GENERALIZED N-BODY PROBLEMS: A FRAMEWORK FOR SCALABLE COMPUTATION

A Thesis
Presented to
The Academic Faculty

by

Ryan Nelson Riegel

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

Georgia Institute of Technology
December 2013

GENERALIZED N-BODY PROBLEMS: A FRAMEWORK FOR SCALABLE COMPUTATION

Approved by:

Professor Alexander Gray, Advisor
School of Computational Science & Engineering
Georgia Institute of Technology

Professor Edmond Chow
School of Computational Science & Engineering
Georgia Institute of Technology

Professor Richard Vuduc
School of Computational Science & Engineering
Georgia Institute of Technology

Professor Gordon Richards
Department of Physics
Drexel University

Professor Charles Isbell
College of Computing
Georgia Institute of Technology

Date Approved: 20 August 2013

For my parents and for my grandparents, whose love and advice have always been invaluable.

ACKNOWLEDGEMENTS

I would like to thank my adviser, Alexander Gray, for his long support of my academic endeavors, and also for his dedication to the spread of both the knowledge and the use of fast algorithms for machine learning and statistics. His efforts have brightened my future and have helped ensure that the body of discourse that encompasses this thesis will remain impactful for years to come.

I would also like to thank the members of my thesis committee for their time and consideration, and to thank my fellow students in the FASTlab for the engaging collaborations we have shared. In particular, I want to thank Garrett Boyer, Abhimanyu Aditya, and William March, whose ideas and efforts have helped shape portions of this work. I also want to thank the Microsoft Corporation for their funding and support of the project described in Chapter 3 of this thesis, and to thank Gordon Richards for heading up the astrophysics application that serves as the primary motivation of Chapter 4.

I would further like to thank Martin Hack for his patience and for his financial assistance in my times of need. Lastly, but certainly not least, I would like to thank my family and especially my wife, Allison, whose constant encouragement, support, patience, and love have made all of this possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

In the wake of the Big Data phenomenon, the computing world has seen a number of computational paradigms developed in response to the sudden need to process ever-increasing volumes of data. Most notably, MapReduce has proven quite successful in scaling out an extensible class of simple algorithms to even hundreds of thousands of nodes. However, there are some tasks—even embarrassingly parallelizable ones—that neither MapReduce nor any existing automated parallelization framework is well-equipped to perform. For instance, any computation that (naively) requires consideration of all pairs of inputs becomes prohibitively expensive even when parallelized over a large number of worker nodes.

Many of the most desirable methods in machine learning and statistics exhibit these kinds of all-pairs or, more generally, all-tuples computations; accordingly, their application in the Big Data setting may seem beyond hope. However, a new algorithmic strategy inspired by breakthroughs in computational physics has shown great promise for a wide class of computations dubbed generalized N-body problems (GNBPs). This strategy, which involves the simultaneous traversal of multiple space-partitioning trees, has been applied to a succession of well-known learning methods, accelerating each asymptotically and by orders of magnitude. Examples of these include all-k-nearest-neighbors search, k-nearest-neighbors classification, k-means clustering, EM for mixtures of Gaussians, kernel density estimation, kernel discriminant analysis, kernel machines, particle filters, the n-point correlation, and many others. For each of these problems, no overall faster algorithms are known. Further, these dual- and multi-tree algorithms compute either exact results or approximations to within specified error bounds, a rarity amongst fast methods.

This dissertation aims to unify a family of GNBPs under a common framework in order to ease implementation and future study. We start by formalizing the problem class and then describe a general algorithm, the generalized fast multipole method (GFMM), capable of solving all problems that fit the class, though with varying degrees of speedup. We then show $O(N)$ and $O(\log N)$ theoretical run-time bounds that may be obtained under certain conditions. As a corollary, we derive the tightest known general-dimensional run-time bounds for exact all-nearest-neighbors and several approximated kernel summations.

Next, we implement a number of these algorithms in a commercial database, empirically demonstrating dramatic asymptotic speedup over their conventional SQL implementations. Lastly, we implement a fast, parallelized algorithm for kernel discriminant analysis and apply it to a large dataset (40 million points in 4D) from the Sloan Digital Sky Survey, identifying approximately one million quasars with high accuracy. This exceeds the previous largest catalog of quasars in size by a factor of ten and has since been used in a follow-up study to confirm the existence of dark energy.

CHAPTER I

THE GENERALIZED FAST MULTIPOLE METHOD

In this chapter, we present the mathematical foundations of a highly successful algorithmic strategy that has resulted in the fastest algorithms for a diverse collection of computations in physical simulation, computational geometry, machine learning, and beyond. Specifically, we formalize for the first time the class of *generalized N-body problems (GNBPs)*, which unifies N-body simulation, all-nearest-neighbors, kernel density estimation, the n-point correlation, and many others. We then show that these are all solved by the *generalized fast multipole method (GFMM)*, an abstract algorithm that may be specialized into an efficient, higher-order divide-and-conquer solution for any GNBP. Further, we demonstrate the potency of GNBPs and the GFMM by deriving a fast algorithm for the recently described affinity propagation method, achieving dramatic asymptotic speed-up.

This chapter is arranged as follows: We first consider the classical N-body simulation problem in physics, which has been solved efficiently by a number of algorithms characterized by the hierarchical decomposition of the input space into smaller, more manageable subproblems. We then consider several examples of how this same idea may be employed to accelerate methods in different fields, thereby demonstrating its general applicability. Next, we formally define the class of GNBPs and derive the GFMM, which solves them all. Lastly, we show experimental results for the GFMM and thus empirically confirm the orders-of-magnitude speedup that it can provide.

1.1 The classical N-body problem

The N-body problem is a fundamental and challenging problem in computational physics that lies at the heart of physical simulations in astronomy, cosmology, chemistry, and drug design, among other fields. It also appears in the computation of a variety of forces and potentials, such as the gravitational potential, which we will consider throughout the following. Given a set of N point masses with known initial positions and velocities, our task is to determine the trajectories of these masses as determined by Newtonian gravitation. In other words, we would like to know the position and velocity of each mass at any future (or past) point in time. The large number of variables in Newton's equations of motion make an analytic solution impossible in general for $N \geq 3$. Therefore, we must solve the problem numerically.

The standard numerical approach discretizes time. At each (small) time step, we compute the force—i.e. the negative gradient of the potential—on each mass due to all the other masses. We then update the positions and velocities of each mass as if they were subjected to this (constant) force for the entire time interval.[1] At each time step, computing the *exact* force on each point requires $O(N^2)$ operations, which becomes intractable for large N. It is thus of interest to consider *approximated* forces or potentials that can be computed much more quickly while still ensuring bounded error. For the remainder of this thesis, we use the term *"N-body problem"* to refer specifically to the computation of these forces or potentials at a given time step, e.g.:

$$\forall i \in I : \texttt{potential}(x_i) = -\sum_{j \neq i} G \frac{m(x_j)}{d(x_i, x_j)}. \tag{1}$$

Tackling this computational challenge has led the development of what has been judged to be one of the top ten algorithms of the 20th century [12]: the *fast multipole method (FMM)*. This algorithm is one of a family of solutions called *tree codes*, which

[1] More sophisticated time integrations are possible, such as leapfrog and Runge-Kutta. We omit discussion of these because our results do not deal with the time integration step.

can approximate force computations in $O(N \log N)$ overall time, including the cost of building the tree. In fact, after tree construction, the FMM completes all remaining work in just $O(N)$ time, a property commonly seen amongst other GNBPs.

In this section, we review three foundational and yet still commonly used tree codes in some detail: the Barnes-Hut simulation, Appel's algorithm, and the FMM. Throughout, we highlight the key features of the N-body problem that permit the development of fast algorithms.

1.1.1 The Barnes-Hut simulation

As the name suggests, tree codes perform computation via the traversal of space-partitioning trees. In particular, the *Barnes-Hut* algorithm for N-body simulation [4] traverses a single tree to approximate forces at individual query points. As we shall see, when a node is sufficiently distant from the query, it is possible to approximate the contribution of all points in the node in constant time.

The algorithm begins by constructing an *octree* on the set of input points. The root of an octree is a cube that contains all the points, and child nodes are formed by splitting each parent node's bounding box along the midpoints of each dimension, thereby forming eight equally sized regions. Empty nodes are omitted, and recursive construction stops when nodes contain just one point or, alternately, fewer than some user-specified number of points.

Computation then loops over all input points, invoking Algorithm 1 at each one paired with the octree's root. This computes the approximate potential at each query point q due to the entire input set X; accordingly, line 11 divides computation into subproblems representing the potentials at the query due to each octant of the current node. Recursion terminates in either of two ways: in the *base case* (lines 5–8) or by *pruning* (lines 3–4). The base case occurs when the tree node in question is a leaf, thus preventing any further recursion. In this case, the potential due to the leaf node

Algorithm 1 The Barnes-Hut simulation

```
    function BarnesHut(q, X, s)
        let d ← d(q, center(X))
 3:     if d > s · diameter(X) then
            potential(q) ← potential(q) − G m(X)/d
        else if is_leaf(X) then
 6:         for all x ∈ X, x ≠ q do
                potential(q) ← potential(q) − G m(x)/d(q,x)
            end for
 9:     else
            for all X′ ∈ children(X) do
                BarnesHut(q, X′, s)
12:         end for
        end if
    end function
```

is computed exactly, ignoring reference points in the special case that they match the query.

Pruning, on the other hand, occurs whenever the approximation of the node by a pseudoparticle at its center of mass can only incur at most ϵ relative error in the result. It can be shown that this will always occur when the distance between the query and the center of mass is greater than some multiple $s \in \Theta(\frac{1}{\epsilon})$ of the node's diameter. Because only constantly many nodes at a given depth can fit within the pruning radius, the running time of the algorithm is bounded by the maximum depth of the tree. For uniformly distributed data, the expected maximum depth of an octree is $O(\log N)$, and thus the potential at each may be computed in expected $O(\log N)$ time. Because there are $O(N)$ total queries, the overall running time is $O(N \log N)$, which is coincidentally the same as the cost of building the tree.

The key features from which this algorithm obtains it speedup are, first, the decomposition of problems into more manageable subproblems and, second, the approximation of all but the nearest subproblems with a quick pruning procedure. An additional note of importance is the fact that nodes' centers of mass—needed for pruning—may be found in just $O(N)$ time by an efficient leaf-to-root pass at the

time of tree construction. Storage of this precomputed information within the tree's nodes embodies the notion of *cached sufficient statistics*, a more sophisticated example of which is shown for the FMM.

1.1.2 Appel's algorithm

While the run-time complexity of the Barnes-Hut simulation already matches that of tree building, which it itself *lower bounded* in general at $\Omega(N \log N)$, the constant coefficients associated with computing forces or potentials greatly outweigh those of forming the tree. Therefore, it is desirable to optimize tree traversal to the greatest extent possible, as this ultimately extends the maximum feasible scope of computation.

In this vein, *Appel's algorithm* [2] simultaneously traverses over two trees (or, rather, two copies of the same tree)—one for the reference data and another for the queries—and thereby shares the work of distance computation and pruning over large groups of points. This algorithm relies on the same basic observation as Barnes-Hut: that the potential at a query point due to a distant collection of references is well-approximated by the potential due to a pseudoparticle at the references' center of mass. Appel's algorithm however makes a further observation: that the potential at a collection of nearby queries due to a distant reference point is well-approximated by the potential evaluated at the *queries'* center of mass. In conjunction, these two ideas allow for the frequent pruning of node pairs visited over the course of *dual-tree traversal*.

The entire computation is performed by invoking Algorithm 2 on the query and reference roots, Q and X. Similar to BarnesHut, line 14 recursively decomposes work into smaller problems, but it must additionally split the query node and pair each query child with each reference child to ensure consideration of all pairs of points. The base case (lines 5–10) is relatively unchanged, however special handling (elided)

Algorithm 2 Appel's algorithm

```
    function APPELS(Q, X, s)
        let d ← d(center(Q), center(X))
 3:     if d > s · (diameter(Q) + diameter(X)) then
            potential(Q) ← potential(Q) − G m(X)/d
        else if is_leaf(Q) and is_leaf(X) then
 6:         for all q ∈ Q do
                for all x ∈ X, x ≠ q do
                    potential(q) ← potential(q) − G m(x)/d(q,x)
 9:             end for
            end for
        else
12:         for all Q′ ∈ children(Q) do
                for all X′ ∈ children(X) do
                    APPELS(Q′, X′, s)
15:             end for
            end for
        end if
18: end function
```

is necessary when one node becomes a leaf while the other remains interior. Pruning (lines 3–4) must consider both nodes' diameters, but is otherwise also similar to the Barnes-Hut simulation.

Appel's original claim was that the above method is $O(N \log N)$, however Esselink [19] later proved that the algorithm is in fact $O(N)$ for uniform distributions. This thesis, on the other hand, proves that the algorithm is $O(N)$ even for arbitrarily distributed data. In short, because the pruning radius may again be expressed as a constant multiple of node width, only constantly many nodes at a given depth may ever be paired with a splitting node. Because there are only $O(N)$ total nodes, and because the work performed at any single node pair is constant and is the direct result of a node split, the overall running time is itself $O(N)$.

An important final step of computation is to propagate pruned potentials found for interior nodes to the leaves of the query tree. Rather than performing this step as soon as pruning occurs, which would incur $O(N \log N)$ work overall, propagation of results must be *postponed* until after dual-tree traversal. It can then be completed

in an $O(N)$ root-to-leaf pass—effectively the reverse of the computation of cached sufficient statistics. Again, a more sophisticated example of this is shown for the FMM.

1.1.3 The fast multipole method

The FMM [30] follows in steps of Appel's algorithm by considering pairs of nodes and approximating computation at nodes that are sufficiently separated from one another. However, unlike Appel's algorithm, which employs the *monopole* (center of mass) approximation for well-separated pairs, the FMM maintains higher-order *multipole* expansions about various points, thereby enabling it to prune even more aggressively. In fact, the FMM always prunes all nodes that are not immediately adjacent to the query node, and the specified error tolerance ϵ controls the order p of truncated series expansion instead of the pruning radius.

The FMM uses two different kinds of series expansions: the multipole expansion, which approximates the potential due to a distant group of reference points, and the local expansion, which approximates the potential at a nearby group of query points. The chief task throughout computation of the FMM is then the *translation* of series expansions from one central point to another as well as from multipole to local. Because these translations feature significantly in the computation of cached sufficient statistics and the propagation of postponed results, we explicitly demonstrate these phases (FmmUpward and FmmDownward, respectively) along with the usual dual-tree traversal (FmmSideToSide) in Algorithm 3.

Translation is facilitated by a host of theorems developed by Greengard for both the 2- and 3-dimensional cases [31]. The remarkable intricacy of these theorems is beyond the scope of this thesis, as their particular formulations have no direct bearing on the nature of the GFMM. It is enough to say that there exist means of translation that adequately bound error for $p \in \Theta(\lceil \log \frac{1}{\epsilon} \rceil)$. The cost of operations on

Algorithm 3 The fast multipole method

```
    function FmmUpward(X, p)
        if is_leaf(X) then
            Compute a multipole expansion about X for all x ∈ X
        else
            for all X' ∈ children(X) do
                FmmUpward(X', p)
                Translate multipole expansion about X' into that of X
            end for
        end if
    end function
    function FmmSideToSide(Q, X, p)
        if d_min(Q, X) > 0 then // i.e. if Q and X are nonadjacent
            Convert multipole expansion about X into local expansion about Q
        else if is_leaf(Q) and is_leaf(X) then
            for all q ∈ Q do
                for all x ∈ X, x ≠ q do
                    potential(q) ← potential(q) − G m(x)/d(q,x)
                end for
            end for
        else
            for all Q' ∈ children(Q) do
                for all X' ∈ children(X) do
                    FmmSideToSide(Q', X', p)
                end for
            end for
        end if
    end function
    function FmmDownward(Q, p)
        if is_leaf(Q) then
            Evaluate the local expansion about Q at all q ∈ Q
        else
            for all Q' ∈ children(Q) do
                Translate local expansion about Q into that of Q'
                FmmDownward(Q', p)
            end for
        end if
    end function
```

series expansions is bounded by a constant $O(p^2)$ and thus, considering the restricted pruning radius, the algorithm is again $O(N)$ overall.

At this point, we feel it is important to note that, while most discussion of the FMM revolves around the creation and manipulation of series expansions as its primary source of speedup, in fact the FMM is fast for the same reasons that the Barnes-Hut simulation and Appel's algorithm are fast. All three of these tree codes embody the concepts of divide and conquer, and bounding the total number of divisions that must ultimately be "conquered" is the key to deriving their worst-case run-time complexities. The translation theorems of the FMM are simply a means of proving that its particularly aggressive form of pruning is safe; the other tree codes with their loosened pruning radii are nonetheless slower by at most a constant dependent on ϵ, and at that their cost of pruning is significantly reduced. We shall see in Chapter 2 that a generalized notion of the pruning radius may in fact be used to bound the running times of all GNBPs.

1.2 Tree codes in other applications

In light of the breakthrough algorithms in physical simulation described above, efforts have been made to apply similar solution techniques to other computations that traditionally require $O(N^2)$ time or worse. In particular, a great deal of effort has been made in the field of computational geometry to derive algorithms for nearest-neighbors search and related problems based on efficient traversals of spatially-informed data structures. One example is the well-separated pair decomposition [11], which was the first work to unify the computations of all-nearest-neighbors and the N-body problem. Another example, considered in some detail below, is Gray's discussion of "N-body problems" in statistical learning [25], which explores all-nearest-neighbors, kernel density estimation, and the n-point correlation. Following Gray's publication, a series of other papers have applied the same algorithmic techniques in solving yet

Algorithm 4 Nearest-neighbors search

```
    function NnSearch(q, X)
        if d_min(q, X) ≥ distance(q) then return
 3:     else if is_leaf(X) then
            for all x ∈ X do
                distance(q) ← min{distance(q), d(q, x)}
 6:         end for
        else
            for all X′ ∈ children(X) prioritized by d_min(q, ·) do
 9:             NnSearch(q, X′)
            end for
        end if
12: end function
```

other *generalized N-body problems* [43, 44, 65, 34, 46], a class of problems that we formally describe for the first time in Section 1.3 of this thesis.

1.2.1 Nearest-neighbors search

Nearest-neighbors search is a fundamental problem in computational geometry that serves as a preliminary step of many other methods, including many methods in machine learning. For instance, k-nearest-neighbors classification uses found neighbors directly as "votes" to predict the class of each query point [44]. On the other hand, some other methods sparsify input similarity matrices so as to only represent relationships between each point's k nearest other points [18]. Yet further applications also exist in manifold dimensionality reduction [66, 74] and clustering [57, 45].

For being such a core analysis in so many methods, the *nearest-neighbors search problem* [59] is relatively easy to express:

$$\texttt{neighbor}(q) = \arg\min_{j \in J} d(q, x_j); \tag{2}$$

that is, for a given query point q, find the reference point x_j with minimum distance to q. Alternately, the k-nearest-neighbors search problem slightly extends the above to return a list containing the k nearest such points, which we indicate with $\arg\min_k$.

Algorithm 5 All-nearest-neighbors

```
    function AllNn(Q, X)
        if d_min(Q, X) ≥ distance(Q) then return
 3:     else if is_leaf(Q) and is_leaf(X) then
            for all q ∈ Q do
                for all x ∈ X, x ≠ q do
 6:                 distance(q) ← min{distance(q), d(q, x)}
                end for
            end for
 9:         distance(Q) ← max{distance(q) : q ∈ Q}
        else
            for all Q' ∈ children(Q) do
12:             for all X' ∈ children(X) prioritized by d_min(Q', ·) do
                    AllNn(Q', X')
                end for
15:         end for
            distance(Q) ← max{distance(Q') : Q' ∈ children(Q)}
        end if
18: end function
```

Another variation of this problem is *all-nearest-neighbors*,

$$\forall i \in I : \texttt{neighbor}(x_i) = \arg\min_{j \in J} d(x_i, x_j), \tag{3}$$

which in the *monochromatic* case—i.e. when $I = J$—must be careful not to identify input points x_i as their own nearest neighbors.

Clearly, if implemented by way of looping over all input data, the nearest-neighbors search problem is $O(N)$ and all-nearest-neighbors is $O(N^2)$. As a result—and because some past methods bounding the run-time complexity of nearest-neighbors search have proven impractical—many studies that require the identification of points' nearest neighbors have so far focused on small datasets, and a number of approximate nearest-neighbors search methods have been proposed, e.g. [37]. However, as has been demonstrated [23, 25, 6] and is reinforced in this thesis, *fast, practical, and exact* nearest-neighbors search methods are possible, even in high-dimensional and/or non-Euclidean spaces.

Algorithms 4 and 5 depict pseudocode that quickly solves the nearest-neighbors

search problem and all-nearest-neighbors, respectively. Pruning is made possible by annotating each query node (or the query point) with an upper bound on the distance to any contained point's candidate nearest neighbor. Then, whenever the minimum distance between the query and a reference node is greater than this upper bound, the reference node may be pruned with no additional work. Accordingly, key in both algorithms is the prioritization of reference tree traversal (e.g. line 12 of Algorithm 5) so that nearer nodes are visited first. This heuristic increases the chances of finding nearer candidate neighbors earlier, which in turn encourages earlier and more frequent pruning.

The difference between the fast algorithms for nearest-neighbors search and all-nearest-neighbors is the same as the difference between the Barnes-Hut simulation at a single query and Appel's algorithm. Correspondingly, nearest-neighbors search can be shown to be $O(\log N)$ in various settings [23, 6] and all-nearest-neighbors can be shown to be $O(N)$ after index construction [11, 6]. In this thesis, we derive for both methods bounds with the tightest known dimensional coefficients as corollaries to our abstract analysis of the GFMM.

1.2.2 Kernel density estimation

Probability density is a core concept in machine learning and statistics that plays a major role in topics as fundamental as computing the expected value of a random variable. While it is rarely possible to know probability density exactly, even estimated probability densities can result in extremely powerful classifiers [65], regressors [35], and other forms of prediction [76]. *Kernel density estimation (KDE)* [68] is a nonparametric means of estimating probability density and, as such, is able to adapt to the data's underlying distribution without explicit specification of a data model. This property makes KDE ideal for many tasks where the underlying model is poorly understood or hopelessly complicated, such as computer vision [16, 50] and

bioinformatics [20, 52].

KDE models a dataset's overall probability density function (pdf) as the average of a multitude of simple pdfs centered at each point in the dataset. While not strictly necessary, it is often convenient to restrict allowable pdfs to kernel functions of distance:

$$\widehat{f}(q) = \frac{1}{N} \sum_{j \in J} K_h(d(q, x_j)). \tag{4}$$

For example, one might define kernel K_h with bandwidth h to be the Gaussian pdf:

$$K_h(v) = \frac{1}{(h\sqrt{2\pi})^D} e^{-\frac{1}{2}(\frac{v}{h})^2}; \tag{5}$$

alternate kernels, such as the Epanechnikov kernel [17], also exist with various desirable properties. Regardless, an important prerequisite of any application of KDE is optimal bandwidth selection. It can be shown that optimal h shrinks as $O(N^{-1/5})$ [68]; however, in practice it is best to choose h to minimize *leave-one-out cross-validation (LOOCV) error* for the method at hand. LOOCV kernel density estimates may be computed

$$\forall i \in I : \widehat{f}(x_i) = \frac{1}{N-1} \sum_{j \neq i} K_h(d(x_i, x_j)), \tag{6}$$

which itself may be computed by summing over all $j \in I$ and then subtracting $K_h(0)$ before normalization, though this is less numerically stable.

