macro generates a mapping to the processing elements and routing lines of the nearest multiple of complete rows in the AP chip. At run-time, multiple instances of the macro can

be created with different values for parameters and loaded into the AP. This process only requires the compiler to map instances of the macros to different rows of the AP-chip. This can be computed at a much faster rate. In fact, the time required for this compilation is lower than the time required to load the instances into the AP.

3. **Avoiding Frequent Reprogramming**: Reprogramming an AP board consists of generating new automata, compiling these new automata and loading the result into the AP-board. This can incur significant overhead, and therefore should be used judiciously. Here a distinction should be made between reprogramming the board and relabeling the STEs of already loaded automata. The latter can be done significantly faster, within a few milliseconds.

4. **Avoiding Frequent Output Generation**: As discussed in Section 3.6.2 of Chapter 3, the processing of output in a symbol-cycle is much slower than the input. Therefore, if the output is generated too often, the output-buffer fills up stalling the entire processing pipeline. The design of automata and algorithms should take cognizance of this aspect.

7.2 Finding Acyclic Paths in Graphs

In this section, we present automata and algorithms we designed to find a set of acyclic paths in a graph that satisfy a specific criterion (e.g., all paths of length k, all paths of length up to k, all paths containing a specific node, etc.). Our algorithmic strategy relies on the string representation of the graph described above. These automata and algorithms are leveraged to solve the problem of finding Hamiltonian paths and cycles in a graph. Subsequently, we illustrate optimization techniques for extracting maximum performance from the AP.

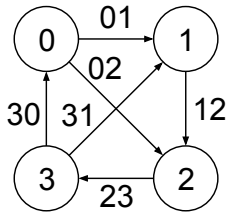
7.2.1 Overview Of The Algorithm

We find acyclic paths in a graph by programming the AP to contain *edge automata*, and streaming partially constructed paths to extend their length. Partially constructed paths are maintained on the CPU and the extension of these paths are calculated with the aid of the AP. The length of a path is defined as the number of nodes contained in the path. Let $u \in V$ be the source node from which acyclic paths have to be found. We consider all outgoing edges of u as partially constructed paths of length 2 for the first iteration. Thereafter, in iteration i, paths of length $i+1$ are streamed through the AP programmed with edge automata to create paths of length $i+2$ for the next iteration.

A path is extended by finding an edge whose source node is the last node on the path, and whose destination node does not occur anywhere in the path. The latter requirement enforces the acyclic property. The edge automaton is designed to reach an accept state if a streamed path satisfies these properties with respect to the edge represented by the automaton. If a path cannot be extended in an iteration, it is dropped from consideration in future iterations. The paths obtained after running the algorithm for $n-2$ iterations are declared as Hamiltonian paths which originate from the source node. Figure 38b illustrates our strategy for finding Hamiltonian paths starting from node 0 of the graph shown in Figure 37.

7.2.2 Basic Path Extension Edge Automaton

We use $\lceil \log_{256} 2n \rceil$ length strings to represent a graph with n nodes. We describe a string using space separated characters, such as 212 005 137 079 145. We reserve strings starting with the character 255 for representing special strings called *delimiters*, and hence they are not available for representing graph nodes. Throughout the paper, we demonstrate automata designs by considering graphs whose nodes are represented using strings of length 4, or equivalently, graphs with $2^{23} < n \leq 2^{31}$ nodes. Note this represents a range of approximately 8 million to 2 billion nodes which encompasses most large-scale graphs. Let (u,v)



(a)

	Iter. 1 ⇒ In: 01, 02; Out: 012, 023					
Edge automata	01	12	23	30	02	31
Matched path		>01	>02			
	Iter. 2 ⇒ In: 012, 023; Out: 0123, 0231					
Edge automata	01	12	23	30	02	31
Matched path			>012	>023(X)		>023

(b)

Figure 38: Iterative search for Hamiltonian paths starting at node 0. In iteration 2, the path with string label 023 is marked with an X because extending it with edge with string label 30 generates a cycle, hence is not reported in output.

denote an edge, where the string labels of u and v are 121 144 169 196 and 251 233 227 211, respectively. The edge automaton corresponding to the edge (u, v) is shown in Figure 39.

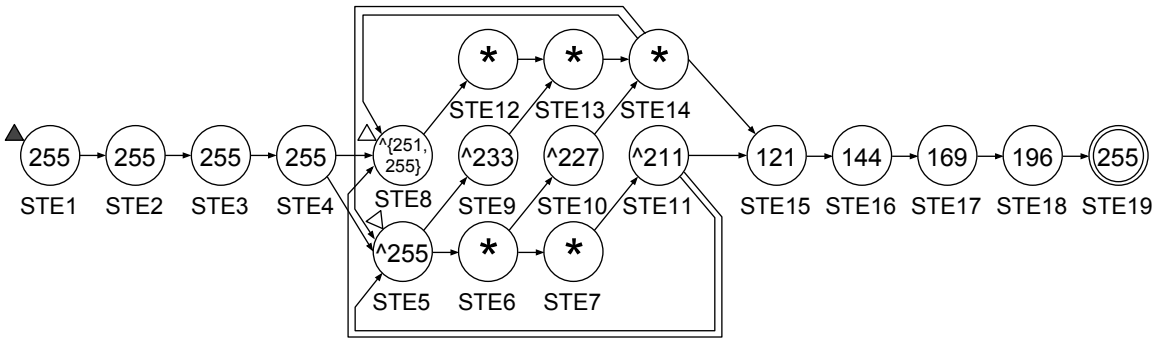


Figure 39: Basic path extension edge automaton for edge (121 144 169 196 , 251 233 227 211).

The automaton works as follows. The *end-of-path* delimiter, 255 255 255 255 is used to mark the end of a path. $STE5$ and $STE8$ are active at the beginning of every path in the data flow. They are active at the beginning of the first path by virtue of being start-of-data STEs (indicated in the diagram by arrowhead pointing to the STE). They are made active at the beginning of the rest of the paths in the data flow by $STE4$. $STE1$ through $STE4$

attempt to match the end-of-path delimiter at every offset in the data flow. This is realized by making $STE1$ as an all-input-start STE (indicated in the diagram by a filled arrowhead pointing to the STE) which is active on every symbol of the data flow. Note that the end-of-path delimiter cannot be confused with any four-byte sequence in the data flow containing the suffix of one string label and the prefix of another.

Within a path, $STE5$ through $STE14$ are used to verify that the string label of v is not matched by the string label of any node in the path being streamed. This is done by ensuring that the string label of every node in the path differs from the string label of v in at least one character. $STE8$, $STE9$, $STE10$ and $STE11$ check for difference in the first, second, third and fourth characters, respectively. If $STE11$ or $STE14$ are matched at the fourth character, then $STE5$ and $STE8$ are activated to check the string label of the next node in the path. $STE11$ and $STE14$ also activate $STE15$ to allow $STE15$ through $STE18$ to check for the string label of node u in the path. If none of the string labels of nodes in the path matches the string label of v, and the the string label of u appears at the end of the path, i.e., it is followed by the end-of-path delimiter, then a match report is generated. Here, $STE19$ is used to check the first symbol of the end-of-path delimiter only, because it is different from the first symbol of the string label of every node in the graph.

7.2.3 Path Extension Edge Automata

7.2.3.1 Basic Edge Macro

The edge automata for different edges differ only in the labels of $STE8$ through $STE11$, and $STE15$ through $STE18$. This observation allows us to define and compile the edge automaton as a macro shown in Figure 40. Recall that macros can be compiled offline to yield a significant reduction in overhead. An instance of the macro is created for every edge $(u,v) \in E$. Parameters $p1$ through $p4$ are used to encode the string label of node v, and $p5$ through $p8$ to encode the string label of node u. For our example edge, the parameter values are: $p1 = \hat{}\{251,255\}$, $p2 = \hat{}233$, $p3 = \hat{}227$, $p4 = \hat{}211$, $p5 = 121$, $p6 = 144$, $p7 = 169$, and $p8 = 196$.

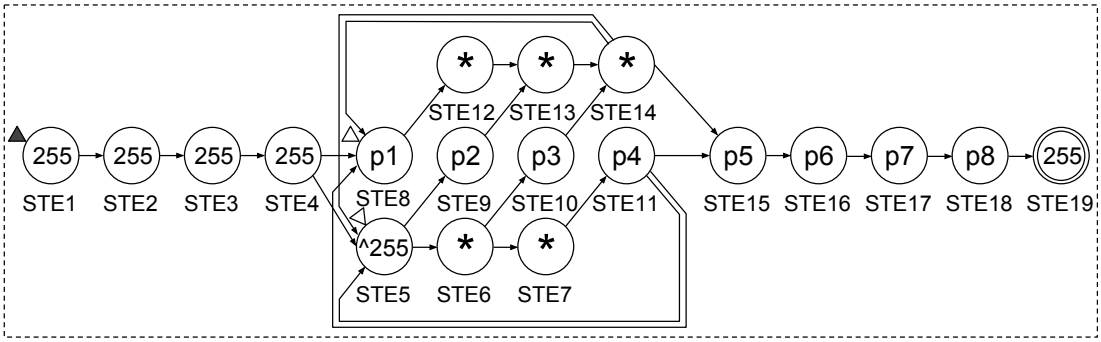


Figure 40: Basic edge macro for extending acyclic paths.

7.2.3.2 Multiple Edge Macro

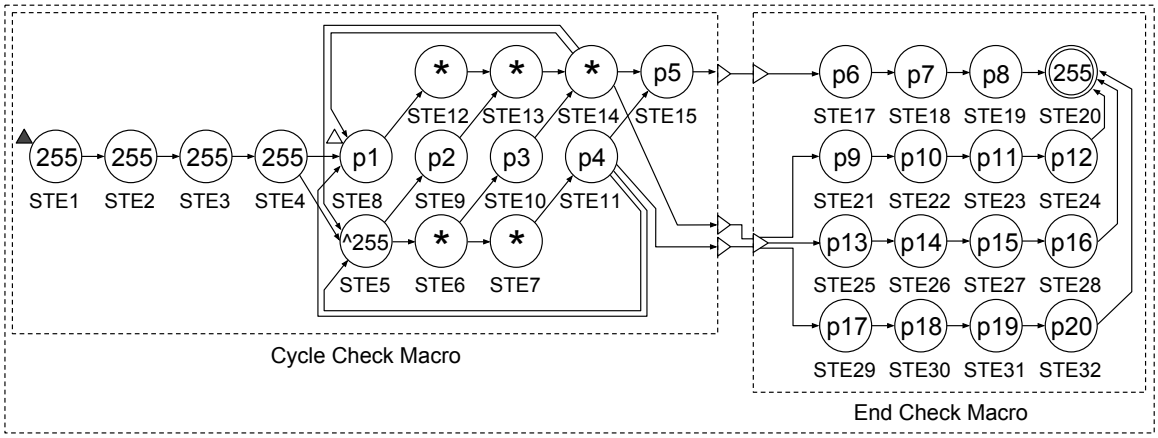
The basic edge macro shown in Figure 40 and described above is not efficient in terms of resource utilization. Recall that each macro is compiled to the nearest multiple of a physical row in the AP chip. The basic macro has 19 STEs, which when compiled is mapped to two rows containing 16 STEs each. This is only about a 59% utilization. The resource utilization can be enhanced by combining up to four incoming edges incident to the same node v into a single macro as shown in Figure 41a. The macro saves the total number of STEs needed by using a single STE path corresponding to node v for all four edges, and in addition brings the total number of STEs much closer to a multiple of 16 to result in 97% resource utilization. The incoming edges into a node v are grouped into macros of size four, with the last group possibly containing fewer than 4 edges. To use the same topology even for the last group, the string label 255 255 255 254 is used as the source node to add additional non-existent edges to make the total a multiple of four.

The optimizations we incorporated into the design of the multiple edge macro are described below.

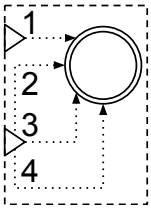
1. **Avoiding routing bottleneck**: The resource utilization in the AP is constrained by the available physical routing lines. As discussed earlier, STEs within a row can be directly connected to each other. However, connections between elements from different rows use up *block routing* lines, of which there are only 24 per 16 rows

in a block, i.e., on average three lines per two rows. In our multiple edge macro shown in Figure 41a, both the constituent macros, namely the *cycle check* macro and the *end match* macro fit inside a single row each. Under this condition, the number of block routing lines required by the complete macro is obtained by counting the number of outgoing ports of the constituent macros. This number for the multiple edge macro is three. Therefore, the routing lines do not form a bottleneck towards resource utilization.

2. **Minimizing redundancy**: Notice that a single reporting STE is shared by the four edges programmed in a multiple edge macro. Therefore, this STE cannot identify the specific edge for which the macro reports an output. However, recall that every edge in the macro has a distinct source node, and for an edge to extend a path, its source node must be identical to the end node of the path. Therefore, this node can be used



(a) Path extension multiple edge macro to handle up to four edges.



(b) Shorthand notation for *end check* macro.

Figure 41: Path extension multiple edge macro.

to uniquely identify the edge.

3. **Optimizing output generation**: Generating output at the end of each path allows the interleaving of operations on the CPU and the co-processor. However, this scheme may result in frequent output generation events when the length of the paths is small. Therefore, we propose to calculate partial paths of length up to 64 (256 symbols) on the CPU and continue the extension of these paths on the AP.

4. **Minimizing run-time overhead**: Even before the input graph is provided, multiple instances of the macro which fill up the entire AP board with dummy values for parameters can be compiled and kept ready for loading. This eliminates the need for the compiler to even place macros at run-time. Instead, only new labels need to be generated based on the edges in the input graph and loaded into the AP. As described in Section 7.1, this operation can be completed in a matter of a few milliseconds.

7.2.4 Applications

The algorithm and automata developed for finding acyclic paths, described in the preceding subsections, can be used to solve multiple graph problems of significance.

Finding Hamiltonian Paths From A Single Node The solution for finding Hamiltonian paths originating from a single node follows directly. We begin with outgoing edges from the node as the initial set of paths. If a path cannot be extended by any edge in an iteration, then the path is dropped from consideration for future iterations. If none of the paths can be extended in any given iteration, then no Hamiltonian path exists starting at node u. All paths remaining at the end of $n-2$ iterations are Hamiltonian paths starting at u.

Finding All Hamiltonian Cycles All Hamiltonian cycles in G must pass through node u. Therefore, our method to find all Hamiltonian paths originating from node u can be easily extended to find all Hamiltonian cycles in G as follows. Let H be a Hamiltonian path in G

originating in node u and terminating in node v. Then, H is a part of a Hamiltonian cycle if and only if the edge $(v,u) \in E$.

Finding All Hamiltonian Paths Let G' be the graph obtained by adding an extra node u' to G, and connecting it to all the nodes in G, i.e., let $G' = (V', E')$, where $V' = V \bigcup \{u'\}$ and $E' = E \bigcup \{(u', u) : \forall u \in V\}$. Find all Hamiltonian paths in G' that originate at u' as per the algorithm described earlier. Removal of the first edge from the paths found will result in all Hamiltonian paths for G.

The number of iterations necessary to solve the Hamiltonian path problems can be cut down in half by searching for edge extensions at both ends of the paths. This requires incorporating additional considerations into the design of the algorithms and automata. We do not discuss the associated details in this paper owing to space constraints.

7.2.5 Heuristics For Pruning Search Space

Finding Hamiltonian paths and cycles are known to be NP-complete problems. Various heuristics have been proposed over the years to prune the search space, while the worst-case remains exponential. Since our algorithms are also based on storing and extending paths, they can leverage the existing wealth of knowledge related to heuristically pruning the search space. For example, all the edges in the graph need not be programmed and some of the extensions can be trivially calculated on the CPU itself. This is because some edges must be present in all and others cannot be present in any Hamiltonian path/cycle in a graph [63]. Extension/dropping of paths when they reach these edges can be done on the CPU itself.

7.2.6 Job Partitioning

Even with the most optimized design, the automata corresponding to all the edges in a large graph cannot fit inside an AP board at a given time. We define the *edge-packing ratio* as the number of edges that can be programmed into a single row of the AP chip. In the automata

discussed so far, the best edge-packing ratio is obtained for the multiple edge macro where four edges can be programmed into two rows, giving rise to an edge-packing ratio of 2:1. This allows us to fit $6,144$ edges in the $3,072$ rows available in an AP chip. In the largest AP board containing 48 AP chips, the number of edges that can be programmed at any given time is $294,912$. For graphs with a larger edge-count, we need to partition the edges and handle them in rounds.

This can be used for better job partitioning by matching source nodes to edges to end nodes of paths. Automata for the edges with the same source nodes are programmed onto the same AP chip(s) and in the same round(s). Similarly, paths are organized on the basis of their end nodes and streamed only in the round(s) and to the chip(s) where the edge automata containing the same nodes as source are programmed.

7.2.7 Estimating Speed-up

Using the AP, investigation of all possible extensions of a partial path ending in node u can be performed in parallel. This involves checking all outgoing edges of u. Therefore, the speedup due to this approach is the average outdegree of a node in the graph. Without considering self-loops (edges from nodes to themselves), this ratio is $\frac{|E|}{|V|}$. If the AP board has p AP chips, and the ratio of clock-speed of the CPU to that of the AP is f, then speed up can be estimated to be in the order of $\frac{|E|}{|V|} \times \frac{p}{f}$.

Note that we are not able to report actual run-times as the AP is not yet available. We have extensively used the available Software Development Kit (SDK) to verify the correct functioning of all the automata described here. However, the SDK cannot simulate problems at the scale that can be handled by the hardware, hence simulated run-times cannot be reported.

7.2.8 Additional Improvements

Extended Multiple Edge Macro An adaptation of the multiple edge macro, called the *extended multiple edge* macro is shown in Figure 42. The use of this macro leads to a higher

edge-packing ratio when the indegree of a node is greater than four. The macro contains a single instance of the cycle check sub-macro to represent the common destination node, i.e. v. The number of instances of the end check sub-macro is dependent on the indegree of v. The macro shown in Figure 42 can handle 11 incoming edges, where the first two instances of the end check sub-macro handle four edges each, and the third handles three edges. The edge packing ratio of this macro is 11 : 4. This macro can be extended to use all the 16 rows in a block to handle 47 edges incident on v, giving rise to an edge packing ratio of 47 : 16, i.e. nearly 3 : 1. Note that this is higher than the edge-packing ratio of 2 : 1 obtained by making several copies of the multiple edge macro.

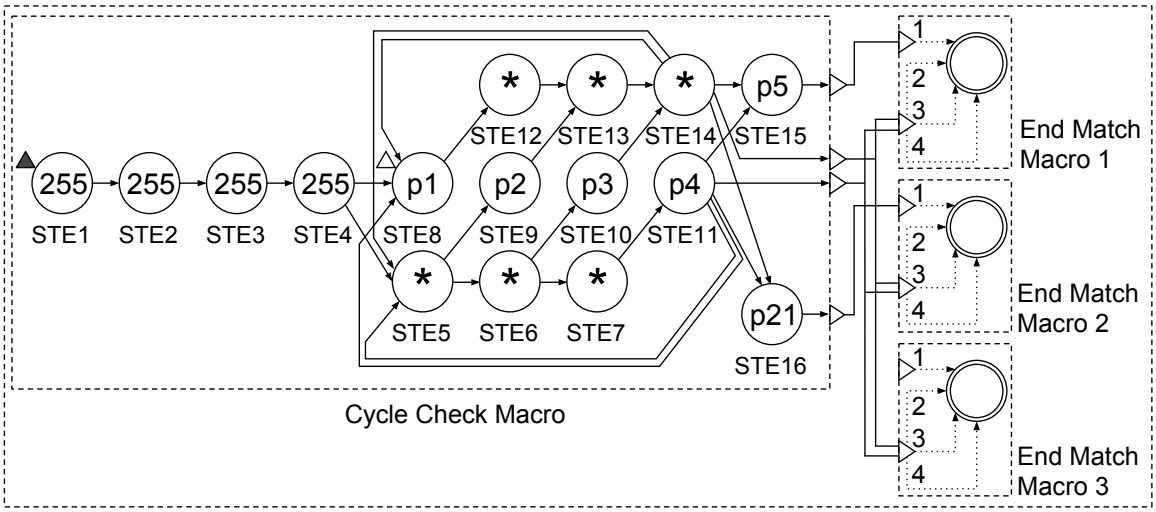


Figure 42: Extended path extension multiple edge macro to handle up to 11 edges.

The resource utilization of such a design is also high. For example, the macro shown in Figure 42 utilizes 61 out of 64 STEs available in four rows. This results in utilization of over 95%. The average number of block routing lines required per row is 1. Therefore, the routing lines do not create a bottleneck.

However, the macros for all edges incident to a node are no longer identical. They are now dependent on the indegree of the node. Hence, these macros have to be computed and compiled at runtime on the basis of the input graph. Although such a compilation will involve pre-compiled macros only (not individual STEs), it will still incur some overhead.

On the other hand, by packing 1.5 times more edges per round, the algorithm will require fewer rounds, thus incurring lower associated latency and overhead. The trade-off between the two needs to be carefully evaluated through experiments. The option of pre-compiling entire blocks with different combinations of macros, which can handle different number of edges, can also be considered.