As with both the N-body problem and all-nearest-neighbors, implementation of LOOCV KDE as nested loops results in a run-time complexity of $O(N^2)$. Fast $O(N \log N)$ and $O(N)$ non-hierarchical solutions have been proposed via the fast Fourier transform [69] and series expansions [32, 78], respectively, but these either are restricted to low-dimensional Euclidean space or fail to perform as expected under various conditions [8]. On the other hand, dual-tree algorithms like the one shown in Algorithm 6 have been shown to be fast both in practice [27, 28, 43, 41, 42] and in theory [62], again even for non-Euclidean metric spaces. The resulting algorithms

Algorithm 6 All-pairs KDE

```
    function KDE(Q, X, h, ε)
        if K_h(d_min(Q, X)) − K_h(d_max(Q, X)) < ε then
 3:         density(Q) ← density(Q) + |X| · K_h(d(center(Q), center(X)))
        else if is_leaf(Q) and is_leaf(X) then
            for all q ∈ Q do
 6:             for all x ∈ X do
                    density(q) ← density(q) + K_h(d(q, x))
                end for
 9:         end for
        else
            for all Q′ ∈ children(Q) do
12:             for all X′ ∈ children(X) do
                    KDE(Q′, X′, h, ε)
                end for
15:         end for
        end if
    end function
```

are very similar to Appel's algorithm and the FMM, as both N-body simulation and KDE represent sums over functions of distance. However, because kernel functions generally lack the relationship present in the N-body problem between relative error in distance and in the result, pruning must operate under slightly different circumstances, as shown in line 2. As we shall see, this difference has some impact on the algorithm's run-time analysis. Nonetheless, this thesis proves KDE to be $O(N)$ after index construction, again as a corollary to the run-time bound of the GFMM and again resulting in the tightest known dimensional coefficients for the general dimensional case.

1.2.3 The n-point correlation

In astronomy, a correlation function is a measure of fractal dimension that, among other applications, may be used to differentiate between data distributions—e.g. between the distributions of simulated and observed datasets. The *n-point correlation function* [56] in particular is a measure of the data's "lumpiness." It has been used to study properties of the cosmic microwave background radiation [71], the large-scale

structure of the universe [54, 49, 48], and dark energy [1].

In the case that $n = 2$, the n-point correlation function hinges upon the sum

$$X^{(2)}(h) = \sum_{i \in J} \sum_{j > i} [d(x_i, x_j) \leq h], \tag{7}$$

where Iverson bracket $[\cdot]$ returns 1 when its contents are true and 0 otherwise. This has the effect of counting the number of unique unordered pairs of distinct points (x_i, x_j) such that x_i and x_j are within distance h of one another. For $n > 2$, the relevant sum of the n-point correlation might instead be expressed

$$X^{(n)}(h) = \sum_{j_1 \in J} \sum_{j_2 > j_1} \cdots \sum_{j_n > j_{n-1}} [d(x_{j_k}, x_{j_l}) \leq h : 1 \leq k < l \leq n], \tag{8}$$

which instead counts n-tuples of points that satisfy the bracketed *matcher*. More sophisticated matchers are also possible, which for instance might bound distance both above some h_{lo} and below some h_{hi}, or might have separate bounds for each of the $\binom{n}{2}$ distances between the n points [46, 47]. While these complicate computation, we shall see that they play only a minor role in the asymptotic running time.

Algorithm 7 approximates the 2-point correlation to within ϵ absolute error in the threshold distance h. It is easy to see that, because all pairs within $h - \epsilon$ will be counted as usual and all pairs outside $h + \epsilon$ will be excluded as usual, the resultant approximation $\widehat{X}^{(2)}(h)$ is bounded between the exact results of $X^{(2)}(h - \epsilon)$ and $X^{(2)}(h + \epsilon)$. If one then plots $\widehat{X}^{(2)}(h)$ as a function of h, this is equivalent to having error bars along the the x-axis rather than the usual y-axis.

Observe that Algorithm 7 is different from preceding algorithms in that it obeys *triangular recursion* with lines 2 and 6, which enforces the constraint that $j > i$ in Equation 7. Further, TwoPt is able to prune not only via *exclusion* of node pairs that are distant from one another (line 3) but also via *inclusion* of pairs that are close-in (lines 4–9). While both of these optimizations have the effect of reducing the amount of overall work, they have no impact on the asymptotic running time.[2]

[2]Monochromatic all-nearest-neighbors, LOOCV KDE, and the N-body problem can also all make

Algorithm 7 The 2-point correlation

```
    function TwoPt(X_1, X_2, h, ε)
        if begin(X_1) > end(X_2) then return 0
 3:     else if d_min(X_1, X_2) > h − ε then return 0
        else if d_max(X_1, X_2) ≤ h + ε then
            if X_1 = X_2 then
 6:             return (1/2)|X_1|(|X_1| − 1)
            else
                return |X_1| · |X_2|
 9:         end if
        else if is_leaf(X_1) and is_leaf(X_2) then
            ... // Base case elided
12:     else
            let sum ← 0
            for all X'_1 ∈ children(X_1) do
15:             for all X'_2 ∈ children(X_2) do
                    sum ← sum + TwoPt(X'_1, X'_2, h, ε)
                end for
18:         end for
            return sum
        end if
21: end function
```

For $n' = \min\{n, D+1\}$, previous papers have given both conjectured [25] and observed [51] asymptotic run-time bounds of $O(N^{n' - \frac{1}{D}\binom{n'}{2}})$ for certain exact n-point correlation functions in Euclidean space. These bounds only apply to rigid, simplex-like matchers, and yet have exponent *strictly* greater than 1 for all dimensionalities $D > 1$. This thesis, on the other hand, proves theoretical run-time bounds for a broader class of 2-point and n-point correlation functions in the general dimensional case, both of which are surprisingly $O(N)$, though with dimensional coefficient growing exponentially in n. These bounds, along with our bound for the Axilrod-Teller potential in Section 2.7, also represent the first such bounds for GNBPs of order greater than 2, which we explain in the following section.

use of triangular recursion, effectively halving the number of distance computations and visited node pairs. Further, KDE with the Epanechnikov kernel is capable of both exclusion and inclusion pruning, as demonstrated in Chapter 4. Because these forms of pruning can occur even when $\epsilon = 0$, accelerated exact computation is possible for both the n-point correlation and Epanechnikov KDE; however, asymptotic run-time bounds are less favorable for this case.

1.3 Generalized N-body problems

It is clear from the discussion above and from the content of Algorithms 1–7 that very similar algorithmic strategies can efficiently solve the N-body problem, nearest-neighbors search, KDE, and the n-point correlation despite these problems' disparate properties and requirements. Further, the mathematical representations of each of these problems given in Equations 1–4 and 6–8 all visibly adhere to a motif of aggregating over values computed at all pairs (or larger groupings) of inputs. Considering these two observations, this section provides the first formal definition of an important class of GNBPs, which unites the problems described above along with many others into a single family and thereby lays the foundations for the major claim of this thesis:

Claim 1 (Thesis statement)**.** All generalized N-body problems (GNBPs) may be solved by an abstract algorithm, the generalized fast multipole method (GFMM), which can be implemented as a generic software framework and can be shown to be asymptotically fast under certain circumstances.

1.3.1 Reductions

As noted above, the aggregation—or *reduction*—is a characteristic feature of GNBP computation. Throughout this thesis, we consider reductions over *index sets*, e.g. over all $j \in J$, as opposed to directly over the data. This both enables datasets to contain identical points at distinct indices and allows us to consider alternate uses of indices, such as testing whether $i < j$ in the 2-point correlation. We thus define:

Definition 1. Given a commutative, associative binary operation $\oplus\colon \mathcal{P} \times \mathcal{P} \to \mathcal{P}$ with identity element $e_\oplus \in \mathcal{P}$, a *reduction* $\bigoplus$ is defined such that, for any (finite)

indexed family $(v_j \in \mathcal{P})_{j \in J}$,

$$\bigoplus_{j \in J} v_j = v_{j_{(1)}} \oplus v_{j_{(2)}} \oplus \ldots \oplus v_{j_{(N)}}$$

if index set J is itself defined $J = \{j_{(1)}, j_{(2)}, \ldots, j_{(N)}\}$.

Note that this is distinct from higher-order function *fold* in that fold operates on an ordered list and thus permits binary operations that are neither commutative nor associative. We require these properties because, in order to accelerate computation, we must have the ability to partition and rearrange subcomputations in a manner more conducive to pruning. We further require an identity element for later convenience; note, however, that any $\mathcal{P}$ may be augmented with an element $e_\oplus$ defined by fiat to be the identity of $\oplus$. Accordingly, operation $\oplus$ in conjunction with its domain of *partial results* $\mathcal{P}$ specifies a commutative monoid.[3]

Again looking at the problems discussed above, we observe that some GNBPs bear more than one (of the same) reduction, as in the 2-point and n-point correlations, but also that many GNBPs involve *nonreduced* index sets, as in all-nearest-neighbors and LOOCV KDE. Such index sets serve as batch queries, i.e. they produce separate results for each $i \in I$. Throughout the above, we achieve this behavior with a combination of the universal quantifier $\forall$ and the assignment of results to a family of variables p_i indexed by $i \in I$. An alternate representation of this same behavior instead makes use of the *disjoint union* to form the set containing all pairs (i, p_i):

Definition 2. The *disjoint union* $\biguplus$ is defined such that, for any (finite) indexed family $(p_i \in \mathcal{P})_{i \in I}$,

$$\biguplus_{i \in I} p_i = \bigcup_{i \in I} \{(i, p_i)\}.$$

[3]This is not a group because we do not require inverse elements, and indeed useful reductions such as min and max do not offer these.

Further, for indexed families of disjoint unions $(P_i \subseteq \mathcal{I}^k \times \mathcal{P})_{i \in I}$ and any integer $k \geq 1$, we generalize

$$\biguplus_{i \in I} P_i = \bigcup_{i \in I} \{((i, i'), p) : (i', p) \in P_i\}.$$

If index set I is defined $I = \{i_{(1)}, i_{(2)}, \ldots, i_{(M)}\}$, then $\biguplus_{i \in I} p_i$ may be interpreted as forming a length M vector of results.[4] Likewise, nested disjoint unions may be interpreted as forming matrices or, more generally, tensors of results. Because the disjoint union losslessly preserves all of its inputs, we do not consider it to be a true reduction; rather, throughout the below, we understand it more as an explicit means of annotating nonreduced index sets, the exact representation of how they store their results being of no great importance.

Notational conventions. We establish a convention of using the symbol I for nonreduced index sets and the symbol J for reduced index sets. If there are multiple nonreduced index sets, we write them as I_k for $1 \leq k \leq m$. Similarly, if there are multiple reduced index sets, we write them as J_k for $1 \leq k \leq n$. The number $m + n$, or the total number of index sets, is said to be the *order* of the GNBP. We further define capital symbols $M = |I|$ and $N = |J|$ (with subscripted versions intuited as above); these establish the "N-body problem" as referring to *the size of the dataset* while the "n-point correlation" refers to *the number of reductions*. All index sets are taken to be subsets of the domain of indices $\mathcal{I}$. Lastly, we sometimes make use of a shorthand notation to bind an index variable via binary relation with another index variable, e.g. $\bigoplus_{j \neq i}$, in which case the left-hand side is understood to originate from

[4]Here and in Definition 1 we are careful not to insist that index sets I and J actually be, e.g., the integers from 1 to M because it is notationally useful to allow $(x_i)_{i \in I}$ and $(x_j)_{j \in J}$ to refer to distinct query and reference sets (for $I \cap J = \emptyset$). One can instead think of I and J as referring to disjoint consecutive ranges of memory addresses pertaining to the two datasets.

the same index set as the (previously bound) right-hand side.

1.3.2 Mapped functions

While, as we shall see, the number of disjoint unions and repetitions of the chosen reduction have a profound impact on the overall shape of computation, just as important in problem specification is the *mapped function* to be evaluated at all visited combinations of indices. Each nonreduced and reduced index set contributes one indexed input—e.g. x_i or x_j—to the mapped function. These inputs denote points in a metric[5] space $\mathcal{X}$; however, some problems also require metadata at each point, such as class labels, regression targets, or the point's mass or charge as in the classical N-body problem. When needed, we indicate these inputs with the appropriate symbols, but in the general case, we lump them together into a nonmetric set $\mathcal{Y}$. Hence:

Definition 3. Given a metric space $\mathcal{X}$ and a (possibly trivial) nonmetric set $\mathcal{Y}$, the *mapped function* of a problem with m nonreduced index sets and n reduced index sets is a function f of the form[6]

$$f\colon \mathcal{X}^m \times \mathcal{X}^n \times \mathcal{Y}^m \times \mathcal{Y}^n \to \mathcal{P}.$$

In our generalized formulation of GNBPs, applications of f are written in terms of m- and n-tuples of elements of $\mathcal{X}$ and $\mathcal{Y}$, exactly as suggested by Definition 3:

$$f(\mathbf{x_i}, \mathbf{x_j}, \mathbf{y_i}, \mathbf{y_j}) \tag{9}$$

[5]A true metric space may not be strictly necessary. Pseudometrics (relaxed identity: $d(a,b) = 0$ for $a \neq b$), quasimetrics (relaxed symmetry: $d(a,b) \neq d(b,a)$), and perhaps inframetrics (relaxed triangle inequality: $d(a,c) \leq \rho \max\{d(a,b), d(b,c)\}$) may be tolerable with some adaptations, but are not explored in this thesis.

[6]It is tempting to entertain the idea that each of the $m+n$ index sets might need to be paired with a different set $\mathcal{X}_k$; however, we note that spatial comparisons make little sense between points residing in distinct metric spaces. On the other hand, some problems may require one or either of x_i or x_j, etc., to represent bounded regions of space rather than points. This may be achieved with $\mathcal{X}' = 2^{\mathcal{X}}$ and an appropriately defined pseudometric (e.g. $d(a', b') = \inf\{d(a,b) : a \in a', b \in b'\}$).

for tuples $\mathbf{i} = (i_1, \ldots, i_m)$ and $\mathbf{x_i} = (x_{i_1}, \ldots, x_{i_m})$, etc. A given problem's particular instantiation of f may then make use of any arrangement of the contents of these tuples to compute its result.

Note that we have defined mapped functions in a manner that permits them to consider spatial properties of data other than distance, such as the inner product or angle. For example, the Axilrod-Teller potential is a problem that, in part, depends upon angles induced by triples of points, and recent publications have proposed novel data structures to accelerate both SVD [36] and maximum inner-product search [61]. Functions of distance are, however, by far the most common in the body of GNBPs examined thus far.

A final note about mapped functions is that, while their formulation does not explicitly allow them to consider input indices, it is often convenient for them to be able to do so. For instance, many methods must not visit index pairs $i = j$ because doing so would either taint results or cause mathematical singularities. One means of avoiding this is to have f test whether $i = j$ and, if so, return $e_\oplus$ in place of its usual result.[7] Reconciling this practice with Definition 3, we observe that elements $y_i \in \mathcal{Y}$ can be taken to contain their own index, with for example $y_i = (i, y_i')$.

Notational conventions. While, in the general case, mapped functions f are expressed as functions of four tuples, specific cases of f may be awkward to represent in exactly that manner. In particular, when either or both of m or n are small or 0, or when either or both of $\mathbf{y_i}$ or $\mathbf{y_j}$ are unused, the general formulation may seem needlessly verbose. Accordingly, we either omit unused or empty function arguments or explicitly note their lack of use with placeholder $\cdot$, as in

$$f(\cdot, \mathbf{x_j}, \cdot, \cdot), \tag{10}$$

[7] While such tests are essential for proper computation, we typically omit them when expressing specific functions f because they are intuitive and yet overcomplicate the underlying form.

which happens to be appropriate for the n-point correlation function. Additionally, if m or n are small, say 1 or 2, we de-tuple the content of the affected inputs into separate arguments, as in

$$f(x_i, x_{j_1}, x_{j_2}), \tag{11}$$

which is appropriate for the expression of the $m = 1$, $n = 2$ Axilrod-Teller potential.

1.3.3 Problem class definition

Combining the reduction, disjoint union, and mapped function from Definitions 1–3, we arrive at an important and already very broad first class of GNBPs, the *GNBPs of one reduction*, which we regard as the basis of all GNBPs:

Definition 4. Given m nonreduced and n reduced index sets, a binary operation of reduction $\oplus$, and a mapped function f, a *generalized N-body problem of one reduction (1-GNBP)* is a problem $P_{m,n,\oplus,f}$ of the form

$$P_{m,n,\oplus,f}(\mathbf{I}, \mathbf{J}; X, Y) = \biguplus_{\mathbf{i}\in\mathbf{I}} \bigoplus_{\mathbf{j}\in\mathbf{J}} f(\mathbf{x_i}, \mathbf{x_j}, \mathbf{y_i}, \mathbf{y_j}),$$

where nonreduced indices $\mathbf{I}$ and reduced indices $\mathbf{J}$ are (possibly trivial) sets of index tuples formed[8]

$$\mathbf{I} = I_1 \times \cdots \times I_m \quad \text{and} \quad \mathbf{J} = J_1 \times \cdots \times J_n;$$

indexed families $X = (x_i)_{i\in I\cup J}$ and $Y = (y_i)_{i\in I\cup J}$ denote metric and nonmetric input datasets, respectively; function arguments $\mathbf{x_i}$, $\mathbf{x_j}$, $\mathbf{y_i}$, and $\mathbf{y_j}$ are tuples of elements of X and Y defined, e.g.,

$$\mathbf{x_i} = (x_{i_1}, \ldots, x_{i_m}) \quad \text{for} \quad \mathbf{i} = (i_1, \ldots, i_m);$$

reduction $\bigoplus$ is the iterated form of $\oplus$; and $\biguplus$ is the disjoint union.

[8] We further define combined index sets $I = \bigcup_{1\le k\le m} I_k$ and $J = \bigcup_{1\le k\le n} J_k$, thereby establishing

Table 1: All problems seen in this thesis so far in terms of 1-GNBP $P_{m,n,\oplus,f}$.

Problem	m	n	$\oplus$	f	Notes
Barnes-Hut	0	1	$+$	$G\frac{m_j}{d(q,x_j)}$	$y_j = m_j$
Appel's alg.	1	1	$+$	$G\frac{m_j}{d(x_i,x_j)}$	$y_j = m_j$
NN search	0	1	$\arg\min$	$d(q, x_j)$	
All-NN	1	1	$\arg\min$	$d(x_i, x_j)$	
KDE	0	1	$+$	$\frac{1}{N}K_h(d(q, x_j))$	
LOOCV KDE	1	1	$+$	$\frac{1}{N-1}K_h(d(x_i, x_j))$	
2-point corr.	0	2	$+$	$[d(x_{j_1}, x_{j_2}) \leq h]$	
n-point corr.	0	n	$+$	$[d(x_{j_k}, x_{j_l}) \leq h : 1 \leq k < l \leq n]$	

Without further modification, this formulation successfully generalizes all the problems we have explored in this thesis so far, as shown in Table 1.

Order. As mentioned above, the value $m + n$, or the combined number of index sets making up $\mathbf{I}$ and $\mathbf{J}$, is said to be the *order* of the 1-GNBP. For all but one of the problems explored in Sections 1.1 and 1.2, this is just 1 or 2, specifically with $m = 0$ and $n \in \{1, 2\}$ or $m = n = 1$. In such lower-order problems, we borrow from mapped functions the notational conventions of omitting or de-tupling components that are either unused or small. For instance, if $m = 0$ and $n = 2$, we might write

$$p_{0,2,\oplus,f}(J_1, J_2; X, Y) = \bigoplus_{j_1 \in J_1} \bigoplus_{j_2 \in J_2} f(x_{j_1}, x_{j_2}, y_{j_1}, y_{j_2}), \tag{12}$$

which becomes the 2-point correlation if $\oplus = +$ and $f(x_{j_1}, x_{j_2}, \cdot, \cdot) = [d(x_{j_1}, x_{j_2}) \leq h]$. Here, we use p instead of P because, without $\biguplus$, the result is not the usual set of

indexed families X and Y as representative of all involved datasets.

key-value pairs.[9] On the other hand, if $m = n = 1$, we might write

$$P_{1,1,\oplus,f}(I, J; X, Y) = \biguplus_{i \in I} \bigoplus_{j \in J} f(x_i, x_j, y_i, y_j), \tag{13}$$

replacing $\mathbf{I} = I_1$ with just I and $\mathbf{x_i} = x_{i_1}$ with just x_i, and likewise for $\mathbf{J}$ and $\mathbf{y}$. This becomes all-nearest-neighbors if we set $\oplus = \arg\min$ and $f(x_i, x_j, \cdot, \cdot) = d(x_i, x_j)$.

Chromaticity. A problem is said to be *monochromatic* if all of its index sets are the same—e.g. if $I = J$ or $J_1 = J_2$—and is said to be *bichromatic* or, more generally, *multichromatic* if these sets are disjoint.[10] In the monochromatic case, it may be necessary to adjust computation so it does not consider points grouped with themselves or so it only considers unique unordered groupings of points. As suggested above, we can achieve this by defining f to return identity $e_\oplus$ under certain conditions on $\mathbf{i}$ and $\mathbf{j}$. For example, monochromatic all-nearest-neighbors' $f(x_i, x_j)$ should return $e_{\min} = +\infty$ if $i = j$ to prevent points from identifying themselves as nearest neighbors.[11] Similarly, the 2-point-correlation's $f(x_{j_1}, x_{j_2})$ should return $e_+ = 0$ if $j_1 \geq j_2$, which restricts computation to the unique unordered pairs of distinct points.

1.4 Abstract algorithm derivation

The conventional, exhaustive solution approach for all 1-GNBPs follows immediately from their definition: visit all index combinations, evaluate f at each, and reduce as appropriate via iteration of $\oplus$. Assuming f and $\oplus$ to be constant time operations, such direct computation requires $O(N^{m+n})$ time if all index sets are themselves $O(N)$ in size. This approach may be tolerable for first-order problems, where $m + n = 1$, but—even admitting 1-GNBPs to be embarrassingly parallelizable—quickly becomes

[9] For $m = 0$, $\mathbf{I}$ is the set containing the empty tuple. Thus, if written, $\biguplus$ would visit a single (empty) $\mathbf{i}$ and package the result into a singleton set $P_{0,n,\oplus,f}(\mathbf{I}, \mathbf{J}; X, Y) = \{((), p_{0,n,\oplus,f}(\mathbf{J}; X, Y))\}$.

[10] The case where index sets are neither equal nor disjoint is not considered. There may, however, be interesting such cases, such as when one set is a subset of another.

[11] This is different from defining $f(x_i, x_j) = +\infty$ for $x_i = x_j$: distinct indices $i \neq j$ can nonetheless yield overlapping points $x_i = x_j$, and all-nearest-neighbors should be able to find these.

infeasible for all higher-order problems as N tends towards infinity.

In contrast to exhaustive computation, we have seen in Sections 1.1 and 1.2 that basically the same higher-order divide and conquer algorithm efficiently solves a host of 1-GNBPs, requiring only a minimum of adaptations to each specific problem. In this section, we develop an abstract algorithm that captures all of the nuances of Algorithms 1–7 and demonstrate it to be correct for all 1-GNBPs. We then explore the concept of *pruning*, by which our abstract algorithm derives all of its speedup, and enumerate several conditions that can make pruning possible.

1.4.1 Problem decomposition

The successful application of divide and conquer in any algorithm requires the ability to break larger problems into a handful of smaller problems of the same type. In the case of 1-GNBPs, properties of $\uplus$ and $\bigoplus$ enable convenient decompositions based on hierarchical partitionings—i.e. *trees*—built on the input datasets. Throughout the following, we consider subproblems $P_{m,n,\oplus,f}(\mathbf{I}', \mathbf{J}'; X, Y)$ for subsets $\mathbf{I}' \subseteq \mathbf{I}$ and $\mathbf{J}' \subseteq \mathbf{J}$ formed

$$\mathbf{I}' = I'_1 \times \cdots \times I'_m \quad \text{for} \quad I'_k \subseteq I_k,\ 1 \leq k \leq m \tag{14}$$

and similarly for $\mathbf{J}'$. Eventually, we will understand subsets I'_k of individual index sets I_k to represent nodes in trees built on each I_k; however, for our current purposes, any subsets formed as partitions of I_k will do.

Our first decomposition is a direct result of the disjoint union being defined in terms of the usual set union, itself a commutative, associative binary operation:

Lemma 1. Let *split* $\{\mathbf{A}, \mathbf{B}\}$ of $\mathbf{I}'$ be defined such that

$$\mathbf{A} = I'_1 \times \cdots \times A \times \cdots I'_m \quad \text{and} \quad \mathbf{B} = I'_1 \times \cdots \times B \times \cdots I'_m$$

for some partition $A \cup B = I'_k$, $A \cap B = \emptyset$, and $1 \leq k \leq m$. Then,

$$P_{m,n,\oplus,f}(\mathbf{I}', \mathbf{J}'; X, Y) = P_{m,n,\oplus,f}(\mathbf{A}, \mathbf{J}'; X, Y) \cup P_{m,n,\oplus,f}(\mathbf{B}, \mathbf{J}'; X, Y).$$

Proof. Direct consequence of Definitions 2 and 4 in conjunction with the observation that $\mathbf{A}$ and $\mathbf{B}$ partition $\mathbf{I}'$. □

Our second decomposition is very similar to the first, but focuses on $\oplus$ instead of the disjoint union. Before we can proceed, however, we must define a *vectorized* equivalent of $\oplus$ because the decomposition's operands will be sets of key-value pairs rather than elements of $\mathcal{P}$:

Definition 5. Given a binary operation of reduction $\oplus\colon \mathcal{P} \times \mathcal{P} \to \mathcal{P}$, we define *vectorized* operation $\oplus\colon 2^{\mathcal{I}^m \times \mathcal{P}} \times 2^{\mathcal{I}^m \times \mathcal{P}} \to 2^{\mathcal{I}^m \times \mathcal{P}}$ such that

$$P \oplus Q = \{(\mathbf{i}, p \oplus q) : (\mathbf{i}, p) \in P, (\mathbf{i}, q) \in Q\}.$$

Note that we use the same symbol for both vectorized and elemental $\oplus$ because it is always clear from context which of the two is intended. Observe that vectorized $\oplus$ is still both commutative and associative, and further that it distributes over $\cup$ because non-matching indices in either operand are ignored. We can now decompose:

Lemma 2. Let split $\{\mathbf{A}, \mathbf{B}\}$ of $\mathbf{J}'$ be defined such that

$$\mathbf{A} = J'_1 \times \cdots \times A \times \cdots J'_n \quad \text{and} \quad \mathbf{B} = J'_1 \times \cdots \times B \times \cdots J'_n$$

for some partition $A \cup B = J'_k$, $A \cap B = \emptyset$, and $1 \leq k \leq n$. Then,

$$P_{m,n,\oplus,f}(\mathbf{I}', \mathbf{J}'; X, Y) = P_{m,n,\oplus,f}(\mathbf{I}', \mathbf{A}; X, Y) \oplus P_{m,n,\oplus,f}(\mathbf{I}', \mathbf{B}; X, Y).$$

Proof. Consequence of Definitions 1, 2, 4, and 5 in conjunction with the observation that $\mathbf{A}$ and $\mathbf{B}$ partition $\mathbf{J}'$. □

Combined, these two decompositions form the recursive step of the GFMM:

Theorem 3. Consider an algorithm that solves a 1-GNBP $P_{m,n,\oplus,f}$ by recursively substituting

$$P_{m,n,\oplus,f}(\mathbf{I}', \mathbf{J}'; X, Y) \leftarrow \begin{cases} \biguplus_{\mathbf{i} \in \mathbf{I}'} \bigoplus_{\mathbf{j} \in \mathbf{J}'} f(\mathbf{x_i}, \mathbf{x_j}, \mathbf{y_i}, \mathbf{y_j}) & \text{if } \mathbf{I}', \mathbf{J}' \text{ are leaves,} \quad (3a) \\ P_{m,n,\oplus,f}(\mathbf{A}, \mathbf{J}'; X, Y) & \\ \quad \cup P_{m,n,\oplus,f}(\mathbf{B}, \mathbf{J}'; X, Y) & \text{if should split } \mathbf{I}', \quad (3b) \\ P_{m,n,\oplus,f}(\mathbf{I}', \mathbf{A}; X, Y) & \\ \quad \oplus P_{m,n,\oplus,f}(\mathbf{I}', \mathbf{B}; X, Y) & \text{otherwise,} \quad (3c) \end{cases}$$

where $\mathbf{A}$ and $\mathbf{B}$ are chosen in accordance with Lemmas 1 and 2 and whether to split $\mathbf{I}'$ or $\mathbf{J}'$ is decided arbitrarily so long as the splitting index set is not a leaf. Overall computation of $P_{m,n,\oplus,f}(\mathbf{I}, \mathbf{J}; X, Y)$ in this manner has the same result as direct computation.

Proof. The base case (line 3a) is the definition of a 1-GNBP and so is correct, and Lemmas 1 and 2 prove the decompositions in lines 3b and 3c to be correct if their subproblems are themselves correct. Because index sets are finite and thus capable of at most finitely many decompositions before reaching the base case, we reason by induction that all subproblems are solved correctly and thus, ultimately, that overall computation must also be correct. □

The above result does not by itself constitute a fast algorithm because, without pruning, computation will still visit all $O(N^{m+n})$ index combinations. Rather, this algorithm serves as a starting point for the derivation of the GFMM, which recurses similarly but also curtails work whenever it becomes provably unnecessary.

1.4.2 Subproblem summarization

Pruning is characterized by the substitution of subproblems $P_{m,n,\oplus,f}(\mathbf{I}', \mathbf{J}'; X, Y)$ with abbreviated, more easily computed results. Sometimes, these abbreviated results can

be exact, such as, for example, when $f(\mathbf{x_i}, \mathbf{x_j}, \mathbf{y_i}, \mathbf{y_j})$ is provably $e_\oplus$ for all $\mathbf{i} \in \mathbf{I}'$ and $\mathbf{j} \in \mathbf{J}'$. In other problems, however, pruning is only possible when results are approximated, and in yet other problems, pruning—either exact or approximate—requires the knowledge of bounds on results from other parts of computation. We encompass both of these concepts with a generalized definition of bounding, or *summarization*, which the GFMM employs to enable fast computation.

Summary results are derived from *cached sufficient statistics* computed at each subset (tree node) I'_k and J'_k. Such statistics always include a bounding structure, such as a box or ball, to represent the metric points x_{i_k} for $i_k \in I'_k$, but may also include other measures such as the points' mean and covariance as well as measures on nonmetric data y_{i_k}. We express these measures as $\sigma(I'_k; X, Y) \in \mathcal{S}$ and similarly for J'_k; we also define

$$\sigma(\mathbf{I}', \mathbf{J}'; X, Y) = (\sigma(I'_1; X, Y), \ldots, \sigma(I'_m; X, Y), \sigma(J'_1; X, Y), \ldots, \sigma(J'_n; X, Y)) \quad (16)$$

for aggregate index sets $\mathbf{I}'$ and $\mathbf{J}'$ as used throughout the above.

Definition 6. For a 1-GNBP $P_{m,n,\oplus,f}(\mathbf{I}', \mathbf{J}'; X, Y)$, a *summary result* is a set of potential results $\widehat{P}_{m,n,\oplus,f}(\mathbf{I}', \mathbf{J}'; X, Y)$ that satisfies the relation[12]

$$\widehat{P}_{m,n,\oplus,f}(\mathbf{I}', \mathbf{J}'; X, Y) \supseteq \{P_{m,n,\oplus,f}(\mathbf{I}', \mathbf{J}'; X', Y') : \sigma(\mathbf{I}', \mathbf{J}'; X', Y') = \sigma(\mathbf{I}', \mathbf{J}'; X, Y)\}.$$

Intuitively, $\widehat{P}_{m,n,\oplus,f}$ is the set of all possible results of $P_{m,n,\oplus,f}$ given what we know about X and Y in the ranges specified by $\mathbf{I}'$ and $\mathbf{J}'$. However, because the precise set of all possible results is often costly or impractical to represent, we allow $\widehat{P}_{m,n,\oplus,f}$ to be a (preferably tight) superset of this value. In most cases—i.e. when the domain of

[12] An alternate definition of summary results instead allows them to satisfy the stated relation with high probability, thereby enabling the computation of results that are *probably approximately correct*. This approach tends to achieve even greater speedup than methods with guaranteed error bounds [42, 35], though we will not discuss it further in this thesis.

partial results $\mathcal{P} = \mathbb{R}$—superset summary results may be expressed

$$\forall P \in \widehat{P}, (\mathbf{i}, p_{\mathbf{i}}) \in P \colon \mathtt{lo}_{\widehat{P}} \leq p_{\mathbf{i}} \leq \mathtt{hi}_{\widehat{P}} \tag{17}$$

which requires the storage of just two values $\mathtt{lo}_{\widehat{P}}$ and $\mathtt{hi}_{\widehat{P}}$. We typically compute such results by considering, for instance, the minimum and maximum values that f could possibly return for $\mathbf{x}$ and $\mathbf{y}$ as constrained by $\sigma(\mathbf{I}', \mathbf{J}'; X, Y)$.

In order to make use of summary results as prunes and as bounds on other parts of computation, we need the ability to compose the summaries of decomposite subproblems into aggregate summaries ultimately representative of overall computation. This process is basically the reverse of the problem decompositions described above, and is assisted by a pair of *summery composition operations* that we define presently:

Definition 7. Given a binary operation of reduction $\oplus \colon \mathcal{P} \times \mathcal{P} \to \mathcal{P}$, we define *summary composition* operation $\oplus \colon 2^{2^{\mathcal{I}^m \times \mathcal{P}}} \times 2^{2^{\mathcal{I}^m \times \mathcal{P}}} \to 2^{2^{\mathcal{I}^m \times \mathcal{P}}}$ such that

$$\widehat{P} \oplus \widehat{Q} \supseteq \{P \oplus Q : P \in \widehat{P}, Q \in \widehat{Q}\},$$

where $\widehat{P}$ and $\widehat{Q}$ are summary results and $P \oplus Q$ refers to the vectorized version of $\oplus$. We similarly define a summary composition operation $\uplus$ such that[13]

$$\widehat{P} \uplus \widehat{Q} \supseteq \{P \cup Q : P \in \widehat{P}, Q \in \widehat{Q}\}.$$

Like atomic summary results, composed summary results are sets of possible valuations of subproblems $P_{m,n,\oplus,f}(\mathbf{I}', \mathbf{J}'; X, Y)$, though they are derived from more precise understandings of X and Y given the cached sufficient statistics available from these problems' decomposed ranges of $\mathbf{I}'$ and $\mathbf{J}'$. We still permit superset relationships between composed summaries and the true set of all possible composed results in the

[13]Here, we use the symbol $\uplus$ instead of overloading $\cup$ because, unlike $\oplus$, $\cup$ can be confused as the usual set union when applied to summery result operands.

interest of feasible representation.

Again, when $\mathcal{P} = \mathbb{R}$, composed summaries are often given as in (17), and in fact a host of lemmas support the composition of bounded results for specific operations $\oplus$:

Lemma 4. Operations $\oplus = +$, min, and max may compose summary results by applying (elemental) $\oplus$ to the summaries' respective upper and lower bounds:

$$\big(\forall P{\in}\widehat{P}, (\mathbf{i}, p_\mathbf{i}){\in}P\colon \mathtt{lo}_{\widehat{P}} \le p_\mathbf{i} \le \mathtt{hi}_{\widehat{P}}\big), \big(\forall Q{\in}\widehat{Q}, (\mathbf{i}, q_\mathbf{i}){\in}Q\colon \mathtt{lo}_{\widehat{Q}} \le q_\mathbf{i} \le \mathtt{hi}_{\widehat{Q}}\big)$$
$$\implies \big(\forall P{\in}\widehat{P}{\oplus}\widehat{Q}, (\mathbf{i}, p_\mathbf{i}){\in}P\colon \mathtt{lo}_{\widehat{P}} \oplus \mathtt{lo}_{\widehat{Q}} \le p_\mathbf{i} \le \mathtt{hi}_{\widehat{P}} \oplus \mathtt{hi}_{\widehat{Q}}\big).$$

Lemma 5. Operation $\uplus$ may compose summaries by relaxing their bounds:

$$\big(\forall P{\in}\widehat{P}, (\mathbf{i}, p_\mathbf{i}){\in}P\colon \mathtt{lo}_{\widehat{P}} \le p_\mathbf{i} \le \mathtt{hi}_{\widehat{P}}\big), \big(\forall Q{\in}\widehat{Q}, (\mathbf{i}, q_\mathbf{i}){\in}Q\colon \mathtt{lo}_{\widehat{Q}} \le q_\mathbf{i} \le \mathtt{hi}_{\widehat{Q}}\big)$$
$$\implies \big(\forall P{\in}\widehat{P}{\uplus}\widehat{Q}, (\mathbf{i}, p_\mathbf{i}){\in}P\colon \min\{\mathtt{lo}_{\widehat{P}}, \mathtt{lo}_{\widehat{Q}}\} \le p_\mathbf{i} \le \max\{\mathtt{hi}_{\widehat{P}}, \mathtt{hi}_{\widehat{Q}}\}\big).$$

Proofs. Consequence of Definitions 5 and 7 and the properties of $+$, min, and max: Observe that, for all three of these operations, $a \oplus b \le c \oplus d$ if $a \le c$ and $b \le d$. Then, because $\mathtt{lo}_{\widehat{P}} \le p_\mathbf{i} \le \mathtt{hi}_{\widehat{P}}$ and $\mathtt{lo}_{\widehat{Q}} \le q_\mathbf{i} \le \mathtt{hi}_{\widehat{Q}}$ for all $\mathbf{i}$ and all $P \in \widehat{P}$ and $Q \in \widehat{Q}$, we judge that $\mathtt{lo}_{\widehat{P}} \oplus \mathtt{lo}_{\widehat{Q}} \le p_\mathbf{i} \oplus q_\mathbf{i} \le \mathtt{hi}_{\widehat{P}} \oplus \mathtt{hi}_{\widehat{Q}}$ for all possible combinations $P \oplus Q$.

Regarding $\uplus$, observe that $\min\{\mathtt{lo}_{\widehat{P}}, \mathtt{lo}_{\widehat{Q}}\} \le \mathtt{lo}_{\widehat{P}} \le p_\mathbf{i} \le \mathtt{hi}_{\widehat{P}} \le \max\{\mathtt{hi}_{\widehat{P}}, \mathtt{hi}_{\widehat{Q}}\}$ for all $P \in \widehat{P}$ and likewise for $q_\mathbf{i}$ and all $Q \in \widehat{Q}$ and thus also all possible $P \cup Q$. □

As a result of being defined in terms of the usual $\oplus$ and $\cup$, summary composition operations $\oplus$ and $\uplus$ are both commutative and associative. Also, similar to how vectorized $\oplus$ distributes over $\cup$, it can be shown that summary composition operation $\oplus$ distributes over $\uplus$.

Algorithm 8 A batch tree construction algorithm

```
    function BuildTree(I'_k, X, Y, leaf_size)
        if |I'_k| ≤ leaf_size then
3:          Compute σ(I'_k; X, Y) for all x_i, y_i, i ∈ I'_k
        else
            children(I'_k) ← ⟨some partition of I'_k⟩
6:          for all I''_k ∈ children(I'_k) do
                BuildTree(I''_k, X, Y, leaf_size)
            end for
9:          Compute σ(I'_k; X, Y) given all σ(I''_k; X, Y), I''_k ∈ children(I'_k)
        end if
    end function
```

1.4.3 The generalized fast multipole method

The GFMM is the natural combination of the recursive decomposition of Theorem 3 and the computation and maintenance of summary results. Subcomputations formed by decomposition are then pruned under a number of circumstances that can be expressed as conditions on the summary results.

At this point, it becomes important to think of subsets $I'_k \subseteq I_k$ as nodes in a tree built on each index set I_k. Trees may be constructed in advance and reused between different computations, and—depending on the kind of tree—may be constructed incrementally or in batch. The particular data structure used does not affect the correctness of the algorithm,[14] though it does play a key role in bounding its asymptotic running time, as discussed in Chapter 2. Tree construction is also a convenient time for the computation of cached sufficient statistics $\sigma(I'_k; X, Y)$, as shown in Algorithm 8. As such, we view tree construction as the GFMM's equivalent of the classical FMM's upward pass (FmmUpward in Algorithm 3).[15]

[14] Nodes' children are, however, generally required to partition their parents, i.e. the same index may not be repeated in multiple child nodes. This requirement is void in the special case that $\oplus$ is *idempotent* (having $a \oplus a = a$), such as when $\oplus = \min$ or $\max$.

[15] As suggested by Algorithm 8, we permit non-binary trees. While Lemmas 1 and 2 as well as Definition 7 expect binary decompositions, it is possible to extend these to the non-binary case simply by chaining them over multiple splits of the same index set.

Algorithm 9 The generalized fast multipole method

function GFMMEXPANSION($m, n, \oplus, f, \mathbf{I}, \mathbf{J}, X, Y$)
  Push $(\mathbf{I}, \mathbf{J})$ onto `heap`
  while `has_next`(`heap`) **do**
    Pop $(\mathbf{I}', \mathbf{J}')$ from `heap`
    if CANPRUNE($\widehat{P}_{m,n,\oplus,f}(\mathbf{I}', \mathbf{J}'; X, Y)$, COMPLIMENT($\mathbf{I}, \mathbf{I}'$)) **then**
      `postponed`($\mathbf{I}'$) $\leftarrow$ `postponed`($\mathbf{I}'$) $\oplus$ $\widehat{P}_{m,n,\oplus,f}(\mathbf{I}', \mathbf{J}'; X, Y)$
    else if `is_leaf`($\mathbf{I}'$) **and** `is_leaf`($\mathbf{J}'$) **then**
      for all $\mathbf{i} \in \mathbf{I}'$ **do**
        for all $\mathbf{j} \in \mathbf{J}'$ **do**
          $p_{\mathbf{i}} \leftarrow p_{\mathbf{i}} \oplus f(\mathbf{x}_{\mathbf{i}}, \mathbf{x}_{\mathbf{j}}, \mathbf{y}_{\mathbf{i}}, \mathbf{y}_{\mathbf{j}})$
        end for
      end for
    else
      for all $\mathbf{I}'' \in$ `children`($\mathbf{I}'$) **do**
        for all $\mathbf{J}'' \in$ `children`($\mathbf{J}'$) **do**
          Push $(\mathbf{I}'', \mathbf{J}'')$ onto `heap`
        end for
      end for
    end if
  end while
  GFMMPOSTPROCESS($\oplus, \mathbf{I}$)
end function
function GFMMPOSTPROCESS($\oplus, \mathbf{I}'$)
  if `is_leaf`($\mathbf{I}'$) **then**
    for all $\mathbf{i} \in \mathbf{I}'$ **do**
      $p_{\mathbf{i}} \leftarrow p_{\mathbf{i}} \oplus$ `postponed`($\mathbf{I}'$)
    end for
  else
    for all $\mathbf{I}'' \in$ `children`($\mathbf{I}'$) **do**
      `postponed`($\mathbf{I}''$) $\leftarrow$ `postponed`($\mathbf{I}''$) $\oplus$ `postponed`($\mathbf{I}'$)
      GFMMPOSTPROCESS($\oplus, \mathbf{I}''$)
    end for
  end if
end function

Algorithm 9's GfmmExpansion function corresponds to the FMM's side-to-side pass (FmmSideToSide). Unlike the classic FMM's side-to-side pass, however, GfmmExpansion does not enforce strict depth-first traversal; rather, it uses a heap to manage the order of subproblem decomposition. The heap priority function is algorithm specific and, like the choice of tree, does not impact correctness but often does affect the algorithm's run-time performance. In this thesis, we consider two broadly applicable heap priority functions, or *traversal patterns*: depth-first traversal, which implements the heap as a stack, and hybrid-breadth-first traversal, which implements the heap as a stack of queues. These two patterns are of particular interest both because they greatly simplify the maintenance of summary results and because they feature in bounding various algorithms' run-time complexities.

Pruning. The pivotal feature of GfmmExpansion that distinguishes it from the $O(N^{m+n})$ recursive decomposition of Theorem 3 is its call to CanPrune in line 5. This conditional test is the *heart* of fast computation: it must itself be reasonably quick, it must be true frequently enough to affect asymptotic running time, and yet it must only be true when summary result $\widehat{P}_{m,n,\oplus,f}(\mathbf{I}', \mathbf{J}'; X, Y)$ captures its represented subproblem well enough to yield provably tight bounds on the overall result. A general specification of CanPrune is given

$$\textsc{CanPrune}(\widehat{P}, \widehat{Q}) = \big(\exists \bar{P} \colon \forall P \in \widehat{P}, Q \in \widehat{Q} \colon \bar{P} \oplus Q \overset{R}{\approx} P \oplus Q\big), \tag{18}$$

which for *candidate prune* $\widehat{P}$ and *complimentary summary* $\widehat{Q}$ (to be defined below) ensures we can choose some $\bar{P}$ (usually in $\widehat{P}$) that gets us sufficiently close to the true result, no matter what the rest of computation yields. "Sufficiently close" is defined

by a relation R, which for example might bound absolute error[16]

$$\big(\bar{P} \overset{\text{abs}}{\approx} P\big) = \big(\forall(\mathbf{i}, \bar{p}) \in \bar{P}, (\mathbf{i}, p) \in P \colon |\bar{p} - p| < \epsilon\big), \tag{19}$$

or relative error

$$\big(\bar{P} \overset{\text{rel}}{\approx} P\big) = \big(\forall(\mathbf{i}, \bar{p}) \in \bar{P}, (\mathbf{i}, p) \in P \colon |\bar{p} - p| < \epsilon \cdot |p|\big). \tag{20}$$

Postprocessing. Similar to FmmDownward in Algorithm 3, the final stage of computation in Algorithm 9, GfmmPostprocess, recursively passes cached pruning information from each node to the leaves of the nonreduced indices, i.e. to each individual query. Because this ultimately writes $O(N^m)$ unique results—one for each $\mathbf{i} \in \mathbf{I}$—its asymptotic running time is itself no better than $O(N^m)$. Alternately, some algorithms may tolerate non-unique results found for ranges of nonreduced indices, thereby allowing GfmmPostprocess to shortcut and conceivably leading to asymptotic speedup. Regardless, postprocessing is almost always extremely fast compared to GfmmExpansion, especially considering the fact that m is often 1 or 0; hence, this thesis does not further consider the finalization of results.

1.4.4 Complimentary summarization

In the case that $\mathcal{P} = \mathbb{R}$, CanPrune is often a simple function of the values $\texttt{lo}_{\widehat{P}}$ and $\texttt{hi}_{\widehat{P}}$ attributed to candidate prune $\widehat{P}$, with no dependence upon any other part of computation. For example, guaranteeing absolute error for $\oplus = +$ is as easy as testing

$$\textsc{CanPrune}(\widehat{P}, \cdot) = \big(\tfrac{1}{2}(\texttt{hi}_{\widehat{P}} - \texttt{lo}_{\widehat{P}}) < \tfrac{|\mathbf{J}'|}{|\mathbf{J}|}\epsilon\big). \tag{21}$$

On the other hand, both relative error and operations min and max benefit from the knowledge of bounds on the results of the rest of computation: in the former case,

[16] In either of these cases, depending on $\oplus$, it may be appropriate to scale ϵ to reflect the size of the candidate prune's subset of reduced indices. For instance, if $\oplus = +$, $\frac{|\mathbf{J}'|}{|\mathbf{J}|}\epsilon$ guarantees at most ϵ error in the overall result.