7.2.8.1 Path Extension Path Automata

Extending partial paths using edge automata suffers from the limitation that the length of the paths grows by one per iteration. Therefore, finding Hamiltonian paths and cycles requires $O(n)$ iterations. A potential improvement to this approach could be the design of path automata which identify paths that can be concatenated to one another. Such a solution can bring down the number of required iterations to $O(\log n)$. However, any implementation using path automata needs to overcome the following challenges. First, the size of a path automaton will grow with the length of the path. However, the hierarchical routing network of the AP places restrictions on the size of any programmable automaton. Second, the AP board has to be reprogrammed with new automata for every iteration, which incurs additional cost. However, short paths can be programmed and may be effectively used to reduce the overall run time. This remains an open avenue for investigation.

7.3 Building Shortest Path Trees in Graphs

In this section, we provide algorithms for finding single-source shortest paths and connected components in a graph. These algorithms revolve around finding a *shortest path* tree for a node $u_1 \in V$, denoted as $T(u_1)$. The shortest path tree of u_1 is defined as a spanning tree rooted at node u_1, containing the shortest path between u_1 and v, $\forall v \in \{V - u_1\}$ where a path exists between u_1 and v in G. In other words, this tree captures the *breadth-first search* traversal in G originating at node u_1.

7.3.1 Overview Of The Algorithm

The algorithm for finding shortest path trees works iteratively, similar to the one for finding acyclic paths. The tree is maintained on the CPU and is initialized with the first two levels in the tree. Node u_1 is placed at level 1 as the root of the tree. Any node connected to u_1 by an outgoing edge is placed at level 2 along with the accompanying edge.

In iteration i, the nodes at level $i+2$ of the tree are calculated. Two lists are readied in preparation for an iteration. The first list, called *visited list* is represented as $L1_i$ and consists of the nodes in the tree from levels 1 to $i+1$. The second list, namely *last-visited list* is represented as $L2_i$ and contains the nodes at level $i+1$ only. The data flow contains the string labels of the nodes in the visited list, followed by the string labels of the nodes in the last-visited list. The order of the nodes in both the lists is inconsequential, and the end of each list is demarcated by a special *end-of-list* delimiter. The end-of-list delimiter is chosen as 255 255 255 253, which is easily distinguishable from the string labels of the nodes in the lists through its first symbol (255).

An edge automaton is programmed into the AP for each edge $(u,v) \in E$, where $u \neq v$ and $u,v \neq u_1$. The automaton generates a report if the source node is in the last visited list of nodes and the destination node is not in the visited list node, i.e. $u \in L2_i$ and $v \notin L1_i$.

The last-visited list for the next generation, $L2_{i+1}$ contains all nodes visited in the current iteration, i.e., node $v \in L2_{i+1}$ if the edge automaton for (u,v) generated an output in iteration i. If $L2_{i+1}$ is empty, then the processing stops because all nodes in G which are reachable from u_1 have been captured in the tree. Otherwise, the visited list for the next iteration, $L1_{i+1}$ is calculated by combining all the nodes visited in the i^{th} iteration with all the nodes visited in the previous iterations, i.e. $L1_{i+1} = L1_i \bigcup L2_{i+1}$. With $L1_{i+1}$ and $L2_{i+1}$ ready, the streaming for the next iteration starts.

Meanwhile, the CPU updates the tree with the nodes at level $i+2$. If there are more than one shortest path between node u_1 and node v, then edge automata for multiple edges ending in v generate an output in iteration i. For the shortest path tree, we need to retain

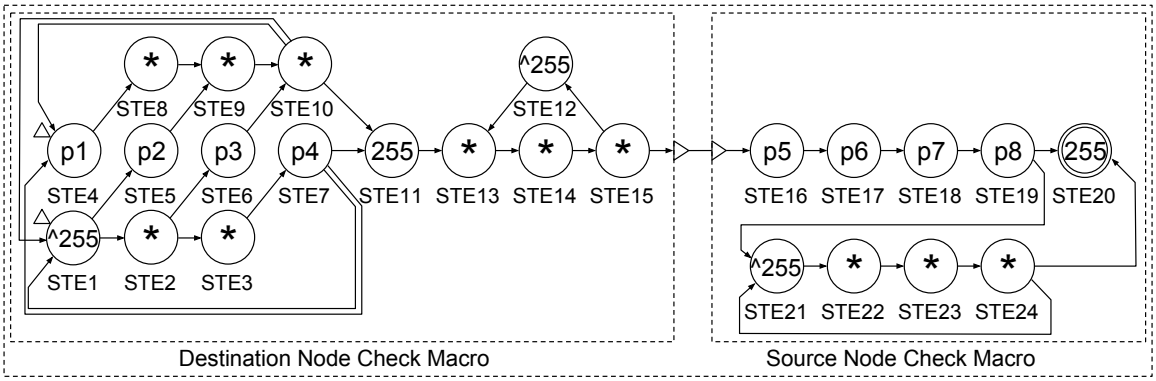


Figure 43: Basic tree extension edge macro to check if an edge can be attached at the next level of the tree.

only one of these edges. We choose the edge encountered last by the host-application during the iteration. This allows us to overwrite the record for node v without performing any conditional checks.

7.3.2 Basic Edge Macro

All automata described in this section are also designed as macros, so that they can be replicated and loaded quickly into the AP. Figure 43 shows the *basic edge* macro for any edge $(u,v) \in E$. If this macro generates an output during iteration i, then v is added to level $i+2$ of the shortest path tree. Parameters $p1$ to $p4$, and $p5$ to $p6$ are set according to the string labels of v and u respectively. For example, given the edge (121 144 169 196 , 251 233 227 211), the values of $p1 =\hat{}\{251,255\}$, $p2 =\hat{}233$, $p3 =\hat{}227$, $p4 =\hat{}211$, $p5 = 121$, $p6 = 144$, $p7 = 169$, and $p8 = 196$.

The macro is made up of two constituent macros namely *source node check* macro and *destination node check* macro. These two constituent macros map to two rows of STEs on the AP chip and use a single block routing line. This design results in high resource utilization and does not suffer from any routing bottlenecks.

$STE1$ through $STE10$ check for the absence of node v in the visited list. Its working is very similar to that of the *cycle check* macro described in Section 7.2.3.2. The checking of

the next string label continues only if the current string label is non-identical to the string label of v. This is ascertained by making sure that at least one of the symbols in both the string labels mismatch. Only if this condition is true for the string labels of all the nodes in the list, $STE11$ is activated to check the first symbol of the end-of-list delimiter streamed at the end of the list.

$STE16$ through $STE24$ check the presence of node u in the last-visited list. If node v is absent in the visited list, then $STE15$ activates $STE16$ to check the first symbol of the string label of every node in the last-visited list. It accomplishes this by cycling through activation of the four nodes $STE12$ through $STE15$ after matching the end-of-list delimiter. If the string label of any of the nodes in the last-visited list matches the string label of u, then $STE19$ activates nodes $STE20$ and $STE21$. If u is the last node in the list, it generates an output immediately after parsing the first symbol of the end-of-list delimiter which immediately follows. Otherwise, $STE21$ to $STE24$ cyclically activate each other to make sure that $STE20$ is active to match the first symbol of the end-of-list delimiter whenever it is encountered.

As the size of the tree grows, the number of visited nodes becomes larger than the number of unvisited nodes. Therefore, it is efficient to maintain and stream the latter when this occurs. This new list is called the *unvisited list* and denoted by L'_i.The switch from streaming the visited list to the unvisited list can be performed when $|L1_i| \geq |V|/2$. At this point, the unvisited list can be calculated as $L'_i = V - L1_i$. For all subsequent iterations, the last-visited list continues to be updated as before; while L'_{i+1} is calculated as $L'_{i+1} = L'_i - L2_{i+1}$. All other operations on the CPU remain the same.

An automaton to work on the modified data flow containing the string labels of nodes in the unvisited list is called the *alternate basic edge* automaton and shown in Figure 44. For an edge (u,v), it creates an output when $v \in L'_i$ and $u \in L2_i$. The automaton consists of two sub-macros, and maps to two rows in the AP chip.

The *alternate destination node check* macro checks for the string label of v in the string

labels of nodes in the unvisited list. For our example edge, parameters $p1$ to $p4$ are now set to values $p1 = 251$, $p2 = 233$, $p3 = 227$, and $p4 = 211$. If v is not the first node in the unvisited list, then $STE1$ to $STE4$ ensure that $STE5$ is activated to check the first symbol of the string label of every node in the list. If node v is found in the unvisited list, then $STE13$ is activated to start matching the end-of-list delimiter. However, if v is not the last node in the list, $STE9$ to $STE12$ are used to cyclically activate $STE13$ to check the first symbol of the end-of-list delimiter, whenever it is streamed.

The *alternate source node check macro* contains $STE12$ to $STE24$ of the basic edge macro rechristened as $STE14$ to $STE26$. The redistribution of the STEs across the sub-macros is carried out so that each sub-macro fits inside a single row in the AP chip, and uses as few block routing lines as possible.

7.3.2.1 Multiple Edge Macro

The edge-packing ratio of the basic edge macro is 1 : 2. We can increase this ratio by using the *multiple edge* macro shown in Figure 45a. The source node check sub-macro is replaced with a *multiple source nodes check macro* which allows the automaton to handle three edges ending in the same node v. This automaton can be programmed into three rows, giving rise to a edge-packing ratio of 1 : 1. Similar to the multiple edge macro designed for the acyclic path problem, if a node has less than three incoming edges left to be programmed, then a string label of 255 255 255 254 can be used for remaining source nodes in the automaton.

The alternate basic edge macro to check for destination nodes in the unvisited list can also be modified in a similar manner. Instead of replacing the alternate source node check macro, a *multiple source nodes check macro* is added to the automaton as shown in Figure 46. In this case, four edges can be mapped to four rows, again giving rise to an edge-packing ratio of 1 : 1.

7.3.3 Applications

The algorithm described above can be used directly or adapted for solving various applications as described below.

7.3.3.1 Single-source Shortest Paths

The shortest path tree enumerates the shortest path from the node u_1 to every node in G that is reachable from u_1. Hence, it acts as the solution to the single-source shortest path problem for unweighted graphs. This problem lies at the heart of the popular *Graph500* [2] benchmark used to evaluate the performance of high performance computers.

If the actual paths between the nodes are not desired, but only the shortest distance between them are, then the tree need not be stored. If node v is added during initialization, then its distance from u_1 is 1. Otherwise, if v is *visited* (added to the visited list or removed from the unvisited list) in iteration i, then its distance from u_1 is $i+1$.

7.3.3.2 Connected Components

A connected component of an undirected graph is defined as the subgraph in which any two nodes are connected to each other through a path, and any node lying outside the subgraph is not connected to any node within the subgraph. The connected component of a node is defined as the connected component of the graph to which it belongs. In our algorithm, the connected component of node u_1 is the visited list (or the complement of the unvisited list) generated at the end of the last iteration. Note that, for calculating the connected component of u_1 we do not need to store the shortest path tree of u_1.

The search for all connected components in a graph can be carried out in two ways. In the first technique, the components are discovered serially. A node $u_1 \in V$ is randomly selected and its connected component is identified. The nodes in this component are marked as *covered*. Then, another node in V is selected which is not yet covered, and its connected component is identified. The process is continued until all the nodes in G are covered.

Alternatively, the connected components in a graph can be identified by building a forest of trees in parallel. The trees are initialized with two levels each, such that every node in the graph is contained in only one tree and all the nodes in the graph are covered. The trees are ordered from left to right, and the data flow comprises of the visited and last-visited lists of the trees arranged in the same order. The automaton for an edge (u, v) is modified to report an output if v is not present in the visited lists of the current tree and all the trees to its left; and u is present in the last-visited list of the current tree. The modification corresponding to the macro shown in Figure 45a is shown in Figure 47.

The destination node check macro continues to check for the string label of v against the string labels of nodes in the visited list of trees in the data flow until a match is found. The logic for termination of the check already exists in the previous macro. However, the logic for continuation of search for the next tree is added through the inclusion of $STE43$. If v is not encountered in the visited list of the current tree, $STE43$ is periodically activated by $STE15$ at an interval of four symbols to check for the first symbol of the end-of-list delimiter. Only on encountering the delimiter at the end of the last visited list of the current tree, $STE43$ activates $STE8$. $STE8$ to $STE10$ match the rest of the delimiter and activate $STE1$ and $STE4$ at the start of streaming of the visited list of the next tree.

A tree stops growing under two circumstances. Either the tree already contains all the nodes in a connected component, or all the candidate edges that can be appended to the tree end in nodes which appear in a tree to the left. If a tree stops growing, the algorithm retains the tree, but stops the inclusion of the visited and last-visited lists of the tree in the data flow for subsequent iterations. The algorithm terminates when all the trees stop growing.

After the final iteration, every connected component in the graph is represented by at least one tree. Among these trees, only the leftmost tree contains all the nodes in the component, and should be retained. The identification of such trees can be performed as follows. Every node in the graph is initially marked as uncovered. Starting from the leftmost tree, if the root node is marked as uncovered, then the tree is retained and all the

nodes in the tree are marked as covered. Otherwise, the tree is dropped, and the process continues with the next tree to the right. At the end of this pruning, only one tree per connected component remains and its visited nodes are declared as a connected component.

The above algorithm is illustrated in Figure 48. In iteration 1, edge $(13, 7)$ is not reported for the third tree as node 7 appears in the tree to the left. Although the third tree is retained, streaming of its visited and last-visited list is stopped from iteration 2 onwards. This is depicted by a horizontal line underneath the tree signifying that the tree has stopped growing. In iteration 3, all the trees stop growing, and hence the iterations stop. All trees with root nodes which do not appear in a tree to the left are declared as representative of connected components in the graph, and their visited lists are declared as an individual connected component.

The above algorithm can be enhanced by combining trees for which edge(s) with a common end node is/are reported in the same iteration. Note that, although we have conceptually described this algorithm using trees for better clarity, only the visited lists and the last visited lists of the trees need to be maintained. Therefore, combining trees merely comprises of set-operations on their lists. The visited list and the last-visited list of the combined tree are the union of the visited lists and the last visited lists of the original trees respectively. Note that the last-visited list of the combined tree may contain nodes which lie in the interior of the tree. However, this does not influence the outcome of the following iterations.

7.3.4 Future directions

Remembering states between iterations Currently, we do not remember the states from one iteration to the next. This is because the overhead of remembering the state of an AP chip is currently very high. For our current algorithm, this means incurring the overhead to redundantly stream all the visited nodes in every iteration. However, in the near future, the overhead for remembering the state information may be lowered significantly under

certain conditions. To weigh the cost of incurring this lowered overhead instead of the cost streaming the visited nodes redundantly for each iteration would be an interesting exercise.

Another key area of research is the handling of graphs with edges assigned small integer weights. The weight of these edges can be captured by a proportionally long chain of STEs, i.e. the number of symbols required to traverse the edge. Alternately, a weighted edge can be represented as proportionally long chain of nodes in the shortest path tree. In other words, adding an edge (u,v) with weight w to the tree would mean adding a chain containing $w-1$ *don't-care* nodes, one each from layers $i+1$ to $i+w-1$, followed by node v at layer $i+w$. In this graph, the minimum distance between v and the node at which the tree is rooted is $i+w-1$.

7.4 Summary

Although, the Automata Processor is primarily designed for accelerating complex string pattern matching applications, it can be used to solve problems on unweighted graphs. In this chapter, we have presented some foundational techniques and algorithms for solving a number of graph problems including Hamiltonian paths/cycles, single-source shortest paths, and connected components using the co-processor. We illustrated strategies for improving the efficiency of the automata by tailoring them to the underlying hardware. The proposed techniques broaden the scope of the co-processor and serve as a demonstration of a non-intuitive use of the Automata Processor for solving more general problems.

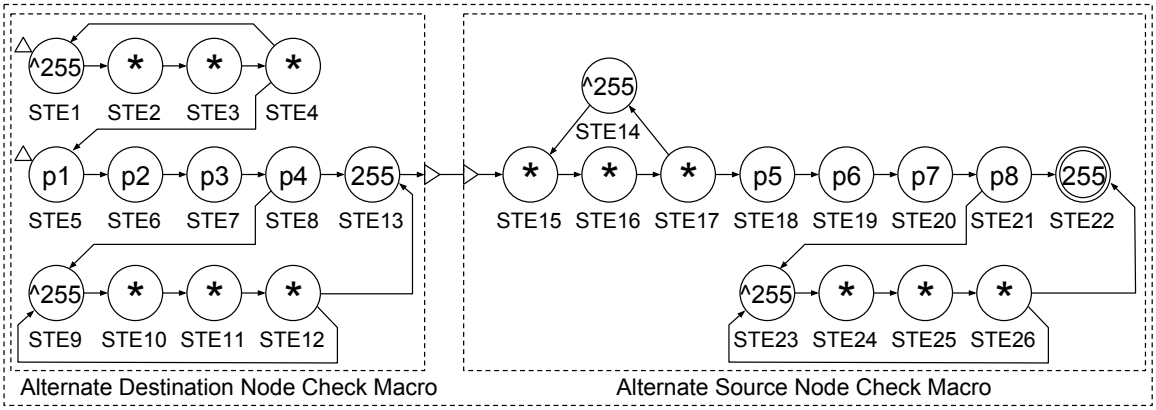
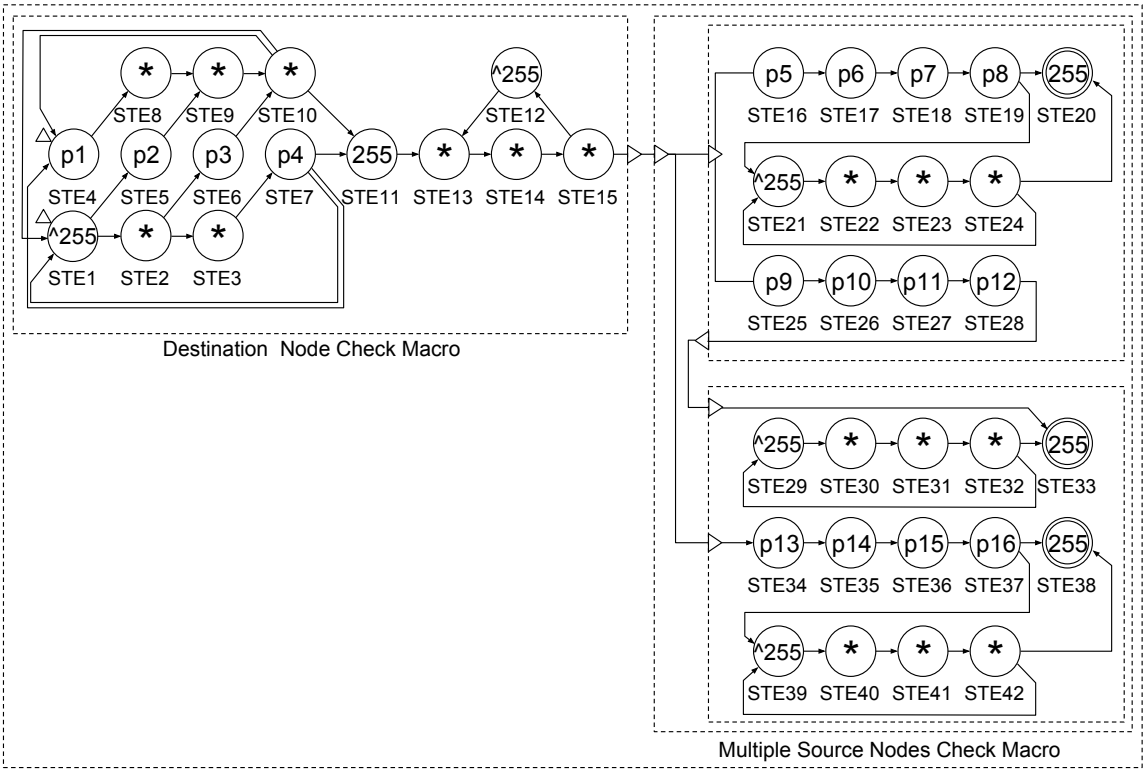
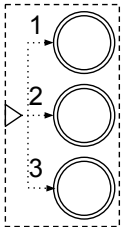


Figure 44: Alternate basic tree extension edge macro to check if an edge can be attached at the next level of the tree.



(a) Tree extension multiple edge macro to check if edges can be attached at the next level of the tree.



(b) Shorthand notation for *multiple source nodes check* macro.

Figure 45: Tree extension multiple edge macro

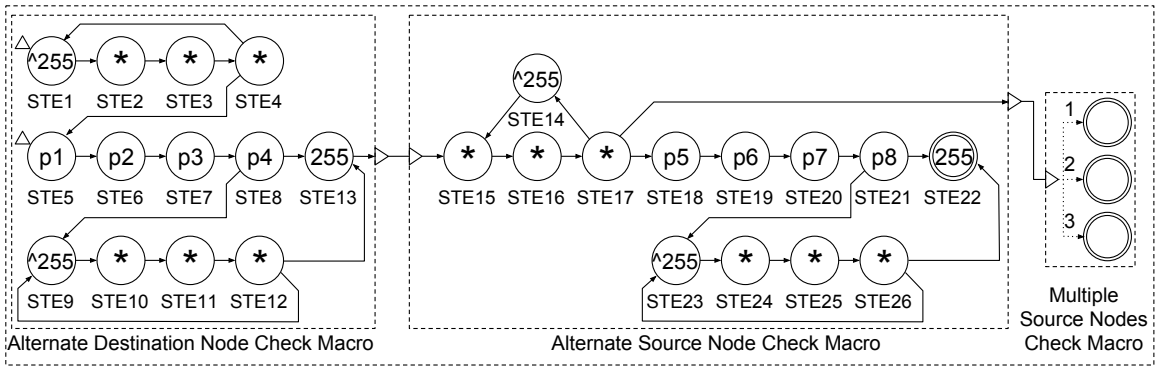


Figure 46: Alternate tree extension multiple edge macro to check if edges can be attached at the next level of the tree.