Algorithm 10 A means of computing the complimentary summary of a target subproblem $\widehat{P}_{m,n,\oplus,f}(\mathbf{I}^*, \mathbf{J}^*; X, Y)$

```
    function COMPLIMENT(I', I*)
        let P̂ ← {∅}
 3:     if is_leaf(I') then
            for all i ∈ I' do
                P̂ ← P̂ ⊎ {{(i, p_i)}}
 6:         end for
        else
            for all I'' ∈ children(I'), I'' ∪ I* ≠ ∅ do
 9:             P̂ ← P̂ ⊎ COMPLIMENT(I'', I*)
            end for
        end if
12:     for all (I', J') ∈ heap do // Note: I' already bound!
            P̂ ← P̂ ⊕ P̂_{m,n,⊕,f}(I', J'; X, Y)
        end for
15:     return P̂ ⊕ postponed(I')
    end function
```

this provides a larger minimum value for $|p|$ on the right-hand side of (20), and in the later, it allows for the *exact* pruning of subcomputations bounded such that they are entirely greater than or less than the results of computation so far.

The general version of CANPRUNE embodies these bounds in its second argument, the complimentary summary $\widehat{Q}$ of $\widehat{P}_{m,n,\oplus,f}(\mathbf{I}', \mathbf{J}'; X, Y)$—that is, the composed summary result pertaining to all computation so far on nonreduced index subset $\mathbf{I}'$ and the compliment of $\mathbf{J}'$ in $\mathbf{J}$. Algorithm 10 provides a means of computing the complimentary summary that makes no assumptions about how `heap` is traversed or about the properties of $\oplus$. Its two arguments are a working set of nonreduced indices $\mathbf{I}'$—initially the nonreduced root $\mathbf{I}$—and a target range of nonreduced indices $\mathbf{I}^*$.[17] It works by first recursing to the leaves of $\mathbf{I}^*$ to form bounds on the exact results so far at each $p_{\mathbf{i}}$. Then, on the backward edge or recursion, it uses summary composition operation $\oplus$ to combine these bounds with bounds for any pending work stored in the heap as well as with any previous prunes stored in `postponed`$(\mathbf{I}')$. Because recursion

[17] Note the change of variables between COMPLIMENT's call in line 5 of Alg. 9 and its definition.

starts from nonreduced index root $\mathbf{I}$, and because all reduced index ranges $\mathbf{J}'$ paired with all nodes $\mathbf{I}' \cap \mathbf{I}^* \neq \emptyset$ are one of (a) solved exactly, (b) pruned, or (c) represented by exactly one $(\mathbf{I}'', \mathbf{J}'')$, $\mathbf{I}'' \supseteq \mathbf{I}'$, $\mathbf{J}'' \subseteq \mathbf{J}'$ on the heap,[18] the result of COMPLIMENT can be taken to bound all possible results of $P_{m,n,\oplus,f}(\mathbf{I}^*, \mathbf{J} - \mathbf{J}^*; X, Y)$ given what we have learned from the whole of computation so far.

This implementation of COMPLIMENT is unfortunately too expensive for practical use; however, various optimizations are possible for specific traversal patterns and for special cases of $\oplus$. In particular, for any $\oplus$ that has inverse elements for all $v \in \mathcal{P}$, we can maintain partial results for COMPLIMENT at each node $\mathbf{I}' \subseteq \mathbf{I}$ such that both the maintenance and finalization of complimentary summaries $\widehat{Q}$ require just $O(\log N)$ time per node visit, which happens to be the same as the worst-case cost of popping a node from the heap. More importantly, however, both the depth-first and hybrid-breadth-first traversal patterns allow for the computation of $\widehat{Q}$ in just constant time per node visit, regardless of the properties of $\oplus$. In any case, once we have obtained $\widehat{Q}$, we are free to test, for example,

$$\textsc{CanPrune}(\widehat{P}, \widehat{Q}) = \left(\tfrac{1}{2}(\mathtt{hi}_{\widehat{P}} - \mathtt{lo}_{\widehat{P}}) < \tfrac{|\mathbf{J}'|}{|\mathbf{J}|}\epsilon \cdot (\mathtt{lo}_{\widehat{P}} + \mathtt{lo}_{\widehat{Q}})\right), \tag{22}$$

for $\mathcal{P} = \mathbb{R}^+$, $\oplus = +$, and $R = \text{rel}$, or

$$\textsc{CanPrune}(\widehat{P}, \widehat{Q}) = \left(\mathtt{hi}_{\widehat{Q}} < \mathtt{lo}_{\widehat{P}}\right) \tag{23}$$

for $\oplus = \min$ and similarly (with subscripts $\widehat{P}$ and $\widehat{Q}$ transposed) for max.

1.4.5 Traversal patterns

The heap priority function, or traversal pattern, used in GFMMEXPANSION governs the order in which *node groupings* $(\mathbf{I}', \mathbf{J}')$ are discovered, visited, and ultimately dealt with. This may be a function of node groupings' cached sufficient statistics, of their summary results $\widehat{P}$, or even of the complimentary summary results $\widehat{Q}$ available at the

[18] Except, of course, for $(\mathbf{I}^*, \mathbf{J}^*)$, which has just been popped prior to the call of COMPLIMENT.

time of their formation. For instance, an algorithm might elect to visit first those node groupings with the greatest gap between $\texttt{hi}_{\widehat{P}}$ and $\texttt{lo}_{\widehat{P}}$ in an effort to tighten error bounds as quickly as possible; forced early termination would then tend to yield minimal error given the allotted computation time. Time-budgeted algorithms aside, however, empirical [65] and theoretical [6] results favor certain other traversal patterns that either minimize the overhead cost of expansion (depth-first traversal) or maximize the availability of information (hybrid-breadth-first traversal).

Depth-first traversal. Given that pruning is the GFMM's only source of asymptotic speedup and, further, that the choice of traversal pattern does not impact the correctness of computation, we judge that this algorithmic parameter should be selected so as to enable the most frequent and earliest pruning while yet mitigating its own operational costs. Therefore, in the special case that the traversal pattern has no impact on the GFMM's ability to prune—i.e., whenever CanPrune is strictly a function of its first argument $\widehat{P}$—we should choose the least expensive traversal pattern possible: depth-first recursion.

Depth-first recursion has essentially no overhead time cost per visited node grouping and requires at most $O(\log N)$ space when operating on balanced trees. Accordingly, even though we motivated the depth-first traversal pattern for the case that CanPrune ignores the complimentary summary result $\widehat{Q}$, its extreme speed and simplicity make it desirable even for problems where $\widehat{Q}$ is important. Algorithm 11 shows a means of performing depth-first traversal while maintaining complimentary summaries throughout computation. In reality, it computes $\widehat{Q}$ similarly to Algorithm 10, but using precomputed upper (`unvisited`) and lower (`cached`) portions that are kept available throughout computation. This is possible because the depth-first traversal pattern ensures that the "heap" is always very small and is updated in an extremely predictable manner. If, on the other hand, $\widehat{Q}$ is unused after all, then

Algorithm 11 A depth-first traversal algorithm for the GFMM

function GFMMDEPTH($\mathbf{I}', \mathbf{J}', \texttt{unvisited}$)
  if CANPRUNE($\widehat{P}_{m,n,\oplus,f}(\mathbf{I}', \mathbf{J}'; X, Y), \texttt{unvisited} \oplus \texttt{cached}(\mathbf{I}')$) **then**
    $\texttt{postponed}(\mathbf{I}') \leftarrow \texttt{postponed}(\mathbf{I}') \oplus \widehat{P}_{m,n,\oplus,f}(\mathbf{I}', \mathbf{J}'; X, Y)$
    $\texttt{cached}(\mathbf{I}') \leftarrow \texttt{cached}(\mathbf{I}') \oplus \widehat{P}_{m,n,\oplus,f}(\mathbf{I}', \mathbf{J}'; X, Y)$
  else if $\texttt{is\_leaf}(\mathbf{I}')$ **and** $\texttt{is\_leaf}(\mathbf{J}')$ **then**
    $\texttt{cached}(\mathbf{I}') \leftarrow \{\emptyset\}$
    for all $\mathbf{i} \in \mathbf{I}'$ **do**
      for all $\mathbf{j} \in \mathbf{J}'$ **do**
        $p_{\mathbf{i}} \leftarrow p_{\mathbf{i}} \oplus f(\mathbf{x}_{\mathbf{i}}, \mathbf{x}_{\mathbf{j}}, \mathbf{y}_{\mathbf{i}}, \mathbf{y}_{\mathbf{j}})$
      end for
      $\texttt{cached}(\mathbf{I}') \leftarrow \texttt{cached}(\mathbf{I}') \uplus \{\{(\mathbf{i}, p_{\mathbf{i}})\}\}$
    end for
    $\texttt{cached}(\mathbf{I}') \leftarrow \texttt{cached}(\mathbf{I}') \oplus \texttt{postponed}(\mathbf{I}')$
  else
    $\texttt{cached}(\mathbf{I}') \leftarrow \{\emptyset\}$
    for all $\mathbf{I}'' \in \texttt{children}(\mathbf{I}')$ **do**
      let $\{\mathbf{J}_{(1)}, \ldots, \mathbf{J}_{(k)}\} \leftarrow \texttt{children}(\mathbf{J}')$ **prioritized by** HEUR($\cdot$)
      let $\texttt{unvisited}_{(k)} \leftarrow \texttt{unvisited} \oplus \texttt{postponed}(\mathbf{I}')$
      for k' **from** $k-1$ **down to** 1 **do**
        $\texttt{unvisited}_{(k')} \leftarrow \texttt{unvisited}_{(k'+1)} \oplus \widehat{P}_{m,n,\oplus,f}(\mathbf{I}'', \mathbf{J}_{(k'+1)}; X, Y)$
      end for
      for k' **from** 1 **to** k **do**
        GFMMDEPTH($\mathbf{I}'', \mathbf{J}_{(k')}, \texttt{unvisited}_{(k')}$)
      end for
      $\texttt{cached}(\mathbf{I}') \leftarrow \texttt{cached}(\mathbf{I}') \uplus \texttt{cached}(\mathbf{I}'')$
    end for
    $\texttt{cached}(\mathbf{I}') \leftarrow \texttt{cached}(\mathbf{I}') \oplus \texttt{postponed}(\mathbf{I}')$
  end if
end function

all lines updating `unvisited` and `cached` may be omitted.

One other throwback to heap-based traversal is the use of HEUR, the old heap priority function, in line 17 to prioritize the order in which sibling node groupings are processed. This is important in, for example, nearest-neighbors search, which must be sure to explore nearer reference nodes first so as not to waste time traversing more distant subtrees that otherwise could have been pruned.

Hybrid-breadth-first traversal. The hybrid-breadth-first traversal pattern uses depth-first recursion when splitting nonreduced (query) nodes, but maintains lists of reduced (reference) nodes to be processed for each query node. Each recursive call then considers all reference nodes paired with the query node to see whether they prune; if so, their postponed pruning information is recorded for the query node as usual, but if not, their children are added to the next list of reference nodes to be processed. After all of the reference nodes have been considered, the algorithm recurses on the query children, each paired with the newly formed list of references. Please see Section 4.4.2 for a full exposition.

1.5 Affinity propagation

Affinity propagation [22] is a recent clustering technique that chooses *exemplars* from a data set $X \subset \mathcal{X}$ in attempt to maximize the sum of similarities between all points and their nearest exemplar. Two points x_i and x_j have similarity S_{ij}, and special case S_{ii} is set to parameter p, the *preference* of points to be exemplars. The number of clusters to find is not explicitly specified, but is positively correlated with p. A typical value of p is the median of the similarities between all pairs of points. The algorithm alternatingly updates message matrices R and A with

$$R_{ij} \leftarrow S_{ij} - \max_{j' \neq j}(A_{ij'} + S_{ij'}) \quad \text{and} \quad A_{ij} \leftarrow \Big(\sum_{i' \neq i} (R_{i'j})^{+}_{i' \neq j} \Big)^{-}_{i \neq j} \tag{24}$$

where $(v)^{+}_{\texttt{cond}}$ is $\max\{0, v\}$ for `cond` true and v otherwise, and likewise $(v)^{-}_{\texttt{cond}}$ for $\min\{0, v\}$. To stem oscillations, one customarily mixes computed values of R and A with their previous values according to a given damping factor λ. Upon convergence, exemplars are those points with $R_{ii} + A_{ii} > 0$.

For $|X| = N$, R and A are of size $N \times N$, but may be represented sparsely if $S_{ij} = -\infty$ for most combinations of i and j. On the other hand, we observe that *undamped* dense affinity propagation can be solved efficiently as a pair of second-order 1-GNBPs α and ρ if we rearrange computation

$$R_{ij} \leftarrow S_{ij} + \alpha_{i[j]}, \qquad \alpha_{i[j]} \leftarrow \min_{j' \neq j}(-A_{ij'} - S_{ij'}), \tag{25}$$

$$A_{ij} \leftarrow (\rho_j - (R_{ij})^{+}_{i \neq j})^{-}_{i \neq j}, \qquad \rho_j \leftarrow \sum_{i'}(R_{i'j})^{+}_{i' \neq j}. \tag{26}$$

Because $\alpha_{i[j]}$ depends on j only for exclusion from min, we achieve its behavior by finding the first two minima, returning the second if j is the index of the first. We substitute further to obtain

$$\alpha_{i[\cdot]} \leftarrow \min_{j}{}_2(((S_{ij} + \alpha_{i[j]})^{+}_{i \neq j} - \rho_j)^{+}_{i \neq j} - S_{ij}) \quad \text{and} \quad \rho_j \leftarrow \sum_{i}(S_{ij} + \alpha_{i[j]})^{+}_{i \neq j}. \tag{27}$$

When $\mathcal{X}$ is a metric space and similarity is a monotonically decreasing function of distance, both α and ρ become 1-GNBPs as per Definition 4. Bounding boxes or balls in $\mathcal{X}$ can then yield bounds on similarity, which—when combined with bounds kept for the previous results of α and ρ—can in turn produce summary results $\widehat{\alpha}$ and $\widehat{\rho}$.

1.5.1 Fast algorithms for α and ρ

Algorithm 12 details an algorithm for α that prunes when a subcomputation's results are bounded such that they are entirely greater than the greatest second minimum found so far. As computation depends upon the old value of α, we refer to the new result as α'. Bounds on both values are given, e.g., $\alpha_{\text{lo}}(I)$ for the previous least first minimum over index range I and $\alpha'_{\text{hi}}(I)$ for the current greatest second minimum.

Algorithm 12 A algorithm to compute α

```
    function ALPHA(I', J')
        if α'_hi(I') < α_lo(I') − ρ_hi(J') then return
 3:     else if is_leaf(I') and is_leaf(J') then
            for all i ∈ I' do
                for all j ∈ J' do
 6:                 α'_i[·] = min_2{α'_i[·], ((S_ij + α_i[j])^+_{i≠j} − ρ_j)^+_{i≠j} − S_ij}
                end for
            end for
 9:         α'_hi(I') = max{α'_i[·] : i ∈ I'}
        else
            for all I'' ∈ children(I) do
12:             for all J'' ∈ children(J) prioritized by −ρ_hi(·) do
                    ALPHA(I'', J'')
                end for
15:         end for
            α'_hi(I') = max{α'_hi(I'') : I'' ∈ children(I')}
        end if
18: end function
```

Mapped function $((S_{ij}+\alpha_{i[j]})^+_{i\neq j}-\rho_j)^+_{i\neq j}-S_{ij}$ is bounded below by $\alpha_{\text{lo}}(I)-\rho_{\text{hi}}(J)$. Similar to (23), if $\alpha'_{\text{hi}}(I) < \alpha_{\text{lo}}(I) - \rho_{\text{hi}}(J)$, then the present subcomputation cannot affect results and may thus be pruned. Traversal must then maintain α'_{hi}, a portion of the complimentary result, and work order should be prioritized by least lower-bound contribution. Note Algorithm 12's particular similarity to all-nearest-neighbors in Algorithm 5 despite this problem's considerably more complicated mapped function. This similarity arises from the algorithms' common choice of traversal pattern (depth-first) and from their use of the same pruning strategy for $\oplus = \min$.

Algorithm 13 details an algorithm for ρ that prunes whenever all of a subcomputation's results are exactly 0. We rename bounds $\alpha'_{\text{hi}}(I)$ computed in Algorithm 12 to $\alpha_{\text{hi}}(I)$ and, to simplify bounds, we initialized $\rho_j \leftarrow (S_{jj}+\alpha_{j[j]})^-_{\text{true}}$ instead of $e_+ = 0$ and update with $\rho_j \leftarrow \rho_j + \sum_i (S_{ij}+\alpha_{i[j]})^+_{\text{true}}$.

Observe then that, if $S_{\text{hi}}(I,J)+\alpha_{\text{hi}}(I) \leq 0$, all pair-wise computations must result in the identity and thus the represented subproblem may be pruned. This prune does not depend upon the complimentary summary, so there is no need to maintain

Algorithm 13 A algorithm to compute ρ

```
    init ∀j∈J: ρ_j ← (S_jj + α_j[j])⁻_true
    function RHO(I′, J′)
3:      if S_hi(I′, J′) + α_hi(I′) ≤ 0 then return
        else if is_leaf(I′) and is_leaf(J′) then
            for all i ∈ I′ do
6:              for all j ∈ J′ do
                    ρ_j ← ρ_j + (S_ij + α_i[j])⁺_true
                end for
9:          end for
        else
            for all I″ ∈ children(I) do
12:             for all J″ ∈ children(J) do
                    RHO(I″, J″)
                end for
15:         end for
        end if
    end function
```

bounds on intermediate results, nor is there need to prioritize the order subproblem expansion. The resulting algorithm is then similar to all-pairs KDE in Algorithm 6; however, because pruning occurs only when + is a no-op, results are exact and no postprocessing is required.

Convergence. Performed in alteration, the above update procedures compute exactly the results on undamped affinity propagation. Unfortunately, without damping, implicit matrices A and R tend not to converge; we remedy this problem by damping just ρ. The relationship between this method and standard damping requires further investigation, but our experiments yield comparable results in similarly many iterations.

1.5.2 Experimental results

We implemented algorithms ALPHA and RHO in C++ and directly compare them to Frey and Dueck's quadratic C implementation. Our dataset is a collection of points in $\mathbb{R}^3$ generated by a large-scale gravitational N-body particle simulation. We use

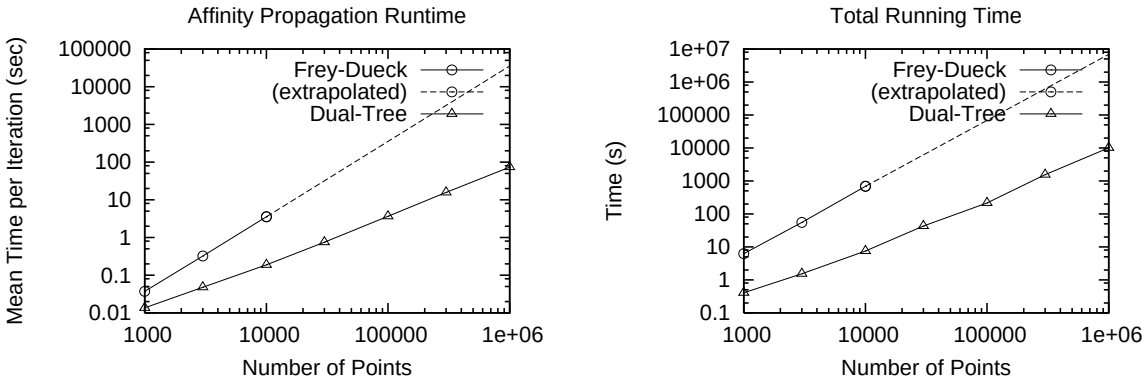

Figure 1: Mean per-iteration (left) and overall (right) running times for affinity propagation. Although the Frey-Dueck implementation runs out of memory after 10,000 points, we extrapolate their algorithm according to its quadratic run-time complexity. Settings: gcc 3.4.6 on a NetBurst-class Intel Xeon 3.0GHz with 8GB RAM running Linux 2.6.9.

the negative Euclidean distance as our similarity function and set preference p to the median pair distance, as is typical.[19] Figure 1 demonstrates asymptotic speedup both per iteration and overall. Empirically, per-iteration running time scales proportional to $N^{1.3}$ and, when we factor in the number of iterations required for convergence, we observe the overall running time to be $O(N^{1.5})$. The quadratic algorithm unfortunately does not support large enough datasets to observe its trend in the required number of iterations; regardless, if we assume its running time to grow exactly proportional to N^2, our results suggest an extrapolated three-hundred-fold speedup for datasets as large as one million points.

1.6 Future work: Beyond 1-GNBPs

While 1-GNBPs can already model many diverse and important computations, there are yet other problems we would like to include in the full class of GNBPs that we

[19]The median pair distance is itself found via bisection of the of the 2-point correlation, further demonstrating the broad applicability of 1-GNBPs.

cannot immediately express in terms of the above. For example, all-pairs Nadaraya-Watson regression [53, 77] is written

$$\forall i \in I : \widehat{y}(x_i) = \frac{\sum_{j \in J} K_h(x_i, x_j) y_j}{\sum_{j \in J} K_h(x_i, x_j)}, \tag{28}$$

which for each x_i predicts target y by way of conditional expectation in conjunction with the kernel density estimate. Observe that, even though we cannot model this problem in its entirety as a single 1-GNBP, we *can* model its component numerators and denominators as separate 1-GNBPs $P_{1,1,+,K_h(x_i,x_j)y_j}$ and $P_{1,1,+,K_h(x_i,x_j)}$. Other examples of non-1-GNBPs include problems with more than one kind of reduction, such as finding the medoid of a group of points [70]

$$p = \arg\min_{i \in I} \sum_{j \in I} d(x_i, x_j), \tag{29}$$

as well as problems with functions between repetitions of the same reduction, such as computing the log-likelihood of a mixture of Gaussians

$$\log \mathcal{L} = \sum_{i \in I} \log \sum_{1 \le j \le k} \pi_j \phi(x_i; \mu_j, \sigma_j). \tag{30}$$

Similar to the above, we can model both of these if we treat their inner reductions as separate 1-GNBPs—in this case $P_{1,1,+,d(x_i,x_j)}$ and $P_{1,1,+,\pi_j\phi(x_i;\mu_j,\sigma_j)}$—and then finalize computation via subsequent reduction over the results. Hence, we define:

Definition 8. A *generalized N-body problem (GNBP)* is a function of 1-GNBPs whose mapped functions may themselves be GNBPs.[20]

We are forced to distinguish fully general GNBPs from their simpler counterparts, the 1-GNBPs, because many of the theorems we have presented above do not as

[20] When the results of multiple 1-GNBPs occur together in the same function, they are said to be arranged *in parallel*. Alternately, when one 1-GNBP occurs inside another 1-GNBP's mapped function, the two are said to be arranged *in series*. In this later case, we understand the nested 1-GNBP to inherit the containing 1-GNBP's index sets as nonreduced index sets, except for those sets whose indices do not feature at all in the nested 1-GNBP. The overall order of a GNBP is then the greatest order of any involved 1-GNBP.

easily apply to the full problem class. Accordingly, except for kernel discriminant analysis, described thoroughly in Chapter 4, this thesis does not further consider GNBPs beyond the scope of 1-GNBPs. Nevertheless, asymptotically fast algorithms are possible for full GNBPs as well; such algorithms can be said to work by solving all component 1-GNBPs simultaneously.

CHAPTER II

ASYMPTOTIC RUNNING TIMES OF THE GFMM

While the algorithmic strategy described above has been shown to be fast in numerous empirical studies, analytical bounds on its running time offer even more convincing evidence of the method's inherent scalability. Alas, not every computation that can be expressed as a GNBP can be proven to have a favorable run-time complexity, and indeed some problems will fail to attain any kind of speedup due to their inability to prune. However, in problems where a certain kind of aggressive pruning is possible, computation can be shown to take only $O(\log N)$ or $O(N)$ time.