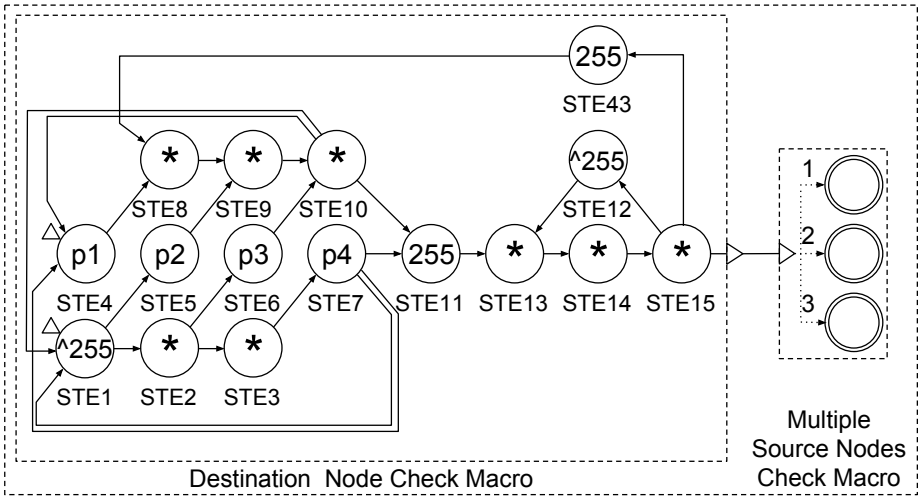
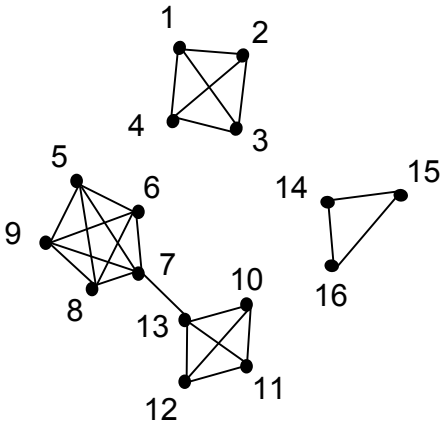
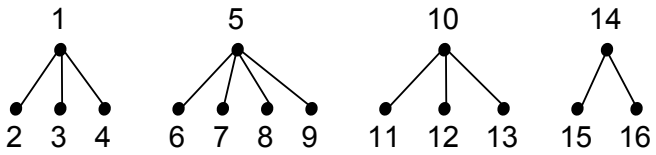


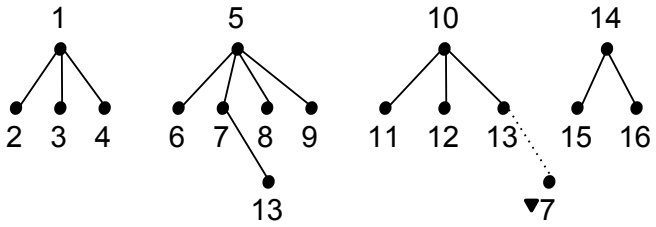
Figure 47: Modified tree extension multiple edge macro to find all connected components in a graph.



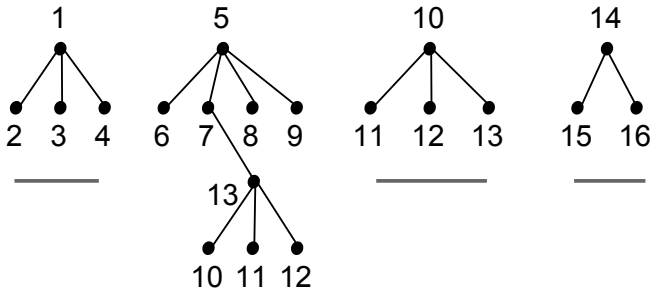
(a) Input graph



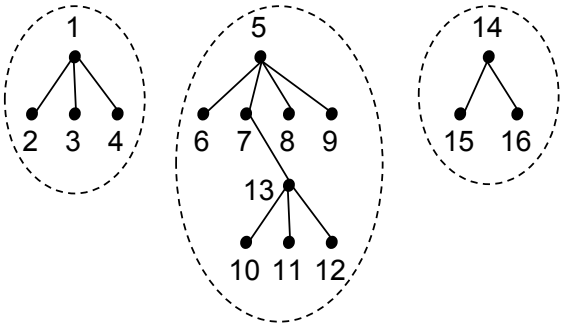
(b) Initialization.



(c) Iteration 1.



(d) Iteration 2.



(e) Iteration 3: All connected components declared.

Figure 48: Finding all connected components in a graph.

Chapter VIII

CONCLUSION

In early 2012, we set out on an interesting journey to use a completely new kind of processor to accelerate various applications in the field of bioinformatics and others. At that time, the capabilities, shortcomings and potential of the processor was unknown. None-the-less, the success of the processor is tied to either its excellence in solving certain commercially important problems, or its applicability to a wide variety of problems. Towards that end, we have established that this processor excels in its designed goals, i.e. accelerate applications which need to search for large number of string patterns in parallel. We have also come up with some fundamental techniques to solve a variety of classic problems in computer science. It is hoped that this work leads to the development of similar algorithms and applications using the processor in future.

8.1 Challenges

In the process of developing these algorithms and applications, we overcame some challenges in using the processor, which may be pertinent to other developers. We list some of them here. The chip was initially designed to perform high-speed string pattern-matching operations in parallel to accelerate the search for *signatures* of intrusion in network data. The signatures occur relatively infrequently in the input data. Therefore, the hardware data pipeline was designed keeping in mind that the output generation frequency is low.

However, this is not the case with many applications, especially those in the field of bioinformatics. We learned this with our attempt to provide an accelerated solution to the *sequence mapping* problem, also known as the *re-sequencing* problem. This problem deals with generating the genomic sequence of an individual belonging to a species whose *reference* genome is already known. The input to this problem is numerous short-length

reads of the individual's genome generated by a modern sequencing machines. However, in the process of generating these reads, their location within the individual's genome is lost. Since an individual's genome is very similar to the reference genome, a potential location is generated by mapping the read along the reference genome based on maximum local similarity. Generally, the first 30-40 characters of the read can be used to place the read at a unique position along the genome. Even if an automaton can be designed to identify this location, an output is generated on every 30^{th} symbol. This is too frequent for the output handling system on the chip. From our studies, we surmised, that in actual practise, the automata that can be fit inside a single AP board could be used to map only 10 symbols from the reads for a human-sized genome. This aggravated the output generation rate even further.

Besides this, our initial automata designs did not map efficiently into the physical hardware. Feedback from these initial designs are not only influencing subsequent automata designs but also the design of the next generation of the chip itself. With these changes, the next generation of the chip is expected to handle a much wider variety of automata than what is possible with the current generation.

The choice of processing elements in the design of an automaton can play a critical role in the efficiency of the solution. For example, for the ease of implementation, we had initially decided to program all the signatures of intrusion in the FAST-Snap application as PCRE. For signatures which required multiple patterns to be matched within a data packet or packet-segment, the PCRE was designed using *lookahead assertions*. However, the AP compiler does not support more than three lookahead assertions per expression. Even for the expressions for which the compilation is possible, the generated automata consumed far too much on-board resources than what was originally expected. This was not a compiler shortcoming. The equivalent ANML-NFA of such an PCRE expression is significantly large.

However, when these signatures are defined as ANML-NFA with a boolean *AND* element as shown in Figure16 of Section 3.4 in Chapter 3, the compiled NFA requires only about one-fourth of the resources. Similar design considerations should be exercised when designing automata which span multiple rows on an AP-chip. If many interconnections exist between elements of one row to elements of another, then the relatively scarce inter-row block routing lines are used up quickly. This may lead to many rows in the block remaining unoccupied. Similarly, with respect to STEs, the counter and boolean elements occur in a ratio of $1 : 64$ and $3 : 64$ respectively. Therefore, over-dependence of automata on the boolean or counter elements may also lead to under-utilization of the resources on the chip.

Another challenge during the initial years of this research was the evolving software. Today, the AP comes with a robust SDK with both command-line and a GUI-based interface (workbench). The workbench allows easy visualization and debugging of the automata. The SDK has matured to provide a clean, well-defined and mostly bug-free interface. When we started our research, these system software were still being designed, developed, debugged and modified.

However, the most significant challenge was to understand how to exploit the new operations and avenues of parallelism provided by the AP technology. In order to do this, it was imperative to cast problems into string-matching problem, for which automata can be designed. For lower execution times, these automata should compile, load and execute efficiently on the AP. For exploiting maximum parallelism, 1) the automata had to be designed such that a large number of automata can be fitted into the AP at anytime, 2) a significant number of automata should be active for every byte being streamed to chip, and 3) the output generation rate should be slow so that the rest of the pipeline is not stalled.

Examples of lowering programming overhead can be seen in MOTOMATA and all graph applications. In these applications, the automata cannot be defined until input is provided at run-time. However, doing so may lead to unacceptable programming overhead. Therefore, all the automata are designed as macros, in which the STEs and the connections

between them do not change from one instance of the macro to the other. At runtime, only the labels of the STEs are generated and the pre-compiled automata can be loaded into the AP almost instantaneously.

8.2 Achievements

The challenges notwithstanding, the processor has many distinct features which provide some unique capabilities. We exploited these capabilities to develop various applications such as Fast-SNAP, PROTOMATA and MOTOMATA, boolean SAT, various graph applications including finding Hamiltonian path and cycles, connected components, single-source shortest paths etc. All the automata designed map efficiently into the actual hardware and consists parts which can be reused in future automata designs which require similar functionality.

Our first application, Fast-SNAP uses the AP in its designed role, i.e. checking for thousands of string patterns in parallel. Using a single AP-board, deep packet inspection of network data against $4,396$ signatures of intrusion can be carried out at up to 11 Gbps. A more fine-tuned solution with better load-balancing based on actual network data-flows may yield even better scanning rates. This is at least an order of magnitude faster than other accelerated solutions.

PROTOMATA adds to this by providing the additional capability of dynamically enabling or disabling a programmed automaton at run-time. It also accumulates output generating events into a single symbol cycle to avoid the output-handling bottlenecks and provides better performance. For large datasets, PROTOMATA is over half a million times faster than its single CPU counterpart.