2.1 Related work

A number of past papers have included proofs of bounded running time for special cases of the GFMM. Perhaps the most studied of these are nearest-neighbors search and, in particular, the all-nearest-neighbors problem. Early theoretical results for this problem include publications by Shamos [67], Dobkin and Lipton [15], and Bentley [5]; however, it was Vaidya [73] who first proved all-nearest-neighbors to be solvable in (optimal) $O(N \log N)$ time even in general-dimensional Euclidean space. Subsequent work has generalized Vaidya's result [10] and has also provided new algorithms for all-nearest-neighbors in non-Euclidean spaces [13, 38, 39, 40, 6].[1]

Another well-studied GNBP with multiple published theoretical run-time bounds is the classical N-body simulation problem in physics, as thoroughly discussed in Section 1.1. For uniformly distributed two- and three-dimensional data, the Barnes-Hut algorithm can be shown to be $O(\log N)$ at a single point [4], while Appel's

[1] An excellent review of modern nearest-neighbors search methods is available in [14].

algorithm and the FMM are $O(N)$ overall [19, 30]. It should be noted, however, that uniform data distributions cannot be expected even in gravitational particle simulations, and yet these algorithms continue to perform well for non-uniformly distributed datasets. This suggests that an even more general run-time bound exists for these algorithms, and in fact this paper demonstrates such a bound based on the data's expansion constant instead of its distribution.

The first method to unify both nearest-neighbors search and N-body simulation is the well-separated pair decomposition (WSPD) [10], which observes that both problems are efficiently solved by decomposing work into ranges of queries and references paired with one another, not unlike the decompositions described in Section 1.4. Callahan and Kosaraju prove a generalization of Bentley's k-d trees to be constructable in $O(DN \log N)$ time for dimensionality D—an important result in its own right—and further show that both of the above problems need only visit $O(O(\sqrt{D})^D N)$ subproblems in order to compute their result. While the WSPD's treatment of kernel summation is largely the same as Appel's algorithm, its handling of exact all-nearest-neighbors algorithm tends towards theory and away from practice: After (fast) linear-time decomposition, the algorithm must repeatedly partition and refine size $O(O(1)^D)$ sets of neighbor-seeking points, once per node, in a process that necessarily involves rampant shuffling of points or indices.

To date, the most practical exact nearest-neighbors search algorithm that nonetheless offers favorable theoretical run-time bounds is the *cover tree algorithm* developed by Beygelzimer, Kakade, and Langford [6]. This algorithm hinges upon its specialized search index, the *cover tree*, which is linear in size and can be built in $O(c^6 N \log N)$ time, where *expansion constant* c is a property of the dataset related to its intrinsic dimensionality. Beygelzimer et al. prove that breadth-first traversal on this structure solves nearest-neighbors search at a single query in $O(c^{12} \log N)$ time and monochromatic all-nearest-neighbors in $O(c^{16} N)$ time. Ram [62] revisits these proofs to extend

them to the bichromatic case as well as to a family of kernel summations. We note, however, that while the proofs in both of these papers serve general-dimensional as well as non-Euclidean data, they make certain assumptions about recursive behavior that ultimately limit their applicability.[2]

The theorems and proofs contained in this chapter are strongly influenced by [6] and [62], but they avoid their predecessors' assumptions while yielding even tighter bounds on asymptotic run-time. Accordingly, this paper not only represents the first general proof strategy for bounding the running time of single- and multi-tree methods, but also the first assumption-free proofs to bound the running times of all-nearest-neighbors and kernel summation.

2.2 A general *general-dimensional bound*

The general strategy for bounding the running time of the GFMM starts by observing that work is only performed when some node (or more than one node) in a node grouping splits. Thus, if we count the number of times any node can split, we can bound the overall running time of the algorithm. We obtain this value by bounding the size of any given node's *cover set*, or the set of node groupings that the given node can be member of. Because there are are $O(N)$ total nodes, if the cover set's size is bounded by a constant, then the algorithm performs at most $O(N)$ work overall.

We define $\mathcal{C}(v)$ to be a data structure's *containment bound*, or specifically the maximum number of disjoint nodes with radius at least r that fit completely inside a sphere of radius vr for any $r > 0$. Then, using the above methodology, we claim that 1-GNPBs of order 2 have run-time complexity

$$O(\mathcal{C}(2(s+3))N),$$

where s is the *separation constant* from WSPD. The $2(s+3)$ is derived as follows: For

[2]For further discussion of these assumptions, please see Appendix A.

pruning radius $r_{\text{prune}} = sr$, all unpruned nodes fit entirely within a sphere of radius

$$r \text{ (radius of first node)} + r_{\text{prune}} + 2r \text{ (diameter of second node)}$$

with center equal to the center of the first node. The factor of 2 comes from looking for children, assumed to have (outer) radius no less than $r/2$.

Proof. Choose a traversal pattern that splits all nodes of larger radii first: A splitting node can only be paired with nodes that have the same (outer) radii as itself. Only $\mathcal{C}(s+3)$ such nodes can exist, but some of them might split at the same time. Regardless, this set can have at most $\mathcal{C}(2(s+3))$ total children. Thus, each of the $O(N)$ nodes can only ever be introduced in at most $\mathcal{C}(2(s+3))$ node-pairs as a result of its parent node splitting. Assuming $O(1)$ cost of split, prune, and base-case, the algorithm's overall run-time complexity is thus

$$O(\mathcal{C}(2(s+3))N).$$

□

For a particular algorithm, it may also be necessary to prove that splits, prunes, and base-cases are $O(1)$. Also, using the hybrid breadth-first requires heap maintenance of per-node cost

$$\mathcal{C}(s+3)\log\mathcal{C}(s+3),$$

however, as we shall see for the all-nearest-neighbors algorithm, the cost of heap maintenance may be occluded by other work.

2.2.1 Deriving $\mathcal{C}$

Containment is data-structure dependent. For cover trees it is derived as follows: The expansion constant is a measure of intrinsic dimension. It is defined for $X \subseteq \mathcal{X}$ as the least value c such that

$$|B_X(p, 2r)| \leq c \cdot |B_X(p, r)|$$

for all $p \in \mathcal{X}$ and $r > 0$. Cover trees model "infinite" trees composed of spherical nodes at all integer depths i. They have several important properties, but notably among these are:

- Nodes at depth i have radius at most 2^{i+1}.
- Node centers at depth i are at least 2^i apart.

[6] shows how to build these as well as other properties of the cover tree.

Balls of radius 2^{i-1} centered at nodes at depth i are disjoint. $\lceil \log_2 v \rceil + 3$ doublings of this radius is enough for each ball in a radius of $v2^{i+1}$ to cover all the balls. There can thus be at most $\mathcal{C}(v) = c^{\lceil \log_2 v \rceil + 3}$ such balls.

It is also possible to derive $\mathcal{C}$ for quadtrees and octrees. In this case, it can be shown that $\mathcal{C}(v) = (v\sqrt{D})^D$ for dimensionality D; when $D > 3$, we call this data structure a *grid tree*. While this bound is indeed large for large D, at smaller D—say, up to 4—it is often competitive with the equivalent bound for cover trees. In fact, while c can depend on the distribution of the data, D does not, and thus derived run-time bounds pertain to arbitrarily distributed data. On the other hand, it is tempting to think that knowledge of c in high-dimensional Euclidean space can provide a tighter run-time bound for grid trees; however, this does not appear to be the case.[3]

The remainder of this chapter assumes the data structure in use is a cover tree.

2.3 Nearest-neighbors search

The first step in bounding the running time of all-nearest-neighbors is bounding the maximum size of its pruning radius, which is related to its separation constant s:

- At depth i, if the query center's candidate nearest neighbor distance d is more than 2^{i+2}, then there can be no reference points nearer than $d/2 < d - 2^{i+1}$.

[3] One notable issue is the fact that grid trees can have 2^D children per node even for datasets with low c, as shown in Appendix B.

- The pruning distance is $d' = d + 2^{i+2} + 2^{i+1}$, where the minimum distance to the query node ($d' - 2^{i+2}$) becomes greater than the maximum distance between the query node and its candidate nearest neighbor ($d + 2^{i+1}$).
- All descendant points of the cover set are thus contained within $d' + 2^{i+1} = d + 2^{i+3}$. This is bigger than the single-tree case.
- Three doublings of $d/2$ are sufficient to cover everything, so there can be no more than c^3 points left to consider.

The base-case constant is thus slightly larger than that of the single-tree case as shown in [6], but this does not play into the final run-time bound.

Any query node with $d > 2^{i+2}$ can shortcut to exhaustive computation, and this will never incur more than $O(c^3 N)$ overall. On the other hand, for query nodes with $d \leq 2^{i+2}$:

- The pruning distance is $d' = 2^{i+3} + 2^{i+1}$, which is again where the lower-bound distance to the query node ($d' - 2^{i+2}$) becomes greater than $d + 2^{i+1}$.
- Thus all descendant points are within $2^{i+3} + 2^{i+2}$, which has diameter $2^{i+4} + 2^{i+3} < 2^{i+5}$.
- Each child has minimum distance 2^{i-1} between their centers, and thus radius 2^{i-2} balls centered on the children are disjoint. Thus, in 7 doublings of this disjoint radius, the entire pruning radius is covered by a ball centered at any child node within it.

This means the run-time coefficient will be c^7 for the size of the cover set.

The above was reasoned specifically for the number of reference nodes that can be paired with a given query node. More specifically, this is how many reference children can exist beneath a query node's cover set. We must now look into how much work is incurred by reference nodes that must split even when query nodes do not. Because

each reference node can only split once per cover set containing it, an important part of this bound is bounding how many cover sets can contain it. Further, I can avoid a factor of c^4 (to produce the reference node's children) if I instead count how many cover sets can contain a reference node's *parent*.

Whatever shallower depth the parent node is at, the number of cover sets it can be in is bounded the same as for the node itself. Also, we are only interested in cover sets containing parent nodes that do not immediately prune; pruning does not incur any cost to create children and the cost of pruning has already been paid in an amortized sense.

The reference parent cannot be paired with any (explicit) query nodes with shallower implicit depth than it because they would have already split. Also, while paired query nodes can be implicitly deeper, *their* parents cannot be, because *they* would not have split yet. This means there is a one-to-one correspondence between the set of explicit query nodes actually paired with the reference parent and the set of implicit query nodes at the same depth as the reference parent that would have been paired with it in the implicit algorithm. We are thus able bound the number of interested query nodes as if they all had the same depth as the reference parent.

If the reference parent is at depth i, then each cover set has pruning radius at most $2^{i+3} + 2^{i+1}$ (unless computation is about to terminate). Accordingly, only query nodes centered within $2^{i+3} + 2^{i+1}$ of the reference center can still include the reference parent in their cover sets. Query nodes at depth i are at least 2^i apart from one another, and thus balls of radius 2^{i-1} centered on the query nodes must be disjoint. All descendant *query* points are within $2^{i+3} + 2^{i+2}$ of the reference parent, and the diameter of that region is less than 2^{i+5}. Thus, in just 6 doublings we count all the query points. That means that at most c^6 nonterminal query nodes can include the reference parent in their cover sets, where c is properly the expansion constant of the query set, rather than the references. Rather than distinguish query and reference

expansion constants, however, we will assume that c is set to the maximum of the two.

The amount of much work done per reference node per cover set it is born into is $\log c$: Each created node enqueues itself once and later dequeues itself once, and the queue will only ever grow to size c^6 (because immediately pruned children will not ever be pushed, which leads to a maximum of c^6 parents plus c^6 unpruned children). Note that the $c^6 \log c$ cost of pushing all unpruned children is occluded by the c^7 cost of visiting all child nodes of the cover set.

It is important that the size of the heap actually does remain bounded by a constant, as suggested above. If permitted to grow to size $O(N)$, cost of heap operation becomes $O(\log N)$, which ultimately introduces a $\log N$ term in the final run-time complexity. A case in which the heap's size can become large is if, for instance, *leaf* nodes accumulate in the heap that ordinarily would prune but never get popped because they are infinitely deep. It is thus key to eliminate nodes that can prune as soon as possible. Query splits already do this without additional effort, but the reference heap needs help. Accordingly, we introduce a second heap, keyed by distance to the query node center, and maintained alongside the depth heap. Each heap's values store pointers into the other heap. Whenever a node pivots in one heap, it updates the other heap's return pointer (which the node itself conveniently points to). Recursive calls thus check to see if the distance heap suggests any nodes that can immediately prune and, if so, removes said nodes from both heaps. Once no prunable nodes remain, they split the shallowest nodes in the depth heap, again updating the distance heap to reflect. Query splits, on the other hand, always rebuild both heaps: distances will completely change and, thus, so will pointers into the distance heap. All heap maintenance costs an amortized $\log c$ per pop, since heap size is now truly bounded as $O(c^6)$.

For N_q the size of the queries and N the size of the references, and c_q the query expansion constant and c that of the references, we have $O(c^3 N)$ work for terminal nodes, $O((c^7 + c^6 \log c) N_q)$ work for the reference children seen and pushed at each query split, and $O(c_q^6 \log cN)$ work for reference splits in all the cover sets that can contain them. The $\log c$ terms also amortize for when the node may ultimately prune independent of a split. This is, overall, $O(c^7 N_q + c_q^6 \log cN)$, or just $O(c^7 N)$ if we define c and N as the larger respective value of either the queries or references.

2.3.1 The monochromatic case

There is a simple proof to show that the monochromatic case behaves no worse than the above. Recall that the monochromatic case is when the query set and reference set are the same, and has the special case behavior that queries must not identify themselves as nearest neighbors. At first it seems that pruning will be more complicated than the bichromatic case, but it is actually only barely different.

Consider when a query node contains more than one point, i.e., when it is not a leaf. The query node will always be a member of the cover set because its lower-bound distance is zero (even though queries cannot match themselves, other points in the same node may be arbitrarily close). This node still yields an upper-bound nearest-neighbor distance of 2^{i+1}, because all points other than the center are within 2^{i+1} of the center and vice versa, and—by assumption—there are points other than the center. Thus, pruning is not adversely affected.[4]

Now consider when the query node is a leaf. Its counterpart in the cover set is thus also a leaf, and we can eliminate it so as to avoid matching itself. Pruning and termination then behave exactly the same as in the bichromatic case: either another node is within the threshold distance d or computation is about to terminate. If a node is within the threshold distance, we have the same bound on the pruning radius,

[4]The self-entry in the cover set actually yields an upper-bound distance of only 2^i, because all immediate children are within 2^i of the center, and all other points are within 2^i of them.

and thus the number of cover sets per reference node is bounded the same, and thus overall run time is bounded the same.[5]

2.4 Kernel density estimation

Assume a kernel function satisfies the following on interval $[0, +\infty)$:

- It is continuous, differentiable, nonnegative, and nonincreasing.
- Its result goes to 0 as its input goes to +Inf.
- It is composed of a concave section followed by a convex section, with inflection point h.

The above covers the Gaussian kernel most directly. This yields:

- The second derivative of the concave section is nonpositive.
- The second derivative of the convex section is nonnegative.
- The first derivative is nonpositive.
- The first derivative at h is the most extreme.

In order to bound the run time, it is at first necessary to bound the number of reference nodes that can exist in a cover set. In the case of absolute error in an average, we have ϵ allowable error per reference point. Thus, an entire node is pruned if the difference between the minimum and maximum kernel contribution is within ϵ. Every node in the cover set must then have a greater such difference than ϵ.

[5]Actually, because of the tighter upper-bound distance mentioned in the previous footnote, the number of nodes that can inhabit the cover set of a nonleaf node is just c^5, with a maximum of c^6 children. Leaf nodes, on the other hand, prune just like the single-tree case, with again yields these same bounds. Thus, we may eliminate precisely one factor of c from the bound for the monochromatic case: $O(c^6 N)$.

Nodes at distance $d > h + 2^{i+1}$ are entirely within the convex section of the kernel. Their delta is such that

$$\epsilon < K(d - 2^{i+1}) - K(d + 2^{i+1}) < -2^{i+2}K'(d - 2^{i+1}). \tag{31}$$

Nodes at distance $d < h - 2^{i+1}$ are entirely within the concave section of the kernel and have delta such that

$$\epsilon < K(d - 2^{i+1}) - K(d + 2^{i+1}) < -2^{i+2}K'(d + 2^{i+1}). \tag{32}$$

Nodes with d in $[h - 2^{i+1}, h + 2^{i+1}]$ overlap with the steepest slope and so have delta bounded

$$\epsilon < -2^{i+2}K'(h).$$

From this last statement, [62] derives that

$$i > lg(-\epsilon/K'(h)) - 2 = i_1.$$

Further, it shows

$$-\epsilon/2^{i+2} > K'(d + / - 2^{i+1}) > K'(h),$$

keeping mind that the derivatives are all negative. From here, we can again derive

$$i > lg(-\epsilon/K'(h)) - 2.$$

We can thus prune everything outside $K^{-1}(\epsilon) + 2^{i+1}$. The number of children is bounded by c to the number of doublings it takes for a 2^{i-2} ball centered on any of them to subsume the cover set, which has width $2K^{-1}(\epsilon) + 2^{i+3}$:

- If $K^{-1}(\epsilon) \leq 2^{i+2}$, width is less than 2^{i+4} for c^6.
- If $K^{-1}(\epsilon) > 2^{i+2}$, width is less than $4K^{-1}(\epsilon)$ for $c^{\lg(K^{-1}(\epsilon))-i+4}$.

Because i must be shallower than i_1, we have

$$c^{\lceil \lg(-K'(h)K^{-1}(\epsilon)/\epsilon) \rceil + 6}$$

[62] visits a few additional cases, but the above is now sufficient to bound the overall result. This is the maximum number of children a cover set can have at any depth, and thus $O(c^{\lceil \lg(-K'(h)K^{-1}(\epsilon)/\epsilon)\rceil+6}N)$ bounds the overall running time.

2.5 The N-body problem

For other kinds of kernels, we can tolerate relative error on the distances between points rather than on the resulting kernel sum. This permits more aggressive pruning at every depth, regardless of h or $K^{-1}(\epsilon)$, though error bounds on the result may ultimately be slightly looser.

The idea is to approximate a node with, say, its midpoint whenever node radius becomes less than epsilon times the lower-bound distance. Thus, at depth i, the pruning radius is $2^{i+1}/\epsilon$ and a cover set can have no more than $c^{4-\lfloor \lg \epsilon \rfloor}$ nodes, provided that epsilon is at most $1/2$. Because pruning is not based upon the kernel at all, this bound holds for all kernel sums. Using strict depth-first traversal, dual-tree running time is bounded

$$O(c^{\lceil \lg(1/\epsilon)\rceil+5}N),$$

which is a considerable improvement over the above.

For the relative error in distance method, it is possible to show that the $1/r$ kernel has relative error in its result:

- Kernels are evaluated at d' in the range $[d(1 \pm \epsilon)]$.
- Relative error on each kernel value is thus $|1 - d/d'|$, which is bounded above by $1/(1-\epsilon) - 1$.
- Because relative error propagates through sums, the final result has relative error at most $1/(1-\epsilon) - 1$ as well.

If you instead want to specify ϵ' relative error in your result, then you should set $\epsilon = 1 - 1/(1+\epsilon')$. Bounds for the $1/r^p$ kernel are also possible:

- Relative error on kernel values is now $|1 - (d/d')^p|$.
- This is still maximized when we set $d' = d - \epsilon$ for feasible p and ϵ, which yields a new upper-bound relative error of $1/(1-\epsilon)^p - 1$.
- For ϵ' relative error in the result, instead set

$$\epsilon = 1 - 1/(1+\epsilon')^{1/p}.$$

The above reasoning applies specifically to positive integer values of p, though it may also be possible to extend this error bound to real $p \geq 1$, and perhaps even to $0 < p < 1$ as well.

The binomial approximation suggests that the relationship between ϵ, the error in the distance, and ϵ', the error in the result, is approximately a factor of p, meaning that even large values of p are feasible.

2.6 Range count and the n-point correlation

Although the n-point correlation, the 2-point correlation, and range count (the single-tree equivalent of the 2-point correlation) are well-modeled as kernel sums, the kernel they use is discontinuous, rendering $K'(h)$ infinite or undefined. Accordingly, our strategy for overcoming this problem is to permit error in distance threshold h instead of the returned kernel value.

In a nutshell, the fast algorithm for range count prunes when a node is either entirely inside $h + \epsilon$ or entirely outside $h - \epsilon$; nodes that satisfy both can be handled arbitrarily. The result that you get is then somewhere in between the true results computed for radii $h \pm epsilon$. This avoids doing too much work on nodes that intersect the pruning radius. An alternate, possibly more accurate form is to prune nodes entirely inside or outside of h (like exact computation), as well as nodes that lie between $h \pm epsilon$, say, by counting them as half, or even by percentage of included volume. Nodes with width less than epsilon will thus always prune. The previous

method prunes all nodes with width at most twice epsilon, but does not distinguish definitely excluded regions from regions that sit on the threshold.

Analysis now seems to be similar to the kernel sums above. We have a depth below which everything prunes and, similarly, a radius outside of which everything prunes. This should be able to bound the size of the cover set:

- Observe that the entire content of the cover set is contained by a ball of radius $h + 2^{i+2}$ centered on the query. (Note that we could use $h - \epsilon + 2^{i+2}$ for the less accurate method, but this does not help with analysis.)
- Disjoint balls of radius 2^{i-2} centered on immediate children of the cover set must then double enough times to cover the entire cover set, i.e. twice the above radius.
- If $2^{i+2} \geq h$, then the radius to cover is 2^{i+4}, for 6 total doublings and the number of immediate children bounded by c^6.
- Otherwise, the target radius is $4h$, for a number of doublings equal to $\lceil \lg h - i + 4 \rceil$.
- Because all nodes prune when their widths drop below epsilon, we have $2^{i+2} \geq \epsilon$, or $i \geq \lg \epsilon - 2$.
- We thus need at most $\lceil \lg(h/\epsilon) \rceil + 6$ doublings, for at most $c^{\lceil \lg(h/\epsilon) \rceil + 6}$ children overall.

This is similar to the other bound on kernel sums because both work from the facts that iteration must stop once it reaches a certain node width and that nodes provably prune at a given depth-dependent radius. In both cases, running time depends as much on bandwidth and error tolerance as it does on the number of points.

We now separately bound the size of the cover set for these the upper and lower portions of computation. We have $\lg(h/\epsilon)$ terminal iterations for a grand total of

$c^{\lceil \lg(h/\epsilon) \rceil + 6}$ node visits[6] and $c^2 \log N$ leading iterations times the smaller cover set size of c^6. Hence, range count's running time is overall

$$O(c^{\lceil \lg(h/\epsilon) \rceil + 6} + c^8 \log N).$$

Note that bandwidth and error tolerance have no multiplicative relationship to dataset size. This works for absolute error on Lipschitz continuous kernels as well by replacing $\lg(h/\epsilon)$ with $\lg(-K'(h)K^{-1}(\epsilon)/\epsilon)$ throughout.

Separately counting the upper and lower portions of computation unfortunately does not help the dual-tree algorithm solving the 2-point correlation. As many as $O(N)$ nodes may exist even at the deepest level; therefore, if we again use depth-first traversal, $O(N)$ times the maximum cover set size at any depth gives the algorithms overall run-time complexity of

$$O(c^{\lceil \lg(h/\epsilon) \rceil + 6} N).$$

2.7 The Axilrod-Teller potential

The Axilrod-Teller potential is a three-body potential that largely depends on the $1/r^p$ kernel applied to each pair of points in a triple of points as well as on the angles between the three points. Accordingly, its fast algorithm is similar to that of the N-body problem, but operating on node triples rather than node pairs. Pruning is strictly based on relative error in distance. This ends up translating to reasonable error bounds on the final result; however, those bounds are not quite relative nor absolute. Specifically, the numerator has absolute error (relative error is unreasonable as the numerator can be 0) and the denominator (or its inverse, rather) has relative error. The final result is then a sum over values with these mixed error bounds.

Recursion, for the most part, chooses to split the shallowest node(s) in the triple.