In MOTOMATA, we inadvertently solved our first graph problem by finding all maximal cliques in an n-partitioned graph. By exploring these cliques in parallel, the runtime of this method is significantly better than existing state-of-the-art solutions. MOTOMATA can solve much larger instances of the problem than what is possible by other competing

software running on large CPU clusters.

Although, the underlying problem to solve the SAT problem is same as that in MOTOMATA, the solution needs modifications because of different input parameters on which the run-time depends on. A new algorithm was developed employing compact clique-automata which can handle small, but computationally challenging instances of the problem, where heuristical pruning methods used by existing SAT solvers are ineffective.

Although, the above two applications solve the problem by using graph algorithms, the original input is not a graph. Rather, different parts of the input are represented as nodes in a graph. The edges in the graph are not provided, but a condition for the existence of an edge between two nodes is provided. The task is to search for a set of nodes which are connected in a specific way, i.e. cliques of a certain size. The primary work of the automata is to search for the existence of all the edges required to form such a clique.

The rest of the graph problems handled in this work are different in the sense that they involve generic graphs, where all the edges are already provided in the input. These problems do not involve edge-discovery at all. Instead, they deal with arranging the nodes and edges in the input graph into output graph structures like search trees and acyclic paths. We have overcome some primary challenges to solve such problems. 1) We developed a strategy to transform parts of any generic graph into strings. 2) We have developed algorithmic and automata design techniques which work with these strings to provide solutions to the problems like finding single-source shortest paths, connected components, Hamiltonian paths and cycles in a graph.

8.3 Scope For Future Research

Although, our first attempt to provide a solution to the re-sequencing problem was not successful, a fresh look at the problem has yielded better automata designs with significantly better resource utilization. At the moment of writing this dissertation, a solution to provide

this much needed solution is being evaluated. If such an application is possible with hardware resources that can be co-located with sequencing machines, then it may expedite the advent of the era of *personalized medicine*.

In this dissertation, we have uncovered some basic concepts on how to program generic graphs using this processor. Designers are already building on these concepts to handle various other graph problems. For example, just like the occurrences of known string patterns in network data signals malicious content, occurrences of sub-graphs of known *shapes* may signal various conditions of interest in other application spaces. These graphs and sub-graphs can be represented using string representation defined in this dissertation, or built upon the same. For applications working on large graphs, this may lead to significant acceleration.

Similarly, in this document unweighted edges are represented by capturing only its end nodes. Research is ongoing on representing edges with small integral weights by using a chain of STEs of proportional length.

The AP board can generate enormous data-rates. Each AP-chip on the board can process data at 1 Gbps and generate data at up to 3 Gbps. Supporting this bandwidth throughout the entire system may be a challenge. This is especially true about the external memory (I/O) transfer rates. Though, this has been handled with reasonable success in high bandwidth network switches, it might become a severe bottleneck in many other applications. Incidentally, this is true with most modern accelerators. As a remedy, some recent research [77] have investigated the use of flash-based external memory coupled with smart middle-ware to provide near in-memory performance even for graph-applications which require random I/O access. Coupled with such remedies, AP applications may provide some interesting results. For example, exploiting data-locality in graph analysis on a distributed memory clusters is difficult and leads to communication bottlenecks between nodes in the cluster. Alternatively, accelerated streaming solutions for such problems may be investigated using systems consisting of smart flash-based I/O, a large main-memory bank and a

CPU working in unison with an AP to provide significant time, cost and energy savings.

The success of any new architecture is based on many factors, technical, financial, user-acceptance and even good fortune to certain extent. Only time will tell whether the Automata Processor will succeed or not. However, it certainly has the potential by virtue of being the first truly large-scale commercially available MISD architecture which is both fast and power efficient. It feels in a void in current processing capability space. Therefore, it possesses some unique capabilities to execute some operations faster and more efficiently than other existing processors. This dissertation is an effort to identify those operations and use them in the development novel techniques and algorithms. Hopefully, this research will be helpful to develop exciting new applications in the future using this new processing technology.

Appendix A

BOUNDED MISMATCH COUNT AUTOMATON

For a sequence $S = s_1 s_2 s_3 ... s_i ... s_l$, the *Bounded Mismatch Identification* automaton discussed in Section 5.2.1 of Chapter 5 accepts any input string of length l that is different from S in at most d positions. However, if the count of mismatches is also required, then the *Bounded Mismatch Count* automaton can be used which provides a mechanism to deduce the count based on the accept state reached. This automaton design consists of $2d+1$ rows of STEs arranged in l columns. Column i corresponds to symbol s_i: the label of STEs in that column recognizes s_i if the row number is odd, and recognizes anything other than s_i (described as $\hat{s}_i$) if the row number is even. The STEs in odd rows are connected in a linear chain to reflect continued matching of symbols. Each mismatch causes skipping of two rows, thus allowing the row of accepting STE to reflect the number of mismatches in the accepted pattern. An STE is present in column i and row j only if $j \leq min(2i, 2d+1)$. The last column of STEs are the accepting states. The design is best illustrated through an example (see Figure 49).

This automaton requires a total of $(2d+1)l - d^2$ STEs. The usage of the all-input start STEs allow the matching to occur anywhere within a longer input sequence. Notice that this automaton requires d^2 STEs more STEs than the Bounded Mismatch Identification automaton.

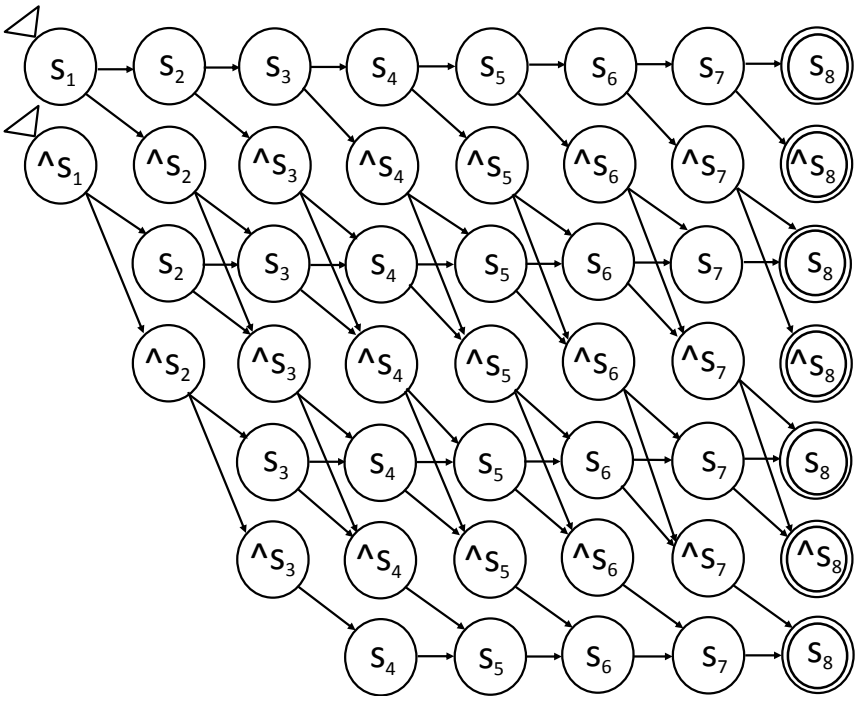


Figure 49: An automaton to accept strings with at most 3 mismatches from $s_1 s_2 \ldots s_8$, and reveal the number of mismatches.

REFERENCES

[1] "Automata Processor Software Delopment Kit: Supported PCRE Syntax." http://www.micronautomata.com/documentation/ap_sdk/h1_known_issues.html. Accessed: Apr, 2015.

[2] "Graph500." http://www.graph500.org/. Accessed: Nov, 2014.

[3] "Perl Compatible Regular Expressions (PCRE) library." ftp.csx.cam.ac.uk/pub/software/programming/pcre. Accessed: Apr, 2015.

[4] "Snort." `https://www.snort.org/`. Accessed: Nov, 2014.

[5] "The Micron Automata Processor Developer Portal." http://www.micronautomata.com/. Accessed: Apr, 2015.

[6] ADLEMAN, L. M., "Molecular computation of solutions to combinatorial problems," *Science*, vol. 266, no. 5187, pp. 1021–1024, 1994.

[7] BAILEY, T. L. and ELKAN, C., "Fitting a mixture model by expectation maximization to discover motifs in bipolymers," in *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology*, pp. 28–36, AAAI Press, Menlo Park, California, 1994.

[8] BANDYOPADHYAY, S., SAHNI, S., and RAJASEKARAN, S., "PMS6: A fast algorithm for motif discovery," in *Proceedings of the IEEE 2nd International Conference on Computational Advances in Bio and Medical Sciences (ICCABS)*, pp. 1–6, IEEE, 2012.

[9] BECCHI, M., FRANKLIN, M., and CROWLEY, P., "A workload for evaluating deep packet inspection architectures," in *IEEE International Symposium on Workload Characterization*, pp. 79–89, Sept 2008.

[10] BECCHI, M. and CROWLEY, P., "Extending Finite Automata to Efficiently Match Perl-compatible Regular Expressions," in *Proceedings of the CoNEXT Conference*, (New York, NY, USA), pp. 25:1–25:12, ACM, 2008.

[11] BECCHI, M. and CROWLEY, P. J., *Data Structures, Algorithms and Architectures for Efficient Regular Expression Evaluation*. PhD thesis, Washington University. Department of Computer Science and Engineering., 2009.

[12] BECCHI, M., WISEMAN, C., and CROWLEY, P., "Evaluating regular expression matching engines on network and general purpose processors," in *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '09, (New York, NY, USA), pp. 30–39, ACM, 2009.

[13] BUHLER, J. and TOMPA, M., "Finding motifs using random projections," *Journal of Computational Biology*, vol. 9, no. 2, pp. 225–242, 2002.

[14] CARVALHO, A. M., FREITAS, A. T., OLIVEIRA, A. L., SAGOT, M.-F., and OTHERS, "A highly scalable algorithm for the extraction of cis-regulatory regions," in *Proceedings of Asia-Pacific Bioinformatics Conference*, pp. 273–283, 2005.

[15] CASCARANO, N., ROLANDO, P., RISSO, F., and SISTO, R., "iNFAnt: NFA pattern matching on GPGPU devices," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 5, pp. 20–26, 2010.

[16] CHIN, F. Y. L. and LEUNG, H. C. M., "Voting algorithms for discovering long motifs," in *Proceedings of third Asia-Pacific Bioinformatics Conference (APBC)*, vol. 3, pp. 261–271, 2005.

[17] CONSORTIUM, U. and OTHERS, "Activities at the Universal Protein Resource (UniProt)," *Nucleic acids research*, vol. 42, no. D1, pp. D191–D198, 2014.

[18] COOK, S. A., "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, pp. 151–158, ACM, 1971.