[6]The deepest iteration has a number of nodes bounded by the given maximum, but each next deepest level's bound is a factor of c less, and c is at least 2. This means that the sum of nodes over all of these levels is at most twice the number of nodes at the deepest level.

The exception is when one "far" node has less than the tolerated error in both of its distances to the other nodes. This far node is then never split again, even though the other two nodes may continue to split. In other words, the far node is pruned, even though the full node-triple is not. It is easy to see that once a node becomes a far node, its relative error in distance will always remain bounded. From here, the proof works by first observing that all visited node-triples result from splitting some node in a node-triple, and further that the splitting node cannot have been a far node. The splitting node must have been paired with at least one other node that is not a far node (otherwise the whole node-triple would have pruned), and that other node must be contained within some radius based on the error tolerance.

- The expansion constant then bounds the number of nodes that can fit in such a radius to below $c^{\lceil \lg(\frac{2}{\epsilon}+3)\rceil+4}$. This quantity squared is the maximum number of node-triples that can contain any given node in which no nodes are far nodes.

- This quantity by itself is also the maximum number of distinct (near) nodes that can occur with any given node in node-pairs also containing a far node. The task is then to determine how many far nodes can be paired with any given pair of near nodes.

- From here, the proof observes that a far node at a given depth i must be outside some radius dependent on i because it "pruned" at some point, but must also be inside some radius dependent on i because its parent *did not* prune. These radii end up defining disjoint nonempty annuli of the same sort as feature in the nearest-neighbors proofs, and thus there are only logarithmically many of them. Further each such shell can only contain a number of nodes again bounded by the quantity above.

- Thus, the running time of Axilrod-Teller is bounded by $O(c^{O(1)} N \log N)$, because each node can be grouped with constantly many near nodes and logarithmically many far nodes.

CHAPTER III

FAST DATABASE-RESIDENT MULTIVARIATE STATISTICS

We develop for the first time fast DBMS-resident algorithms for multivariate statistical operations—including all-nearest-neighbors, kernel density estimation, and the 2-point correlation—based on efficient multi-tree traversals. We implement these methods within a commercial DBMS, Microsoft SQL Server, and demonstrate their performance on real scientific data, the Sloan Digital Sky Survey. Empirical results suggest dramatic asymptotic speed-up over naive SQL implementations, with many orders of magnitude improvement for datasets containing millions of rows. This work demonstrates the scalability of multi-tree methods to datasets that cannot fit in RAM.

3.1 Introduction

The Sloan Digital Sky Survey (SDSS) [79] Catalog Archive Server (CAS) is a pioneering example of the use of modern database management systems (DBMS) for large scale scientific applications [72]. The SDSS is the largest archive of wide-area astronomical observations, detailing hundreds of measurements for each of 933 million unique sky objects. CAS makes 58TB of this data available over the web, permitting web users to perform simple searches and analyses via standard SQL queries. Further, CAS provides a number of rudimentary spatial queries, including nearest-neighbors search with regard to apparent sky position. CAS, however, lacks support for many other operations critical to machine learning and statistics, such as computing probability densities or simultaneously finding nearest neighbors for all objects in the database.

Until very recently, multivariate techniques such as kernel density estimation were not even feasible in the large-scale setting. Unlike the fast univariate analyses that have been possible in databases for years, the best performing multivariate machine learning methods are conventionally $O(N^2)$ or worse. However, recent breakthroughs using spatial data structures offer new hope for the computation of these measures in just $O(N)$ time [25]. These data structures and the multi-tree methods that use them find either exact results or results with guaranteed error bounds, can operate on high-dimensional data, and are broadly applicable to domains ranging from protein folding to market prediction to stellar redshift estimation. Their general strategy is to use divide-and-conquer, cached sufficient statistics, and bounds on partial results to eliminate unnecessary or repeated work whenever possible. Spatial join is a high-profile example of a multi-tree algorithm in DBMS literature, and many other examples exist in the field of machine learning, as discussed below.

Mainstream database management systems—including Microsoft SQL Server—have begun to support certain fast spatial analyses such as multivariate range queries. So far, however, these features have focused on two-dimensional data and only offer a limited, non-extensible selection of algorithms. For other multivariate analyses, including most multi-tree methods, scientists must currently use external programs or specialized databases. Alternatively, our work is a step towards the goal of embedding machine learning in existing database products. We introduce the most successful spatial data structures and algorithms in literature as direct extensions to Microsoft SQL Server. Our library is extensible and works in-place, sidestepping the cost of data transfer and transformation. Our implementation also avoids the need for complicated installation procedures; because our library runs within SQL Common Language Runtime (CLR), it is immediately available to all existing SQL Server users. The work presented in this chapter demonstrates a number of firsts:

- We develop the first library for general-purpose multi-tree computation that

can work on DBMS-resident data and, by extension, out-of-core data.

- Our code implements the first general-dimensional spatial data structure in a commercial DBMS.
- Our data structures exhibit the first use of cached sufficient statistics to accelerate machine learning methods in a DBMS.
- Our library makes available for the first time scalable algorithms for all-nearest-neighbors, kernel density estimation, and the 2-point correlation in a DBMS setting.
- Lastly, we obtain the first results of these three methods for a dataset with as many as 40 million rows.

An important note about the computational problems we treat is that they are batch data analyses, which emit results given the whole of the data and cannot operate by visiting each row, one at a time. Accordingly, we assume that the data is fixed for the duration of our analyses; this differs from the usual regime considered in DBMS literature and thus calls for different data management strategies.

3.2 Related work

A considerable amount of research has been put into the development of efficient indices for spatial data [24], ranging from the grid file to the R*-tree to the Bkd-tree [60]. These structures must strike a balance between fast spatial queries and fast inserts, updates, and deletes, usually obtaining similar asymptotic properties to the B+ tree. The spatial queries they serve often include the window query, which finds all objects within a given bounding box, and the distance query, which finds all objects within some distance of a given location. These queries are corollary to finding all entries within a given range of keys in a B+ tree, and when paired with the

appropriate index, they can exhibit run time performances within 20% of the optimal $O(\log N)$ lookup time for a database of N objects [55].

Another treated operation—one that is more closely related to the multi-tree methods presented in this thesis—is the spatial join. This finds all pairs of objects that overlap or that are within a given distance of each other. One can compute a spatial join in terms of other spatial queries by, for example, iterating over all M query objects and issuing a window query for each one. This at best $O(M \log N)$ approach is, however, inferior to even faster algorithms that exploit the tree structure both to answer individual queries and to share work between queries [9].

Yet another operation of significant interest to research and business communities alike is nearest-neighbors search. This query finds the object or objects in a database with the least distance to a given location. Nearest-neighbors search is often implemented as an extension of the distance query by shrinking the queried distance as objects are found until only the desired number of neighbors remain. Under certain conditions, this technique can be shown to be $O(\log N)$ for data stored in a cover tree [6]. Accordingly, the problem of finding nearest neighbors for a large number of queries can be solved in $O(M \log N)$ time again by iterating over each of the M queries, either one at a time or in small batches [80]. In this thesis, however, we show an even faster all-nearest-neighbors algorithm similar in concept to the faster spatial join algorithms [25].

Lastly, and while still on the topic of related work, we feel it important to note that, even though spatial data structures have been discussed in DBMS literature for decades, only a few have been implemented in mainstream database products. MySQL offers two-dimensional R-trees; Oracle offers two-, three-, and four-dimensional R-trees as well as quadtrees; R-trees are available in PostgreSQL's extensible GiST structure; and SQL Server employs a 4-layer hierarchical grid file. Each of these structures support many useful spatial queries and analyses, but none

of them permit the algorithms described in Sections 1.1 and 1.2.

3.3 DBMS-resident kd-tree construction

The first challenge of implementing any of these algorithms in a relational DBMS is constructing and storing the spatial data structure. Throughout this effort, we have assumed that the data is available in total at the time of tree construction, and thus we have focused on efficient bulk loading methods. The best such methods result in trees that are adaptive to the distribution of the data, e.g. kd-trees. However, this adaptivity inherently requires consideration and manipulation of a large amount of the data, if not all of it, even in the early stages of tree construction. We do not ordinarily have sufficient resources to perform these tasks in memory; thus, tree construction must be achieved via fast SQL queries.

A first attempt at SQL-based out-of-core tree construction directly adheres to the definition of a midpoint splitting kd-tree. The algorithm issues a query to obtain the data's overall minimum bounding box, an $O(N)$ computation because no spatial index already exists on the data. It then divides this box along the midpoint of its widest dimension and recurses to form subtrees for either of the node's newly formed children. Each recursive call again queries for the minimum bounding box of a portion of the dataset, but these subsequent queries are each still $O(N)$. This is because, although these queries represent constricted regions of space, SQL has no means of using the data structure we are presently forming and so must search through the entire dataset to find the queried points. This practice of re-querying must be carried out until the contents of nodes become small enough to fit in memory, allowing for more efficient completion of the bottom sections of the tree. For system memory capable of tree construction on M rows, the overall run time of this procedure is $2\frac{N}{M}O(N) = O(N^2)$, thereby defeating the purpose of forming the spatial index.

Thus, even before we finish constructing our data structure, we must be able

to perform fast, spatially local queries. We achieve this by building a hybrid tree structure, where upper portions are formed via fast but spatially nonadaptive SQL operations and lower portions are formed both efficiently and adaptively in memory. Specifically, we employ a Morton Z-ordering to quickly construct a quadtree-like structure, and then revise the lower portions of this structure by replacing them with kd-trees.

The Morton Z-ordering is a space-filling curve that is particularly easy to compute even in high dimensionalities while still affording good spatial locality properties. In two dimensions, each pair of bits in a point's z-value indicates which hierarchical quadrant the point lies within. As a result, z-values may be found directly from points' positions within their dataset's minimum bounding box. We exploit this property to quickly compute and index z-values for all rows. Afterwards, we are able to select ranges of z-values much more efficiently than we could the contents of arbitrary bounding boxes.

Maximal ranges of z-values sharing a common bit prefix correspond to partitioning hyper-rectangles in the data space. We select the largest such ranges—i.e., ranges with the shortest common prefixes—that can nonetheless still fit in system memory and form kd-trees on their contents. Because the ranges we select correspond to partitioning hyper-rectangles, the formed kd-trees have no overlapping bounding boxes. We proceed to recursively link adjacent kd-trees, adding parent nodes that unite the pair as siblings, until a single over-arching tree structure is formed. The only difference this structure has from a pure kd-tree is that, near the root, splits are chosen to bisect the full data space rather than the data's minimum bounding boxes.

We store formed kd-nodes in their own table, each row detailing a node's ID, child IDs, bounding box, and other useful statistics such as the amount of points it represents. Node IDs form a pre-order sorting of the tree. In addition, points are augmented with unique IDs which increment such that points may be sorted in

the same order as the leaves that contain them. Clustered indices are built on both of these keys for the purpose of rapid sequential access of spatially local nodes and data. Note that our arrangement of nodes on disk does not attempt to minimize the average number of pages visited on root-to-leaf traversals, but instead focuses on filling pages with spatially local leaves. In our experience, this practice assists dual-tree computation more so than careful attention to page height balance.

This tree construction procedure involves a total of three passes over the data and the formation of two clustered indices. Forming the clustered indices matches the kd-trees' construction time complexity at $O(N \log N)$, though this indexing process is heavily optimized by SQL Server.

3.4 DBMS-resident kd-tree interface

Our trees are somewhat specialized for multi-tree computation, but we have nonetheless abstracted their inner workings from the algorithms that might make use of them. The tree manages its own memory for its nodes and the data beneath them. This abstraction helps facilitate rapid algorithmic prototyping and development.

Our kd-tree interface and back-end utilities are coded in managed C# with the intention of being used in SQL CLR to perform complex computations directly on the database server. With as little hassle as possible, the tree provides developers a means of accessing nodes' bounding boxes, children, and represented data as if it were an in-memory structure. Under the hood, the tree uses a set associative cache to manage both its nodes and data. These caches have tunable parameters including page size and the number of sets, and rely on their corresponding tables' clustered indices to help reduce the cost of cache misses.

It is still the developer's responsibility to ensure their code does not run out of memory for other reasons. For instance, both all-nearest-neighbors and KDE require the maintenance of intermediate results at nodes in the query tree. It would be

Table 2: Run times for tree construction. The SQL clustered index is a component of forming the kd-tree, and TPIE additionally requires data to be exported from the database (not timed). Extrapolated results are italicized.

Rows	SQL index	SQL kd-tree	TPIE Bkd-tree
10m	114s	760s	325s
20m	229s	1471s	*690s*
30m	339s	2443s	*1050s*
40m	450s	4824s	*1430s*

too expensive to persist this frequently updated data to disk, so developers must instead keep it in their own buffers and be careful to segment the query tree into more manageable pieces before starting dual-tree recursion. Once computation has finished on a portion of the query tree, the final results may be written to a table.

3.5 Experiments

We perform a series of experiments using a dataset from the SDSS detailing 4D photometric data for 40 million observed sky objects. All of our experiments operate on dual-core 2GHz Athlon workstations with 1GB of RAM running Windows XP and Microsoft SQL Server 2008. We aim to show that our algorithms are vast improvements over even the best exhaustive implementations and, further, that they are competitive with more specialized libraries that incidentally incur significant data transfer and transformation costs.

In our first experiment, we consider the time it takes to build the trees used by our algorithms. For a point of reference, we break out the time SQL Server spends to create the clustered indices involved in our trees. We also compare to Bkd-tree construction in the Transparent Parallel I/O Environment (TPIE) [75] testbed, which implements a host of advanced spatial indices. Table 2 shows that spatial index creation is close to 10% of the cost of our tree construction, suggesting there may be some potential for further code optimization, but certainly no more than an order of magnitude. Also, we demonstrate that we are only around half the speed of TPIE

Table 3: Run times for all-nearest-neighbors. Naive is exhaustive all-pairs computation and Batched is a variation of Naive that amortizes access to the DBMS. Extrapolated results are italicized.

Rows	Naive	Batched	Single	Dual	TPIE
40k	2754s	159s	11s	4s	1.5s
200k	*66700s*	*3850s*	69s	29s	7.0s
2m	*6620000s*	*383000s*	2594s	488s	71s
10m	*5 years*	*0.3 years*	*19600s*	2652s	*360s*
20m	*20 years*	*1.2 years*	*44900s*	5872s	*730s*
30m	*45 years*	*2.7 years*	*72400s*	7423s	*1100s*
40m	*80 years*	*4.8 years*	*101000s*	13677s	*1500s*

Table 4: Run times for KDE and the 2-point correlation with bandwidth/radius set equal to 0.05.

	KDE		Two point	
Rows	Naive	Dual	Naive	Dual
10k	14s	16s	2s	1s
20k	62s	30s	8s	3s
30k	155s	51s	19s	4s
40k	279s	84s	34s	4s
50k	429s	136s	54s	6s
100k	1744s	350s	223s	12s

despite laking a similarly high degree of control over the system's disk storage.

In our second experiment, we demonstrate the capability of these trees to perform fast all-nearest-neighbors computation. We compare against the most obvious exhaustive SQL implementation, a blocked exhaustive implementation, the single-tree method of all-nearest-neighbors (using our trees), and the single-tree method as implemented in TPIE. Table 3 shows that all exhaustive implementations quickly become infeasible and that the dual-tree algorithm handily outpaces single-tree computation. Here, TPIE reaps the benefits of better control over its memory, but it should be noted that, due to design choices, TPIE cannot be easily extended to support dual-tree all-nearest-neighbors, let alone general multi-tree computation.

Next, we consider KDE. Table 4 shows a clear distinction between the quadratic

scaling of the naive exhaustive implementation and the linear scaling of the dual-tree algorithm.

Lastly, we examine the 2-point correlation. Table 4 again shows a clear asymptotic improvement over the naive algorithm.

3.6 Conclusion

We have demonstrated a sophisticated spatial data structure that works by piggy-backing on top of the native B+ trees present in a relational DBMS. Using this, we have shown that it is possible to incorporate advanced algorithms for the best performing machine learning analyses in existing DBMS installations. Among other things, the work illustrates the use of cached sufficient statistics, which has potentially even broader applicability to databases.

This library points the way to do many other statistical and machine learning computations in the DBMS setting that have already been shown in the in-memory case. Some examples on the horizon include the fast multipole method for KDE, nonparametric Bayes classification, and hierarchical clustering via the fast Euclidean minimum spanning tree algorithm. While our spatial trees are general purpose, some of these other techniques require more flexibility than they can currently offer. In particular, a direction we would like to take this work in the future is to permit incremental construction of the trees, perhaps in the style of Bkd-trees, as well as the ability to modify their contents.

Through continued development of our spatial tree infrastructure and the algorithms that it supports, we hope to make advanced machine learning analyses a familiar task among the database community.

CHAPTER IV

MASSIVE-SCALE KERNEL DISCRIMINANT ANALYSIS

In this chapter we take a deep look at a specific GNBP, and in doing so we shift to a different notation.

We describe a fast algorithm for kernel discriminant analysis, empirically demonstrating asymptotic speed-up over the previous best approach. We achieve this with a new pattern of processing data stored in hierarchical trees, which incurs low overhead while helping to prune unnecessary work once classification results can be shown, and the use of the Epanechnikov kernel, which allows additional pruning between portions of data shown to be far apart or very near each other. Further, our algorithm may share work between multiple simultaneous bandwidth computations, thus facilitating a rudimentary but nonetheless quick and effective means of bandwidth optimization. We apply a parallelized implementation of our algorithm to a large data set (40 million points in 4D) from the Sloan Digital Sky Survey, identifying approximately one million quasars with high accuracy. This exceeds the previous largest catalog of quasars in size by a factor of ten.

4.1 Classification Via Density Estimation

Kernel discriminant analysis (KDA) [21], also called kernel density classification [33] or nonparametric Bayes classification, is a nonparametric method for predicting the class of query observations given a set of labeled reference observations. It is particularly useful in scientific applications due to its balance of properties:

1. It is highly accurate, owing to its nonparametric form.

2. It is easy to use in that it requires little understanding of the problem's underlying model but still offers an intuitive means of incorporating prior knowledge.

3. If desired, its classifications are also accompanied by highly accurate probabilities of inclusion in each class, which can be useful in their own right.

We are motivated by a high-profile application in astrophysics pertaining to the identification of quasars. Believed to be active galactic nuclei, quasars are the brightest objects in the universe, typically out-shining their entire host galaxy by several orders of magnitude. As a result, they are the most distant and thus oldest objects we can see, making knowledge of their locations invaluable to the verification of theories such as dark energy. Prior to work with our astrophysicist collaborators [64], the largest available catalog of quasars listed fewer than one hundred thousand examples; massive sky surveys such as the Sloan Digital Sky Survey (SDSS), on the other hand, are expected to contain millions. Our goal is then to predict which of the unlabeled objects in the survey are quasars given a comparatively small (though still prohibitively large) sample of hand-identified objects.

Until recently, there was no feasible means of performing exact KDA when both the number of queries and number of references were large; traditionally, KDA is $O(N^2)$ when there are $O(N)$ queries and $O(N)$ references. Further, existing approximate methods do not provide hard bounds on introduced error. A new algorithmic approach presented in [29], however, offers marked improvement to running-time with *no* introduction of error. In this chapter, we describe a new algorithm similar to that of [29] which achieves dramatic asymptotic speed-up.

The remainder of this section details the mathematics of KDA. Following that, we examine what it takes to make a fast algorithm for KDA, juxtaposing previous work with the primary contributions of this paper. Sections 4.3 and 4.4 establish a framework for our algorithm and subsequently fill in the details. Section 4.5 then

considers accelerations to KDA's learning phase and Section 4.6 discusses the possibility of parallelization. In Section 4.7, we demonstrate our algorithm's superior performance on two data sets.

4.1.1 Bayes' Rule in Classification.

The optimal classifier on M classes assigns observation $x \in \mathcal{X}$ to the class C_k, $1 \leq k \leq M$, that has the greatest posterior probability $P(C_k|x)$ [63]. Applying Bayes' rule,

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}, \tag{33}$$

we assign x to C_k if, for all $l \neq k$, $1 \leq l \leq M$, we have

$$f(x|C_k)P(C_k) > f(x|C_l)P(C_l), \tag{34}$$

where $f(x|C)$ denotes the probability density of data sampled from C and $P(C)$ is the prior probability of C. (Note that Bayes' rule's denominator $f(x)$ cancels from either side.) It is typically given that $\sum_{k=1}^{M} P(C_k) = 1$, i.e. that there are no unexplained classes, and accordingly, that $f(x) = \sum_{k=1}^{M} f(x|C_k)P(C_k)$. In the case that $M = 2$, this implies that it is equivalent to classify x as C_1 when $P(C_1|x) =$

$$\frac{f(x|C_1)P(C_1)}{f(x|C_1)P(C_1) + f(x|C_2)P(C_2)} > 0.5. \tag{35}$$

4.1.2 Kernel Discriminant Analysis.

In place of the typically unknown values of $f(x|C)$, KDA uses kernel density estimates of the form

$$\widehat{f}_h(x|C) = \frac{1}{N}\sum_{i=1}^{N} K_h(x, z_i), \tag{36}$$

trained with bandwidth h on a size N set of reference observations $z_i \in \mathcal{X}$ of class C. Kernel K_h may take many forms but must be non-negative and have

$$\int_{\mathcal{X}} K_h(x, z)dx = 1 \tag{37}$$

for all z, i.e. it must be a probability density function. For $\mathcal{X} = \mathbb{R}^D$, a popular choice of kernel is the Gaussian,

$$K_h(x, z) = \frac{1}{\left(h\sqrt{2\pi}\right)^D} \exp\left(\frac{-\|x - z\|^2}{2h^2}\right), \tag{38}$$

though the optimal kernel for density estimation is in fact the Epanechnikov kernel [68],

$$K_h(x, z) = \max\left\{\frac{D + 2}{2V_D(h)}\left(1 - \frac{\|x - z\|^2}{h^2}\right), 0\right\}, \tag{39}$$

where $V_D(h)$ is the volume of a D-dimensional sphere with radius h. Observe that the Epanechnikov kernel is a parabolic function of distance for $\|x - z\| < h$ and zero otherwise; we will later exploit this property to accelerate computation. Also, note that many kernels are actually functions of distance $d(x, z) = \|x - z\|$; later in this chapter, K_h of one argument is understood to work on the distance term directly.

Proper selection of bandwidths h_k, $1 \leq k \leq M$, is critical for accurate classification results. Theory suggests that the optimal bandwidth for density estimation is related to a data set's variance and the inverse fifth-root of its size [68], but in practice, it is most effective to choose bandwidths that minimize KDA's cross-validation error or some other measure of loss.

4.1.3 Extensions to the Classifier.

It is not uncommon to have additional information $y \in \mathcal{Y}$ available for each observation x that is nonetheless difficult to incorporate into density estimates $\widehat{f}_h(x|C)$. For instance, corresponding information in $\mathcal{Y}$ may not be available for all of the references, or $\mathcal{Y}$ may not be favorable to kernels (e.g. may have no workable interpretation as $\mathbb{R}^D$). If it is reasonable to assume that y has negligible effect on class-conditional densities, we may still capture its effect in the class' priors with

$$P(C|x, y) \propto f(x|C, y)P(C|y) \approx f(x|C)P(C|y). \tag{40}$$

Similarly, for fine-tuned control over density estimates, we may provide weight w_i with each reference z_i, giving

$$\widehat{f}_h(x|C) = \sum_{i=1}^{N} w_i K_h(x, z_i), \tag{41}$$

where $\sum_{i=1}^{N} w_i = 1$.

Another modification possible in the $M = 2$ case replaces the 0.5 in (35) with an alternate confidence threshold t. Rather than performing optimal classification, this has the effect of identifying points with very high (or low) probability of being in C_1, which can be a useful surrogate for fully computed class probabilities. Simple algebra and incorporation of the above gives us

$$(1-t)\widehat{f}_{h_1}(x|C_1)P(C_1|y) > t\widehat{f}_{h_2}(x|C_2)P(C_2|y); \tag{42}$$

we will use this as our classification test for the remainder of this paper. This modification may seem trivial from the mindset of exhaustive computation, which provides fully computed class probabilities for each query, but it is useful under the new algorithmic approach. Our algorithm deals only in bounds on probabilities and aims to terminate computation on a query as soon as it is possible to demonstrate (42) or its converse. While we will know that the query's probability is definitely above or below the threshold, we have no guarantee of being able to show by how far.

4.2 Computational Considerations

An obvious means of performing KDA is to compute the full kernel summation for each query. For $O(N)$ queries and $O(N)$ references, this naive algorithm is $O(N^2)$ overall. While this approach is trivial to implement even in parallel, it does not scale to large N. In the best of conditions, a compute cluster with ten thousand nodes would only be able to handle a problem two orders of magnitude larger than is feasible on a single machine.

In order to make asymptotic improvement to running time, it is necessary to avoid a great deal of the explicit kernel evaluations. A new computational approach developed in [29] achieves this by rearranging work in a manner that permits consideration of subsets of queries and references at an abstract level. It is then hoped that bounds on results obtained at abstract levels can demonstrate that further work is unnecessary and can thus be *pruned*. For example, as suggested above, we may prune all remaining work on a subset of queries if density bounds prove (42) or its converse for each of the queries.

4.2.1 Previous Work.

The algorithm presented in [29] is best described as a high-order divide-and-conquer method for the $M = 2$ case of KDA. In a nutshell, it constructs two binary spatial trees, one on the query data and one on the reference data, with pairs of nodes from either tree representing abstracted computations. It then iteratively refines bounds on class-conditional density estimates $\widehat{f}_{h_1}(x|C_1)$ and $\widehat{f}_{h_2}(x|C_2)$ until each query's classification can be shown. Given bounding boxes for either node in a pair, bounds on the pair's contribution to densities are computed using kernel evaluations at the pair's minimum and maximum distances. Bounds are refined by, at each iteration, heuristically selecting an unrefined pair and replacing its contribution with the combined contributions of its four child pairs. If (42) becomes provably true or false for a query node, i.e. one class' lower-bound probability becomes greater than the other's upper-bound probability, then the queries represented by the node are appropriately labeled and further work on those queries is removed from consideration. Ideally, bounds give a query's classification long before all references are visited exhaustively, thereby abbreviating computation while still yielding the exact result.

4.2.2 This Thesis.

In this chapter, we directly extend the work done in [29]. Our new algorithm is the same in spirit as the old one and is also implemented specifically for the $M = 2$ case[1], but it benefits from several key differences:

- It uses an improved work order that better favors pruning and is implemented with reduced overhead and improved numerical stability.
- It uses the Epanechnikov kernel, which enables a new kind of pruning based on bounded distances between node pairs.
- It can compute results for multiple bandwidth combinations simultaneously, sharing work to accelerate bandwidth optimization.
- It can parallelize computation over the queries, making use of spatial information to reduce each processor's overall workload.

We achieve empirically asymptotic improvement over the previous best algorithm, met with multiple orders of magnitude improvement on the data sets we examined.

4.3 Foundations

Previous work [29] does not formalize the manner of rearranging work it uses to make efficient computation possible. We include the underlying mathematics here for the purposes of rigor and helping to clarify how the algorithm presented in this chapter works.

In all of the following, X is a subset of the full query set $X^{\text{root}} \subset \mathcal{X} \times \mathbb{R}$. Queries are given by pairs (x, π), where x is used in density estimation and $\pi = P(C_1|y)$. Symbol Z_k represents a subset of the weighted references $Z_k^{\text{root}} \subset \mathcal{Z} \times \mathbb{R}$ available for

[1]Note, however, that no inherent properties of the algorithm prevent it from being extended to $M > 2$.

for class C_k. References are pairs (z, w), where z is used in kernel computations and w is the weight. Note that the sum of weights of Z_k^{root} equals 1, but the same of Z_k generally does not. We use "root" to denote full sets because they are semantically equivalent to the roots of trees used by our algorithm.

4.3.1 Recursive Formulation.

We represent density estimation with a set of key-value pairs $\phi_k(X, Z_k)$, where

$$\phi_k(X, Z_k) = \left\{ \left(x, \sum_{(z,w) \in Z_k} w K_{h_k}(x, z) \right) \middle| (x, \pi) \in X \right\}. \tag{43}$$

Classification is then a task of comparing density estimates for matching keys in $\phi_1(X, Z_1)$ and $\phi_2(X, Z_2)$ in accordance with (42).

For partitions $X^L \cup X^R = X$ and $Z_k^L \cup Z_k^R = Z_k$, observe that

$$\phi_k(X, Z_k) = \phi_k(X^L, Z_k) \cup \phi_k(X^R, Z_k) \tag{44}$$

$$= \phi_k(X, Z_k^L) + \phi_k(X, Z_k^R), \tag{45}$$

where addition on sets of key-value pairs is understood to add values for matching keys. This immediately suggests a recursive alternative to the naive algorithm in which the query set and reference sets are repeatedly split until they become singleton, whereupon ϕ_k may be computed directly. Note that this reformulation by itself does nothing to help asymptotic running time because, without pruning, each of the $O(N^2)$ kernel evaluations will ultimately occur. What it does offer is the occurrence of $\phi_k(X, Z_k)$ for nontrivial subsets $X \subseteq X^{\text{root}}$ and $Z_k \subseteq Z_k^{\text{root}}$, instrumental to bounds computation and pruning.

Our algorithm cannot afford to spend much time deciding splits $X^L \cup X^R = X$, etc., so we use space partitioning trees to decide these splits in advance. This has the effect of forcing the same partitions to occur throughout computation, though this is not necessary in the underlying math.

4.3.2 Bounds Computation.

It is typically possible to bound kernel results between points in X and Z_k if we know bounding boxes or bounding radii for these sets. In the case of the Gaussian and Epanechnikov kernels, bounds are functions of upper- and lower-bound distances $d^u(X, Z_k)$ and $d^l(X, Z_k)$ between the regions containing X and Z_k, with

$$K^u_{h_k}(X, Z_k) = K_{h_k}(d^l(X, Z_k)) \tag{46}$$

and vice versa. Bounded contributions to density then assume that all references are either maximally near to or maximally far from the queries, and are given by

$$\phi^u_k(X, Z_k) = W(Z_k)K^u_{h_k}(X, Z_k) \tag{47}$$

and similarly for $\phi^l_k(X, Z_k)$, where

$$W(Z_k) = \sum_{(z,w)\in Z_k} w. \tag{48}$$

Note that bounds have zero width when X and Z_k are singleton and are thus equivalent to exact computation.

To make use of these bounds in classification, we must compose them to form bounds on the full density estimate, i.e. such that they bound a kernel summation over the full reference set. For a query $(x, \pi) \in X$, this is achieved by combining the bounds of P components of computation with

$$\Phi^u_k(x) \leftarrow \sum_{p=i}^{P} \phi^u_k(X^p, Z^p_k) \tag{49}$$

and similarly for $\Phi^l_k(x)$, where Z^p_k, $1 \le p \le P$, forms a partition of Z^{root}_k and $(x, \pi) \in X^p$ for each p. We further define

$$\Phi^u_k(X) \leftarrow \max_{(x,\pi)\in X} \Phi^u_k(x), \quad \Phi^l_k(X) \leftarrow \min_{(x,\pi)\in X} \Phi^l_k(x), \tag{50}$$

which can be computed directly if $X \subseteq X^p$ for all p.

4.3.3 Iterative Refinement.

Computation under the new algorithmic approach implicitly constructs two binary expression trees, one for either class, in accordance with the recursive formulation given in (44) and (45). The nodes of these trees represent partial density estimations $\phi_k(X, Z_k)$, with interior nodes symbolically composing their children via $\cup$ or $+$. Operation begins with trees initialized to single nodes for $\phi_1(X^{\text{root}}, Z_1^{\text{root}})$ and $\phi_2(X^{\text{root}}, Z_2^{\text{root}})$, and at each iteration *expands* a selected, non-singleton leaf into two children reflecting having split either its queries or references. After expansion, the algorithm reassesses bounds on the full density estimates for the involved queries as in (49), using bounded results computed for the newly created leaves in conjunction with bounds already available from other leaves.[2] We stop expanding nodes $\phi_k(X, Z_k)$ for which we can show either of

$$(1-t)\Phi_1^l(X)\Pi^l(X) > t\Phi_2^u(X)(1-\Pi^l(X)), \tag{51}$$

$$t\Phi_2^l(X)(1-\Pi^u(X)) > (1-t)\Phi_1^u(X)\Pi^u(X), \tag{52}$$

where

$$\Pi^u(X) = \max_{(x,\pi)\in X} \pi, \quad \Pi^l(X) = \min_{(x,\pi)\in X} \pi, \tag{53}$$

which decides (42) for all queries in X.

It is easy to see that this procedure terminates. For finite X and Z_k, recursion will ultimately reach singleton sets $X = \{(x, \pi)\}$ and $Z_k = \{(z, w)\}$, where upon contribution bounds computations become exact, i.e. the upper and lower bound become equal, and recursion along that branch ceases. If all branches for some x reach singleton leaves, then bounds on the full density estimate are also exact and thus one of (51) or (52) must be true[3] for each singleton X. If branches do not reach

[2] It is easy to prove by induction that, for any $(x, \pi) \in X$, we can find leaves that satisfy the requirements of (49).

[3] Or either side is exactly equal, but this is degenerate and may be thought of as a third, non-classifying prune.

leaves then it is only because x has already been pruned. Thus, all queries x will eventually prune and iterative refinement will stop. Correctness is more difficult to show rigorously; we omit the proof because it is nonetheless fairly intuitive.

4.4 The Algorithm

In the previous section, we did not impose any requirements on how one chooses partitions $X^L \cup X^R = X$, etc., or on the order of leaves expanded during computation. These details have no impact upon correctness but are nonetheless critical with regards to running time. An additional concern is the practical storage of computed bounds in a manner that assists with finding bounds on the full density estimate. Addressing these issues, we also describe a new pruning opportunity and the bookkeeping that facilitates it, ultimately arriving at the algorithm in Figure 14.

4.4.1 Spatial Trees.

Both the algorithm in [29] and the one presented in this chapter use *kd*-trees [59], a kind of spatially-informed tree, to facilitate the splitting of query and reference sets. We construct two trees, one for X^{root} and one for the combined[4] references Z_1^{root} and Z_2^{root}, in which each interior node represents a partitioning $X^L \cup X^R = X$, etc.; we will reuse these partitions any time we need to split data later in computation. Points stored beneath each node are represented by their bounding box and child nodes are formed by splitting the widest dimension of the parent's bounding box along its midpoint. Leaves are formed when the number of represented points drops below a specified threshold; the algorithm stops partitioning a set once it reaches a leaf and computes $\phi_k(X, Z_k)$ exhaustively if both X and Z_k are leaves. Trees also store useful statistics on represented queries and references, such as bounds on prior probability $\left[\Pi^l(X), \Pi^u(X)\right]$ and summed reference weight $W(Z_k)$, which may

[4] Alternately, separate trees could be used for each class.

be computed rapidly in a bottom-up manner. Lastly, the query tree provides a convenient location to store sums of computed bounds; [29] makes heavy use of this, but the algorithm we propose does not.

The primary motivation for spatial trees is that they tend to tighten bounds rapidly as we expand $\phi_k(X, Z_k)$ even though splits are made without any knowledge of interaction between X and Z_k. In other words, we would not expect to see gigantic improvement in bounds from choosing custom partitions for each expansion, a comparatively expensive procedure. Tree building incurs $O(N \log N)$ computations, but the cost of this step is small compared to the rest of the algorithm except for very large data sets. It is also possible to reuse existing spatial trees on the data, perhaps computed because some prior algorithm needed them as well.

Due to the conjoined nature of our reference tree, matching simultaneous expansions are made to $\phi_1(X, Z_1)$ and $\phi_2(X, Z_2)$. Bounds and pruning on these computations are informed by separate bounding boxes and statistics stored for either class; if ever a reference node ceases to contain points from one of the classes, the node is removed from consideration when bounding that class' kernel summation. We will use $Z_{1,2}$ to denote nodes of the combined tree, from which both Z_1 and Z_2 are available.

Because sets X and Z_k occurring in computation constitute nodes from *kd*-trees, we will henceforth refer to components of computation $\phi_k(X, Z_k)$ as *node-pairs*.

4.4.2 Expansion Pattern.

To guide computation, [29] uses a heuristic favoring the expansion of node-pairs that provide the most bounds tightening over the contribution of their parents. This may sound desirable with regards to testing (51) and (52), but it does not guarantee that further expansion will provide much improvement and is prone to missing node-pairs that are just about to make a "breakthrough." A further problem of this kind of expansion is its dependence upon priority queues (implemented, for instance,

as a heap), which in our experience impose significant computational overhead and cache inefficiency. Also, because potentially any query node may follow the present computation, it is necessary to propagate and later undo all computed bounds to all levels of the query tree to which they apply, possibly impacting numerical stability.

Our algorithm uses a non-heuristic expansion pattern best thought of as a hybrid between depth- and breadth-first. In short, queries are handled depth-first while references are handled breadth-first. This pattern's associated data structure is a stack of lists of node-pairs, where each list pertains to one query node X and all references nodes $Z_{1,2}^p$, $1 \leq p \leq P$, at some level of the reference tree, i.e. with $\bigcup_{p=1}^{P} Z_{1,2}^p = Z_{1,2}^{\text{root}}$. Bounds computation over a list consists of accumulating the partial bounds found for all node-pairs $\phi_1(X, Z_1^p)$ and $\phi_2(X, Z_2^p)$ as per (49). Query splits are performed by replacing the list at the top of the stack with two equal-length lists pertaining to either child of X and the same nodes $Z_{1,2}^p$. Reference splits instead replace the list with an at most double-length list pertaining to the same X and all of the children of nodes $Z_{1,2}^p$. Leaf nodes $Z_{1,2}^p$ are left unsplit until X becomes a leaf, at which point they are processed exhaustively and removed, their contribution added to exact results stored for each query point. In practice, we always split queries and references simultaneously if possible. Processing lists in this manner results in a logarithmically deep stack containing lists of sizes doubling from 1 to $O(N)$, thus consuming $O(N)$ space overall. This is more overhead than depth-first's $O(\log N)$ space, but significantly less than breadth-first's $O(N^2)$ (pessimistically assuming that no prunes occur).

Because bounds computed for one set of queries do not influence the results of another, the order in which nodes are processed from the query tree does not have any effect on pruning. The new expansion pattern exploits this to minimize overhead while still offering tightened bounds from all parts of the reference tree, crucial in demonstrating queries' classifications. Further, each list in the new expansion pattern offers

sufficient information to find the most current bounds on X's full kernel summation: the respective sums of upper and lower bounds obtained for the represented node-pairs. This eliminates any need to propagate or undo bound contributions throughout the query tree, saving time as well as minimizing the effect of limited floating-point precision.

4.4.3 Pruning.

We have already motivated *classification pruning*, where one class' lower-bound probability exceeds the other's upper-bound probability. Each time the hybrid expansion pattern forms a new work list, it finds bounds on the full density estimates and tests (51) and (52); if either of these are true, it labels the represented queries appropriately and pops the list from the stack.

The Epanechnikov kernel makes possible two other very important pruning opportunities. As mentioned before, this kernel is parabolic for distances smaller than bandwidth h and zero otherwise. Thus, if the lower-bound distance between X and Z_k is greater than the bandwidth, those points' contribution to the kernel sum will be exactly zero. This makes for a convenient *exclusion prune*: node-pairs $\phi_{1,2}(X, Z_{1,2})$ with $d^l(X, Z_{1,2}) \geq \max\{h_1, h_2\}$ may be removed from the hybrid expansion pattern's work lists.

Similarly, the Epanechnikov kernel's parabolic interior enables *inclusion pruning*, though this is somewhat more involved. Because a sum of parabolas is a parabola, we may compute the exact kernel summation over a set of references Z_k at once if we can prove it is entirely within bandwidth h of some query x. By extension, if $d^u(X, Z_{1,2}) \leq \min\{h_1, h_2\}$, we may exactly compute $\phi_{1,2}(X, Z_{1,2})$ in $O(N)$ or find tighter than normal bounds in constant time. The derivation of this prune observes that square distance may be computed

$$(x - z) \cdot (x - z) = x \cdot x - 2x \cdot z + z \cdot z, \tag{54}$$

ultimately yielding the expression

$$W(Z_k)\frac{D+2}{2V_D(h_k)}\left(1-\frac{\delta(x,Z_k)}{h_k^2}\right) \tag{55}$$

in place of the original kernel summation, where

$$\delta(x,Z_k) = x\cdot x - 2x\cdot\tfrac{S(Z_k)}{W(Z_k)} + \tfrac{T(Z_k)}{W(Z_k)}, \tag{56}$$

$$S(Z_k) = \sum_{(z,w)\in Z_k} wz, \qquad T(Z_k) = \sum_{(z,w)\in Z_k} wz\cdot z; \tag{57}$$

note that $S(Z_k)$ and $T(Z_k)$ may be precomputed at the same time as $W(Z_k)$. This constitutes a parabola centered at $\frac{S(Z_k)}{W(Z_k)}$. To employ this in pruning, each work list must record respective sums W_k, S_k, and T_k of $W(Z_k)$, $S(Z_k)$, and $T(Z_k)$ for pruned nodes Z_k, replicating these sums into future lists formed via expansion. These terms will eventually provide exact density contributions to individual queries once X is a leaf, but before expansion reaches that point, we compute tighter than normal bounds with

$$W_k K_{h_k}(\delta^l(X,W_k,S_k,T_k)) \tag{58}$$

and vice versa, where

$$\delta^u(X,W_k,S_k,T_k) = d^u\Big(X,\tfrac{S_k}{W_k}\Big) - \tfrac{S_k}{W_k}\cdot\tfrac{S_k}{W_k} + \tfrac{T_k}{W_k} \tag{59}$$

and similarly for $\delta^l(X,W_k,S_k,T_k)$.

4.5 Bandwidth Learning

Class-dependent bandwidths h_k have a significant impact on classification accuracy, making the determination of optimal bandwidths a critical phase in any application of KDA. In practice, the best approach to bandwidth optimization is minimizing cross-validation classification error. Our algorithm is capable of leave-one-out cross-validation (LOO CV) at negligible additional cost. Further, a specialized multi-bandwidth version of our algorithm shares work between multiple bandwidth combinations, greatly reducing the time necessary to explore the space of bandwidth settings.

Algorithm 14 Pseudocode for kernel discriminant analysis. Each recursion begins by testing all nodes from a level of the reference tree against the query node, pruning those it can while simultaneously accumulating the contributions of others and building the next level's work list. Pruned contributions are then applied to computed density bounds and bounds are tested to see if they demonstrate a classification. Handling of base-case omitted.

init $X = X^{\text{root}}$, work $= \{Z_{1,2}^{\text{root}}\}$
init $W_{1,2} = 0,\ S_{1,2} = 0,\ T_{1,2} = 0$
function KDA$(X, \text{work}, W_{1,2}, S_{1,2}, T_{1,2})$
  init $\Phi_1^l = 0,\ \Phi_1^u = 0,\ \Phi_2^l = 0,\ \Phi_2^u = 0$
  init next_work $= \{\}$ // filled below
  for all $Z_{1,2} \in$ work **do**
    if $d^u(X, Z_{1,2}) \leq \min\{h_1, h_2\}$ **then**
      // Inclusion prune; add to parabola
      $W_1 \mathrel{+}= W(Z_1)$; $W_2 \mathrel{+}= W(Z_2)$
      ... // Similar for $S_{1,2}$ and $T_{1,2}$
    else if $d^l(X, Z_{1,2}) < \max\{h_1, h_2\}$ **then**
      // No prune; record contribution
      $\Phi_1^l \mathrel{+}= W(Z_1) K_{h_1}(d^u(X, Z_1))$
      ... // Similar for Φ_1^u, Φ_2^l, and Φ_2^u
      add Z^L **and** Z^R to next_work
    else// Exclusion prune; do nothing
    end if
  end for
  // Account for inclusion prunes
  $\Phi_1^l \mathrel{+}= W_1 K_{h_1}(\delta^u(X, W_1, S_1, T_1))$
  ... // Similar for Φ_1^u, Φ_2^l, and Φ_2^u
  if $(1-t)\Phi_1^l \Pi^l(X) > t\Phi_2^u(1 - \Pi^l(X))$ **then**
    label X **as** C_1; **return**
  else if $t\Phi_2^l(1 - \Pi^u(X)) > (1-t)\Phi_1^u \Pi^u(X)$ **then**
    label X **as** C_2; **return**
  end if
  KDA$(X^L, \text{next_work}, W_{1,2}, S_{1,2}, T_{1,2})$
  KDA$(X^R, \text{next_work}, W_{1,2}, S_{1,2}, T_{1,2})$
end function

Fast bandwidth optimization is arguably the most significant contribution of this chapter. Often times, full density estimates are more useful than just classifications and existing fast algorithms can approximate these with high accuracy [25, 27, 43, 41]. Regardless, these estimators must undergo the same learning as KDA before their results can be trusted. Classification pruning, available in KDA but not regular kernel density estimation, can greatly accelerate this preliminary phase. Further, the results of the KDA learning phase may be used to restrict computations of full density estimates to just those points identified as members of some class of interest.

4.5.1 Leave-one-out Cross-validation.

In cross-validation situations, queries X^{root} and references $Z_{1,2}^{\text{root}}$ represent the same data. It is handy to denote data as a set $X_{1,2} \subset \mathcal{X} \times \mathbb{R} \times \mathbb{R}$, separable into sets X_1 and X_2 for either class and with elements (x, π, w): the observation, its data-dependent prior, and its reference weight.

Our goal is to find the results for each (x, π, w) that would be obtained from normal computation if $Z_{1,2} = X_{1,2} \setminus (x, \pi, w)$. It is sufficient and safe to relax density bounds computed for $X_{1,2}$ to reflect having left out one point from the appropriate classes. In the case of the Gaussian and Epanechnikov kernels, a point certainly contributes the maximum kernel value to itself. Leaving a point out then results in subtracting the maximum kernel value at that point's weight from both the upper and lower bounds for its class, renormalizing bounds such that weights again sum to one. Because any changes we make to bounds must apply to all points represented by $X_{1,2}$, we must find a lower bound on changes made to the lower bounds and an upper bound on changes made to the upper bounds. If $X_{1,2}$ contains points of either class, then a suitable modification to the lower bounds is to subtract the maximum kernel value at the maximum represented weight but *not* renormalize, and a suitable change to the upper bounds is to renormalize *without* having subtracted anything.

The new bounds are then

$$\Phi_k^l(X_{1,2}) - w^u(X_k)K_{h_k}(0), \quad \frac{\Phi_k^u(X_{1,2})}{1 - w^u(X_k)}, \tag{60}$$

where

$$w^u(X_k) = \max_{(x,\pi,w)\in X_k} w. \tag{61}$$

Clearly, if $X_{1,2}$ contains only points from one class (i.e. one of X_1 or X_2 is empty), then tighter modifications can be made. This is imperative in the base-case because a classification may never be found otherwise.

4.5.2 Multi-bandwidth Computation.

Classification accuracy is nondifferentiable, preventing the use of most rapidly converging optimization techniques to find favorable bandwidths. Indeed, it is not uncommon for researchers to test bandwidths by hand until accuracy seems to have peaked, a time-consuming process prone to error and wasted effort. Here, we describe an extension to our algorithm that swiftly and systematically considers all pairs of two sets of bandwidths.