[19] DAVILA, J., BALLA, S., and RAJASEKARAN, S., "Fast and practical algorithms for planted (l, d) motif search," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 4, no. 4, pp. 544–552, 2007.

[20] DAVILA, J., BALLA, S., and RAJASEKARAN, S., "Pampa: An improved branch and bound algorithm for planted (l, d) motif search," tech. rep., 2007.

[21] DE CASTRO, E., SIGRIST, C. J., GATTIKER, A., BULLIARD, V., LANGENDIJK-GENEVAUX, P. S., GASTEIGER, E., BAIROCH, A., and HULO, N., "ScanProsite: detection of PROSITE signature matches and ProRule-associated functional and structural residues in proteins," *Nucleic acids research*, vol. 34, no. suppl 2, pp. W362–W365, 2006.

[22] DESSEN, P., FONDRAT, C., VALENCIEN, C., and MUGNIER, C., "BISANCE: a French service for access to biomolecular sequence databases," *Computer applications in the biosciences*, vol. 6, no. 4, pp. 355–356, 1990.

[23] DINH, H., RAJASEKARAN, S., and DAVILA, J., "qPMS7: A fast algorithm for finding (, d)-motifs in DNA and protein sequences," *PloS one*, vol. 7, no. 7, p. e41425, 2012.

[24] DINH, H., RAJASEKARAN, S., and KUNDETI, V. K., "PMS5: an efficient exact algorithm for the (l, d)-motif finding problem," *BMC bioinformatics*, vol. 12, no. 1, p. 410, 2011.

[25] DLUGOSCH, P., BROWN, D., GLENDENNING, P., LEVENTHAL, M., and NOYES, H., "An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 99, p. 1, 2014.

[26] ESKIN, E. and PEVZNER, P. A., "Finding composite regulatory patterns in DNA sequences," *Bioinformatics*, vol. 18, no. suppl 1, pp. S354–S363, 2002.

[27] EVANS, P. A. and SMITH, A. D., "Toward optimal motif enumeration," in *Algorithms and Data Structures*, pp. 47–58, Springer, 2003.

[28] EVANS, P. A., SMITH, A. D., and WAREHAM, H. T., "On the complexity of finding common approximate substrings," *Theoretical Computer Science*, vol. 306, no. 1, pp. 407–430, 2003.

[29] FLOYD, R. W. and ULLMAN, J. D., "The compilation of regular expressions into integrated circuits," *Journal of the ACM (JACM)*, vol. 29, no. 3, pp. 603–622, 1982.

[30] FLYNN, M., "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. 100, no. 9, pp. 948–960, 1972.

[31] FUCHS, R., "MacPattern: protein pattern searching on the Apple Macintosh," *Computer applications in the biosciences: CABIOS*, vol. 7, no. 1, pp. 105–106, 1991.

[32] FUCHS, R., "Predicting protein function: a versatile tool for the Apple Macintosh," *Computer applications in the biosciences: CABIOS*, vol. 10, no. 2, pp. 171–178, 1994.

[33] GATTIKER, A., GASTEIGER, E., and BAIROCH, A., "ScanProsite: a reference implementation of a PROSITE scanning tool," *Applied bioinformatics*, vol. 1, no. 2, pp. 107–108, 2002.

[34] GEOURJON, C. and DELEAGE, G., "Interactive and graphic coupling between multiple alignments, secondary structure predictions and motif/pattern scanning into proteins," *Computer applications in the biosciences: CABIOS*, vol. 9, no. 1, pp. 87–91, 1993.

[35] HENIKOFF, S. and HENIKOFF, J. G., "Automated assembly of protein blocks for database searching," *Nucleic Acids Research*, vol. 19, no. 23, pp. 6565–6572, 1991.

[36] HERTZ, G. Z. and STORMO, G. D., "Identifying DNA and protein patterns with statistically significant alignments of multiple sequences.," *Bioinformatics*, vol. 15, no. 7, pp. 563–577, 1999.

[37] HOPCROFT, J. E., *Introduction to automata theory, languages, and computation*. Pearson Education India, 1979.

[38] HP-LABS, "The Machine: A new kind of computer." http://www.hpl.hp.com/research/systems-research/themachine/, Nov. 2014.

[39] HULO, N., BAIROCH, A., BULLIARD, V., CERUTTI, L., CUCHE, B. A., DE CASTRO, E., LACHAIZE, C., LANGENDIJK-GENEVAUX, P. S., and SIGRIST, C. J., "The 20 years of PROSITE," *Nucleic acids research*, vol. 36, no. suppl 1, pp. D245–D249, 2008.

[40] IBM-RESEARCH, "A brain-inspired chip to transform mobility and Internet of Things through sensory perception." http://research.ibm.com/cognitive-computing/neurosynaptic-chips.shtml, Nov. 2014.

[41] IUPAC, I., "Nomenclature and symbolism for amino acids and peptides," *Eur J Biochem*, vol. 138, pp. 9–37, 1984.

[42] JÄRVISALO, M., LE BERRE, D., ROUSSEL, O., and SIMON, L., "The international sat solver competitions," *AI Magazine*, vol. 33, no. 1, pp. 89–92, 2012.

[43] KARP, R. M., *Reducibility among combinatorial problems*. Springer, 1972.

[44] KEICH, U. and PEVZNER, P. A., "Finding motifs in the twilight zone," in *Proceedings of the Sixth Annual International Conference on Computational Biology*, pp. 195–204, ACM, 2002.

[45] KELLY JR, T. J. and SMITH, H. O., "A restriction enzyme from *Hemophilus influenza*: II. Base sequence of the recognition site," *Journal of Molecular Biology*, vol. 51, no. 2, pp. 393–409, 1970.

[46] KOLAKOWSKI JR, L., LEUNISSEN, J., and SMITH, J., "ProSearch: fast searching of protein sequences with regular expression patterns related to protein structure and function.," *Biotechniques*, vol. 13, no. 6, pp. 919–921, 1992.

[47] KOZYRAKIS, C. E., PERISSAKIS, S., PATTERSON, D., ANDERSON, T., ASANOVIC, K., CARDWELL, N., FROMM, R., GOLBUS, J., GRIBSTAD, B., KEETON, K., and OTHERS, "Scalable processors in the billion-transistor era: IRAM," *Computer*, vol. 30, no. 9, pp. 75–78, 1997.

[48] LAWRENCE, C. E., ALTSCHUL, S. F., BOGUSKI, M. S., LIU, J. S., NEUWALD, A. F., WOOTTON, J. C., and OTHERS, "Detecting subtle sequence signals: a Gibbs sampling strategy for multiple alignment," *Science*, vol. 262, pp. 208–214, 1993.

[49] LIU, Q., WANG, L., FRUTOS, A. G., CONDON, A. E., CORN, R. M., and SMITH, L. M., "Dna computing on surfaces," *Nature*, vol. 403, no. 6766, pp. 175–179, 2000.

[50] MAGRANE, M., CONSORTIUM, U., and OTHERS, "UniProt Knowledgebase: a hub of integrated protein data," *Database*, vol. 2011, p. bar009, 2011.

[51] MARQUES-SILVA, J., "Practical applications of boolean satisfiability," in *9th International Workshop on Discrete Event Systems (WODES)*, pp. 74–80, May 2008.

[52] MARSAN, L. and SAGOT, M. F., "Extracting structured motifs using a suffix tree - Algorithms and application to promoter consensus identification," in *Proceedings of the Fourth Annual International Conference on Computational Molecular Biology*, pp. 210–219, ACM, 2000.

[53] MITRA, A., NAJJAR, W., and BHUYAN, L., "Compiling PCRE to FPGA for Accelerating SNORT IDS," in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS, (New York, NY, USA), pp. 127–136, ACM, 2007.

[54] NAKAHARA, H., SASAO, T., and MATSUURA, M., "A regular expression matching circuit based on a modular non-deterministic finite automaton with multi-character transition," in *The 16th Workshop on Synthesis and System Integration of Mixed Information Technologies*, pp. 359–364, 2010.

[55] NAKAHARA, H., SASAO, T., and MATSUURA, M., "A regular expression matching circuit based on a decomposed automaton," in *Reconfigurable Computing: Architectures, Tools and Applications*, pp. 16–28, Springer, 2011.

[56] NICOLAE, M. and RAJASEKARAN, S., "Efficient Sequential and Parallel Algorithms for Planted Motif Search," *arXiv preprint arXiv:1307.0571*, 2013.

[57] PEVZNER, P. A., SZE, S. H., and OTHERS, "Combinatorial approaches to finding subtle signals in DNA sequences," in *Proceedings of the Eighth International Conference on Intelligent Systems for Molecular Biology*, vol. 8, pp. 269–278, 2000.

[58] PISANTI, N., CARVALHO, A. M., MARSAN, L., and SAGOT, M.-F., "RISOTTO: Fast extraction of motifs with mismatches," in *LATIN 2006: Theoretical Informatics*, pp. 757–768, Springer, 2006.

[59] PRICE, A., RAMABHADRAN, S., and PEVZNER, P. A., "Finding subtle motifs by branching from sample strings," *Bioinformatics*, vol. 19, no. suppl 2, pp. ii149–ii155, 2003.

[60] RAJASEKARAN, S., BALLA, S., and HUANG, C. H., "Exact algorithms for planted motif problems," *Journal of Computational Biology*, vol. 12, no. 8, pp. 1117–1128, 2005.

[61] ROESCH, M. and OTHERS, "Snort: Lightweight Intrusion Detection for Networks.," in *LISA*, vol. 99, pp. 229–238, 1999.

[62] ROY, I. and ALURU, S., "Finding Motifs in Biological Sequences Using the Micron Automata Processor," in *IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 415–424, May 2014.

[63] RUBIN, F., "A search procedure for Hamilton paths and circuits," *Journal of the ACM (JACM)*, vol. 21, no. 4, pp. 576–580, 1974.

[64] SIBBALD, P. R. and ARGOS, P., "Scrutineer: a computer program that flexibly seeks and describes motifs and profiles in protein sequence databases," *Computer applications in the biosciences: CABIOS*, vol. 6, no. 3, pp. 279–288, 1990.

[65] SIGRIST, C. J., CERUTTI, L., HULO, N., GATTIKER, A., FALQUET, L., PAGNI, M., BAIROCH, A., and BUCHER, P., "PROSITE: a documented database using patterns and profiles as motif descriptors," *Briefings in bioinformatics*, vol. 3, no. 3, pp. 265–274, 2002.

[66] SIGRIST, C. J., DE CASTRO, E., CERUTTI, L., CUCHE, B. A., HULO, N., BRIDGE, A., BOUGUELERET, L., and XENARIOS, I., "New and continuing developments at PROSITE," *Nucleic acids research*, vol. 41, no. D1, pp. D344–D347, 2013.

[67] STADEN, R., "Screening protein and nucleic acid sequences against libraries of patterns," *Mitochondrial DNA*, vol. 1, no. 6, pp. 369–374, 1991.

[68] STERNBERG, M. J., "PROMOT: A FORTRAN program to scan protein sequences against a library of known motifs," *Computer applications in the biosciences: CABIOS*, vol. 7, no. 2, pp. 257–260, 1991.

[69] TIESSEN, A., PÉREZ-RODRÍGUEZ, P., and DELAYE-ARREDONDO, L. J., "Mathematical modeling and comparison of protein size distribution in different plant, animal, fungal and microbial species reveals a negative correlation between protein size and protein number, thus providing insight into the evolution of proteomes," *BMC research notes*, vol. 5, no. 1, p. 85, 2012.

[70] VASILIADIS, G., POLYCHRONAKIS, M., and IOANNIDIS, S., "Parallelization and characterization of pattern matching using GPUs," in *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 216–225, IEEE, 2011.

[71] WALLACE, J. C. and HENIKOFF, S., "PATMAT: a searching and extraction program for sequence, pattern and block queries and databases," *Computer applications in the biosciences: CABIOS*, vol. 8, no. 3, pp. 249–254, 1992.

[72] WANG, H., PU, S., KNEZEK, G., and LIU, J.-C., "MIN-MAX: A Counter-Based Algorithm for Regular Expression Matching," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 1, pp. 92–103, 2013.

[73] WANG, K., STAN, M., and SKADRON, K., "Association Rule Mining with the Micron Automata Processor," in *IEEE 29th International Parallel and Distributed Processing Symposium*, May 2015. To appear.

[74] YANG, Y.-H. and PRASANNA, V. K., "High-performance and compact architecture for regular expression matching on FPGA," *IEEE Transactions on Computers*, vol. 61, no. 7, pp. 1013–1025, 2012.

[75] YANG, Y. H. E., JIANG, W., and PRASANNA, V. K., "Compact architecture for high-throughput regular expression matching on FPGA," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pp. 30–39, ACM, 2008.

[76] YU, Q., HUO, H., ZHANG, Y., GUO, H., and GUO, H., "PairMotif+: A Fast and Effective Algorithm for De Novo Motif Discovery in DNA sequences," *International Journal of Biological Sciences*, vol. 9, no. 4, pp. 412–424, 2013.

[77] ZHENG, D., MHEMBERE, D., BURNS, R., VOGELSTEIN, J., PRIEBE, C. E., and SZALAY, A. S., "FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, (Santa Clara, CA), pp. 45–58, USENIX Association, Feb. 2015.

[78] ZHOU, K., FOX, J. J., WANG, K., BROWN, D. E., and SKADRON, K., "Brill tagging on the micron automata processor," in *IEEE 9th International Conference on Semantic Computing (ICSC)*, pp. 236–239, 2015.

[79] ZU, Y., YANG, M., XU, Z., WANG, L., TIAN, X., PENG, K., and DONG, Q., "GPU-based NFA implementation for memory efficient high speed regular expression matching," *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 129–140, 2012.

VITA

INDRANIL ROY

PERSONAL DATA

PLACE AND DATE OF BIRTH: India | 3 July 1983

ADDRESS: 1027 Hampton Street NW, Atlanta, GA 30318, USA.

PHONE: +1 515 509 4306

EMAIL: iroy@gatech.edu

RESEARCH INTERESTS

Design of parallel algorithms for high performance computing and bioinformatics.

EDUCATION

SPRING 2015 Ph.D.,**Georgia Institute of Technology**, Atlanta, GA.

SCHOOL OF COMPUTATIONAL SCIENCE & ENGINEERING

Minor: COMPUTER ARCHITECTURE & MIDDLEWARE

Dissertation: "Accelerated Pattern Matching using the Micron Automata Processor" | Advisor: Prof. Srinivas ALURU

GPA: 4.0/4.0

SPRING 2011 Master of Science, **Iowa State University**, Ames, IA.

DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

Major: Computer Engineeing

Thesis: "UPC CHECK: A scalable tool for detecting run-time errors in Unified Parallel C" | Advisor: Prof. Glenn R. LUECKE

GPA: 3.84/4.0

SPRING 2006 Bachelor of Engineering, **The National Institute of Engineering**, Mysore, India.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

AGGREGATE OF 8 SEMESTERS: First Class with Distinction (73.61%)

PUBLICATIONS

PEER REVIEWED

- **Roy, I.**; & Aluru, S., "Finding Motifs in Biological Sequences Using the Micron Automata Processor," IEEE 28th International Parallel and Distributed Processing Symposium, pp.415-424, 19-23 May 2014.
- **Roy, I.**, Luecke, G. R., Coyle, J., & Kraeva, M. "A scalable deadlock detection algorithm for UPC collective operations." PGAS, vol. 13, pp.2-15. 2012.
- Coyle, J.; **Roy, I.**; Kraeva, M.; & Luecke, G.R., "UPC-CHECK: a scalable tool for detecting run-time errors in Unified Parallel C," Computer Science - Research and Development, Springer-Verlag, vol. 28, no. 2-3, pp.203-209. 2012.
- Kraeva, M.; Coyle, J.; Luecke, G.; **Roy, I.**; Kleiman, E.; & Hoekstra, J.; "UPC-CompilerCheck: A Tool for Evaluating Error Detection Capabilities of UPC Compilers", COMPUTATION TOOLS, The Third International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking, pp.28-33. 2012.
- Iyenggar, A.; Tripathi, A.; Basarur, A.; & **Roy, I.**, "Unified power management

framework for portable media devices," IEEE International Conference on Portable Information Devices, PORTABLE07. pp.1-5. 2007.

ACCEPTED FOR PUBLICATION

- **Roy, I.**; & Aluru, S., "Discovering Motifs in Biological Sequences Using the Micron Automata Processor", IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB).

SUBMITTED FOR PUBLICATION

- Coyle, J.; Hoekstra, J.; Kraeva, M.; Luecke, G. R.; Kleiman, R.; & **Roy, I**, "UPC Compile-Time Error Detection Test Suite."
- **Roy, I.**; Jammula, N.; & Aluru, S., "Algorithmic Techniques for Solving Graph Problems on the Automata Processor", Submitted to the 27th annual IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing), 2015.

IN PREPARATION

- **Roy, I.**; Srivastava, A.; & Aluru, S., "Programming Non-deterministic Finite Automata on the Micron Automata Processor."
- **Roy, I.**; Srivastava, A.; Becchi, M.; & Aluru, S., "High Performance Pattern Matching using the Micron Automata Processor."
- **Roy, I.**; & Aluru, S., "Checking Boolean Satisfiability (SAT) using the Micron Automata Processor."

PROFESSIONAL EXPERIENCE

Engineer August 2006 - July 2008

Ittiam Systems (P) Ltd., Consulate 1, 1 Richmond Road, Bangalore, India.

- *Power Management Framework for Multimedia Sub-System*: Designed and implemented framework for power optimization in portable multimedia players.
- *ASF/MP4/MPG/AVI Creator*: Designed, implemented, tested and supported the ASF multiplexer. Involved in the design of MP4 multiplexer; and design and maintenance of MPG and AVI multiplexers.
- *ASF Parser*: Redesigned and implemented the ASF de-multiplexer.
- *Creator Subsystem*: Designed, implemented and automated testing of the Creator Subsystem.

PRACTICAL TRAINING

Intern May 2014 - August 2014

Micron Technology Inc. 8000 S. Federal Way, P.O. Box 6, Boise, ID 83707-0006.

- *Architecture Development Group*: Developed parallel algorithms to solve graph problems using the Micron Automata Processor.

Intern May 2013 - August 2013

Micron Technology Inc. 8000 S. Federal Way, P.O. Box 6, Boise, ID 83707-0006.

- *Architecture Development Group*: Developed solutions to problems in bioinformatics using the Micron Automata Processor.

Intern May 2012 - August 2012

Micron Technology Inc. 3060 N 1st St, San Jose, CA 95134.

- *Architecture Development Group*: Identified challenges and bottlenecks towards implementing bioinformatics applications on the Micron Automata Processor.

Research Assistant Aug 2013 - Present (except summer semesters)

Under Prof. Srinivas Aluru Georgia Institute of Technology, Atlanta, GA 30332.

- Designed and developed algorithms and applications using the Micron Automata Processor.

Research Assistant May 2011 - May 2013

Under Prof. Srinivas Aluru Iowa State University, Ames, IA 50011.

- Developed parallel algorithms for analyzing data from Atom Probe Microscope.
- Designed and developed algorithms and applications using the Micron Automata Processor.

Research Assistant January 2009 - May 2011

Under Prof. Glenn R. Luecke Iowa State University, Ames, IA 50011.

- *UPC-CHECK*: Developed a compile-time error detection tool for UPC under *DARPA's High Productivity Computing Initiative*.
- *UPC-CTED*: Wrote test cases for compile-time error detection tool for UPC.

Intern January 2006 - May 2006

Ittiam Systems (P) Ltd. Consulate 1, 1 Richmond Road, Bangalore, India.

- *Power Saving Strategies in Portable Media players*: Worked on increasing the audio play-out, video play-out, and stand-by duration of a portable multimedia player.

Intern January 2005 - June 2005

IBM Global Services India Pvt. Ltd. Airport Road, Bangalore, India.

- *Forecasting for Resource planning using Extrapolation and Explanatory methods in Fulfillments Operations Department*: Used complex extrapolation formulas to come up with very accurate predictions, even with sporadic past information.

PROFESSIONAL SKILLS

- Parallel Algorithm Design, Automata Design, High Performance Computing, Bioinformatics.
- Portable Multimedia-Systems, Power Management, Multimedia container formats.
- Advanced user: C, UPC, MPI.

HONORS AND AWARDS

- Invited Talk on "The Automata Processor for Bioinformatics" at IEEE Workshop on Microelectronics and Electron Devices (WMED), Boise, Idaho, April, 18, 2014.
- Best Paper Award in COMPUTATION TOOLS 2012, The Third International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking, July 22-27, 2012.
- 2^{nd} Prize in national level student paper presentation contest by Computer Society of India at Technologix'05, Mysore, India, April 1-2, 2005.
- 2^{nd} Prize in C-programming contest by IEEE student chapter at The National Institute of Engineering, Mysore, India, May 7-9, 2005.
- Best Volunteer, Technieks'05, Association of Computer Engineers, The National Institute of Engineering, Mysore, India, May 16-20, 2005.

EXTRA-CURRICULAR ACTIVITIES

Education and health-care of kids are important to me. I regularly participate in fund-raising activities for Non-Governmental Organizations which work in this domain, especially in under-developed or developing countries.

I like sports and oratory. I was the captain of my high-school oratory team and won many awards for the school. I represented my junior university in cricket. Currently, I like to play tennis and climb mountains. Besides these, I like to build and fly model airplanes.