Given sorted lists of bandwidths h_1^i, $1 \leq i \leq I$, and h_2^j, $1 \leq j \leq J$, we modify the standard algorithm such that it maintains vectors of corresponding length for computed density bounds $\vec{\Phi}_k^{l,u}$ and pruning information $\vec{W}_k$, $\vec{S}_k$, and $\vec{T}_k$. The modified algorithm iterates over the bandwidths from largest to smallest when considering each item from the work list, reusing distance computations and shortcutting computation on smaller bandwidths when a larger one prunes via exclusion. We annotate work items Z_k with $\text{lo}(Z_k)$ and $\text{hi}(Z_k)$, the lowest and highest indices of bandwidths not yet pruned via inclusion or exclusion. Iteration is constrained to between these indices, and the work item is removed from future lists if no unpruned bandwidths remain. The algorithm then considers all pairs of bandwidths for classification pruning. We also annotate query nodes X with $\text{lo}_k(X)$ and $\text{hi}_k(X)$ for either class, the lowest and

highest indices of bandwidths for which classifications are yet to be made. Iteration for both density computation and classification is constrained to within these indices, and the entire work list is popped when the query node has been classified for all bandwidth pairs. Pseudocode for this algorithm is shown in Figure 15.

There is only a factor of $O(I + J)$ more work needed to compute the bandwidths' associated densities, but there are $O(IJ)$ classification tests to perform at each level of recursion. The classification test is, however, very fast in comparison to density estimation. We should then expect to see running time grow roughly linearly for bounded values of I and J. The sharing of tree building, distance computations, and exclusion pruning also provide speed-up over running the standard algorithm on all of the bandwidth combinations.

We may use this multi-bandwidth version of the algorithm in a rudimentary but effective means of bandwidth optimization. Starting from theoretic approximations of the optimal bandwidths [68] or some simpler measure such as the average kth-nearest-neighbor distance[5], test ten or so bandwidths ranging from an order of magnitude larger to an order of magnitude smaller. Then, iteratively narrow ranges to contain just the few bandwidths found to yield the greatest accuracy, or extend the range if the greatest accuracy bandwidths lie on the edge. This can only hope to offer linear convergence upon a local optimum, but it is conceptually similar to the well-known Nelder-Mead method of optimization and in practice finds reasonable bandwidths in just a few iterations. It is also possible to reuse trees between iterations.

4.6 Parallelization

The naive algorithm for KDA is highly parallelizable, but parallelized implementations still suffer from the method's inherent poor scalability. Even assuming communication cost to be negligible, it is likely uneconomical to purchase one-hundred-fold

[5] A fast algorithm for this problem is suggested in [25].

more processors to tackle a problem just one order of magnitude larger than was previously feasible.

The algorithm we propose is also parallelizable. There are no read/write dependencies between sibling recursions; thus, it is safe for these tasks to be handled by different processors. For P processors and grain size parameter G, we perform median (rather than midpoint) splits in the query tree up to depth $\lceil \log PG \rceil$, enqueuing each node at that depth in a global work list. This ensures that there are at least G grains per processor and that grains are all about the same size. As they become available, processors are assigned computations $\phi_{1,2}(X, Z_{1,2}^{\text{root}})$ for successive nodes X in the global work list. The use of multiple grains per processor helps alleviate inevitable irregularities in the amount of time it takes to classify different regions of the queries; using too many grains, however, both incurs communication overhead and reduces the degree to which work can be shared between queries.

This method of parallelization should be more advantageous than simply breaking the queries into chunks before forming trees. The initial query nodes assigned to processors are more compact than they would otherwise be, thus encouraging the earlier pruning of much of the reference set. This has the effect of restricting the amount of the reference tree needed by individual processors, which can be exploited by an advanced data distribution system to reduce overall communication. We must note, however, that this method presupposes the formation of the query tree before parallelization can begin. At present, tree building is performed in serial, but future work aims to address whether that step can also be parallelized.

4.7 Results

We applied our algorithms to two data sets: a large selection of quasars and stars from the SDSS and the Cover Type data set available from the UCI Machine Learning Repository [3].

Algorithm 15 Pseudocode for the multi-bandwidth algorithm. For either class, the algorithm checks for inclusion prunes at each h and then computes normal bounds until the first exclusion prune. Pruned contributions are then applied for each bandwidth and bandwidth combinations are tested for classification. Base-case omitted.

init $X = X^{\text{root}}$, work $= \{Z_{1,2}^{\text{root}}\}$, $\vec{W}_{1,2} = 0$, $\vec{S}_{1,2} = 0$, $\vec{T}_{1,2} = 0$
function MULTI(X, work, $\vec{W}_{1,2}, \vec{S}_{1,2}, \vec{T}_{1,2}$)
  init $\vec{\Phi}_1^l = 0$, $\vec{\Phi}_1^u = 0$, $\vec{\Phi}_2^l = 0$, $\vec{\Phi}_2^u = 0$, next_work $= \{\}$ // filled below
  for all $Z_{1,2} \in$ work **do**
    for i **from** $\text{hi}(Z_1)$ **to** $\text{lo}(Z_1)$ **do**
      if $d^u(X, Z_1) > h_1^i$ **break**
      // Inclusion prune; add to parabolas
      $\vec{W}_{1(i)} \mathrel{+}= W(Z_1)$; ... // Similar for $\vec{S}_1$ and $\vec{T}_1$
    end for
    for i **continuing to** $\text{lo}(Z_1)$ **do**
      if $d^l(X, Z_1) \geq h_1^i$ **break**
      // No prune; record contributions
      $\vec{\Phi}_{1(i)}^l \mathrel{+}= W(Z_1) K_{h_1^i}(d^u(X, Z_1))$
      $\vec{\Phi}_{1(i)}^u \mathrel{+}= W(Z_1) K_{h_1^i}(d^l(X, Z_1))$
    end for
    // No work for exclusion prunes
    update $\text{lo}(Z_1)$ **and** $\text{hi}(Z_1)$; ... // Similar for Z_2
    if $\text{lo}(Z_1) \leq \text{hi}(Z_1)$ **or** $\text{lo}(Z_2) \leq \text{hi}(Z_2)$ **then**
      add Z^L **and** Z^R **to** next_work
    end if
  end for
  for i **from** $\text{lo}_1(X)$ **to** $\text{hi}_1(X)$ **do** // Account for inclusion prunes
    $\vec{\Phi}_{1(i)}^l \mathrel{+}= \vec{W}_{1(i)} K_{h_1^i}(\delta^u(X, \vec{W}_{1(i)}, \vec{S}_{1(i)}, \vec{T}_{1(i)}))$
    $\vec{\Phi}_{1(i)}^u \mathrel{+}= W_{1(i)} K_{h_1^i}(\delta^l(X, W_{1(i)}, S_{1(i)}, T_{1(i)}))$
  end for
  ... // Similar for $\vec{\Phi}_2^l$ and $\vec{\Phi}_2^u$
  for i **from** $\text{lo}_1(X)$ **to** $\text{hi}_1(X)$, j **from** $\text{lo}_2(X)$ **to** $\text{hi}_2(X)$ **do**
    if $(1-t)\vec{\Phi}_{1(i)}^l \Pi^l(X) > t\vec{\Phi}_{2(j)}^u (1 - \Pi^l(X))$ **then**
      label X **as** C_1 **for** (i, j)
    else if $t\vec{\Phi}_{2(j)}^l (1 - \Pi^u(X)) > (1-t)\vec{\Phi}_{1(i)}^u \Pi^u(X)$ **then**
      label X **as** C_2 **for** (i, j)
    end if
  end for
  update $\text{lo}_1(X)$, $\text{hi}_1(X)$, $\text{lo}_2(X)$, and $\text{hi}_2(X)$
  if $\text{lo}_1(X) \leq \text{hi}_1(X)$ **and** $\text{lo}_2(X) \leq \text{hi}_2(X)$ **then**
    MULTI(X^L, next_work, $\vec{W}_{1,2}, \vec{S}_{1,2}, \vec{T}_{1,2}$)
    MULTI(X^R, next_work, $\vec{W}_{1,2}, \vec{S}_{1,2}, \vec{T}_{1,2}$)
  end if
end function

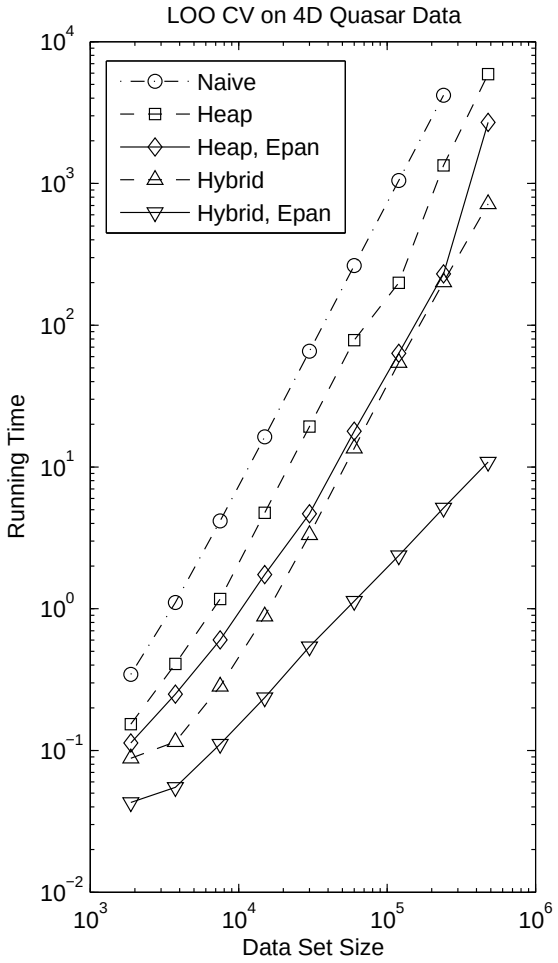

Figure 2: Running times of LOO CV on one bandwidth combination for the naïve algorithm, the algorithm from [29] (Heap), the improved expansion pattern (Hybrid) and pruning opportunities (Epan) separately, and finally the improvements together. Bandwidths were determined to maximize classification accuracy on the largest set and scaled via the theoretic inverse relationship to $N^{\frac{1}{5}}$. Only one processor used.

All experiments are conducted on code compiled by gcc 3.4.6 in Linux 2.6.9 on dual NetBurst-class Intel Xeon 3.0GHz computers with 8GB RAM.

4.7.1 Quasar Identification.

We demonstrate timing results for KDA classifiers on 4D color spectra data pertaining to deep-space objects. The largest reference set we use contains 80k known quasars and 400k non-quasars; our query set consists of 40m unidentified objects. In Figure 2, it is apparent that both the hybrid expansion pattern and inclusion/exclusion pruning are crucial to asymptotic performance. Our new algorithm empirically scales at about $O(N^{1.1})$ and has an extremely low cross-over point with the naive algorithm.

Three multi-bandwidth runs with 10 bandwidths per class, successively narrowing the tested bandwidth ranges about the optimum, take 334, 127, and 86 seconds when parallelized over two processors. Decreasing running time is expected both because bandwidths are easier to prune together if they are more similar to each other and because larger bandwidths, which diminish as ranges tighten, tend to prune the least. Figure 3 demonstrates that the multi-bandwidth algorithm is indeed much faster for a bounded number of bandwidths, which is all that is necessary for optimization. In comparison, the estimated naive running time for the above procedure is 140 hours even if the algorithm utilizes the same density-sharing technique employed by the multi-bandwidth algorithm. Optimal bandwidths yield within-class of accuracies of 95% for quasars and 98% for stars.

Finally, we parallelize over multiple computers to tackle classification of the full data set. Using the a cosmologically informed prior of 4% for the quasar class, three dual-processor machines discover just shy of one million quasars in 456 seconds. Unfortunately, our parallelization methods see poor speed-up due to serial data access and tree building, as shown in Figure 4. These steps alone take about 310 seconds, the better portion of computation; the whole serial computation takes only 640 seconds.

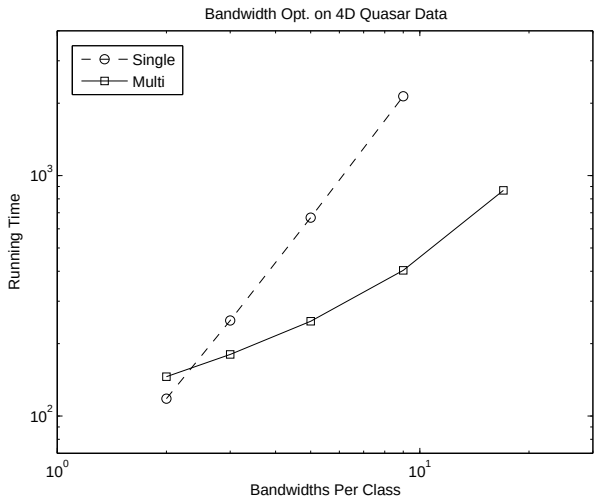

Figure 3: Running times for the multi-bandwidth algorithm and the single-bandwidth algorithm run on all bandwidth combinations. Bandwidth range is constant between runs and bandwidths are tested at linear intervals. Only one processor used.

An important future avenue of research is parallelizing tree building, but at current we are forced to conclude that parallelization is not worthwhile at this scale. In any case, we far improve upon both the estimated naive running time of 380 hours, not to mention classifying the entire unknown data in a tenth the time it takes the old algorithm to test one bandwidth combination.

Using careful tuning of parameters and separate classifiers for separate regions of data, our collaborators have extracted more than one million quasars from the SDSS data. Hand-verification of subsets of this data predicts that identified quasars are 85.6% accurate and, further, that we have identified 93.4% of the quasars to be found.

4.7.2 Forest Cover-Type Data.

Originally used in [7], the forest cover-type data is a 55-dimensional (10 continuous, 44 binary, 1 nominal classification) data set with 580k observations describing the relationship between cartographic variables and seven classes of forest cover-type, i.e. what kind of trees grow there. Because KDA is not particularly good at qualitative

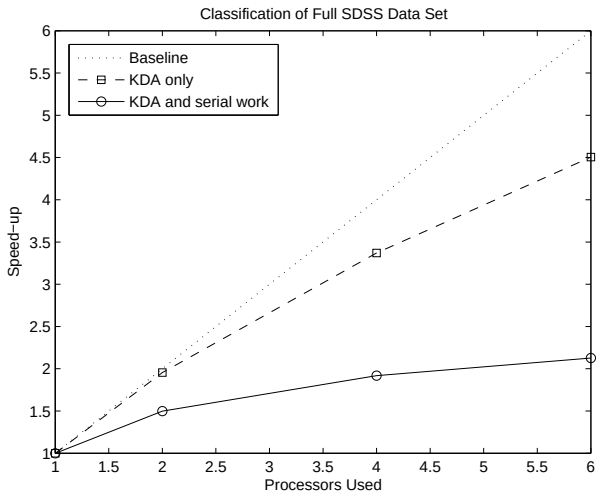

Figure 4: Speed-up for parallelized classification of the full 40m SDSS data set. Excluding data access and tree building shows that the underlying computation has workable speed-up. Future parallelization of tree building may then make this feasible.

features (these would require scaling or a specialized kernel), some preprocessing is required. We normalize each dimension to have unit variance and then perform PCA to reduce the dimensionality to 8. Further, as implemented, our algorithm only supports two-class problems, so we will train our classifiers to distinguish one class from the rest of the data; results are shown for distinguishing the largest class, lodgepole pines, which constitutes nearly half of the data, and the smallest class, cottonwoods and willows, which constitutes only 0.5% of the data.

For lodgepole pines, we optimized bandwidths with three 10-by-10 runs of the multi-bandwidth algorithm, taking 609, 135, and 92 seconds when parallelized over two-processors. The resulting classification accuracy is 90% both within and outside the class. For cottonwoods and willows, the same optimization procedure takes 1037, 248, and 213 seconds and results in a within-class accuracy of 99.4% and 97.7% outside. The estimated running time of the naive algorithm for a single bandwidth combination for the identification of either class is 10.6 hours, suggesting an overall time commitment of 320 hours for the above work, around seven orders of magnitude slower.

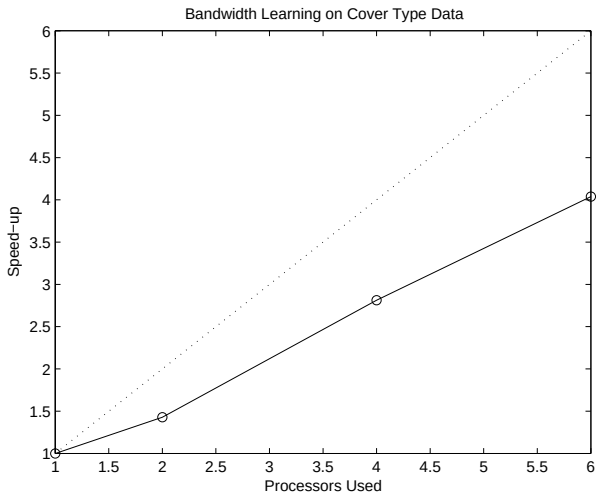

Figure 5: Speed-up for parallelized classification of LOO CV with the multi-bandwidth algorithm. This is more favorable because the multi-bandwidth algorithm performs a larger proportion of parallelizable work than does single-bandwidth classification.

Parallelization of LOO CV with the multi-bandwidth algorithm was much more favorable than with the large SDSS query set. This makes sense given Amdahl's law because the ratio of serial work to parallel work is significantly reduced: more parallel kernel summations and classification tests are performed for the serial act of tree-building. Figure 5 shows workable speed-up, suggesting that iterations of the multi-bandwidth algorithm, especially the long-running ones seen at higher dimensions, still benefit greatly from parallelization.

4.8 Conclusion

We have demonstrated numerous enhancements to the previous best algorithm for KDA. The hybrid expansion pattern and inclusion/exclusion pruning have resulted in very clear gains in asymptotic running time. Further, the multi-bandwidth version of the algorithm and parallelization enables rapid bandwidth learning for use in classification or determination of accurate class probabilities. We have significantly extended the size of problems now tractable under this analysis, arriving at results

in minutes where previously it would have taken weeks.

Our algorithm's orders of magnitude speed-up is obtained with a minimal degree of parallelization. Through the use of a more advanced data distribution and tree building method, we can expect parallelization to yield even more impressive gains. Also, for higher-dimensional problems, techniques such as logistic regression may be used in combination with KDA [26].

Already, an earlier rendition of our quasar catalog has been used to confirm the cosmic magnification effect predicted by the theory of relativity [58]. The new, larger catalog that we have found will undoubtedly lead to many other impressive discoveries. Nonetheless, these are results from one data set in one scientific application. KDA has no special dependence upon quasar identification, and in fact, the general ideas behind our fast algorithm have no special dependence on KDA [25]. Methods like those demonstrated in this chapter have potential to accelerate a wide variety of computations across science and data mining.

Table 5: Running times in seconds of LOO CV on one bandwidth combination for the various algorithms. Plotted on log-log axes in Figure 2.

N	Naive	Heap	Hp.Ep.	Hybrid	**Hy.Ep.**
1875	0.344	0.153	0.113	0.088	**0.043**
3750	1.11	0.408	0.250	0.115	**0.055**
7500	4.17	1.17	0.603	0.283	**0.111**
15k	16.3	4.75	1.74	0.884	**0.236**
30k	65.5	19.3	4.68	3.32	**0.540**
60k	264	78.4	17.9	13.6	**1.13**
120k	1050	199	63.2	54.5	**2.38**
240k	4200	1340	230	202	**5.15**
480k	—	5900	2690	716	**10.8**

APPENDIX A

ERRORS IN THE COVER-TREE PAPER

The single-tree proof assumes that every iteration splits an ancestor of the final set, but this is false because iterations might just work on nodes that ultimately prune before the final set. Thus, the depth-bound on the tree does not (directly) bound the number of iterations.

The extended version of [6] describes the same fast dual-tree algorithm used in this thesis. It gives a proof that its run time is $O(c^{16}N)$, though it elides most detail and makes claims that seem wrong. The 16 is the single-tree proof's c^{12} with four more factors of c derived from a bounds-loosening argument. Specifically, the pruning radius is increased by the radius of the query node, which means that the cover set can be c^2 larger, and the single-tree proof uses the size of the cover set squared. The transition from $O(\log N)$ to $O(N)$ is a bit more hand-wavy, but it seems to stem from the fact that there are only $O(N)$ explicit query nodes in the query tree. This, however, does not appear to count all the work the algorithm might ultimately have to do: the proof follows up with a claim that expanding the cover set to the next deeper level happens just once for each explicit query node, but this is false. It does happen just once for each *implicit* query node, but there is no bound on the number of those. It can happen more than once for an explicit query node if ever the query node does not split for a number of depths but its cover set still does, and this is most easily seen in the case of a shallow query leaf. That the cover set contains the query node in the monochromatic case does not matter; other reference nodes can still split when the query node and its counterpart do not.

A.1 Cover-trees can have shallow leaves

Consider a 1D tree with its first point at the origin. The second point (our leaf) is at just less than 1, so it becomes an immediate child of root at depth 0, with its own first depth being -1. The self-child of root at depth -1 can have immediate children all the way out to 0.5, and the leaf can have immediate children starting from just below 0.5 and up. Now imagine adding a third point, at just less than 0.5, outside the reach of our leaf. This must then be assigned as a child of root's self-child. We can now add a point at just less than 0.75, which while it could be a descendant of the leaf (and would have to be if it was not for the other point), it must be assigned as the third point's child. The new point *does* have to lie outside the leaf's position minus 2^{-2}, lest the point could be assigned as the leaf's self-child's child. Regardless, we can repeat this process as long as we want, with each next point just beyond the leaf's self-decedent's reach but within reach of the previous point. The epsilons we use to force decisions can of course be as small as we want, which means we can place a point arbitrarily close to a leaf in an ordinarily constructed cover-tree. Without additional points, this dataset has an unfavorable expansion constant, but we can add more to the non-leaf branch to help balance things out.

So, there is no minimum distance between leaves and other nodes in a cover-tree. Further, it is entirely possible for a non-leaf to skip many implicit depths before it splits again. The reference tree (even if equal to the query tree) may then have to perform numerous iterations before the next query split and vice versa.

APPENDIX B

GRID TREE LIMITATIONS

The expansion constant does not bound grid tree branching factors. The reasoning is that you can construct a dataset with small expansion constant for arbitrary dimension that nonetheless produces grid tree nodes with 2^D children. The dataset is constructed:

- Choose some $0 < \epsilon \ll 2^{-D}$. We will use doublings of ϵ (i.e. $2^i\epsilon$) for distances between various groups of points.
- We will place three groups of points: one near the origin, one near vector 1, and one near vector 2, where vector x is defined as the point with all values equal to x.
- Each group of points has 2^D members which form all combinations of high and low values for each dimension as given:
  - Near the origin, low is vector 0 (the origin) and high is given by $2^i\epsilon$ for dimensions enumerated i from 0 to $D-1$.
  - Near vector 2, low is $2 - 2^i\epsilon$ and high is vector 2.
  - Near vector 1, low and high are $1 \mp 2^{i-1}\epsilon$, respectively.
- This forces the grid tree to range from the origin to vector 2, which will center its first split at vector 1.[1]

[1]Technically, the tree could choose its root node arbitrarily, but often they are defined as exactly bounding the data. This proof is engineered to show that this common bounding technique fails miserably; further, it is enough to show that even one choice of bounds can fail, as the described groups of points can occur anywhere in the dataset.

- Observe that every single point in the group near vector 1 will go into a distinct child of the root node, and thus all 2^D possible children are formed.
- Further, points are arranged with the expansion constant in mind:
  - Any ball with radius at least $2^i\epsilon$ centered on a point contains all points in the same group with matching values at all dimensions i and up, for a minimum of 2^i included points. This is easily shown via the triangle inequality.
  - Any ball with radius less than $2^i\epsilon$ centered on a point contains no points with values that disagree at dimension i or higher, for a maximum of 2^i included points.
  - Thus, any radius in the range $[2^i\epsilon, 2^{i+1}\epsilon)$, for i from 0 to $D-1$ has at at least 2^i points and doubling it yields at most 2^{i+2}, and thus the expansion constant in this range of radius is bounded by 4.
  - For smaller radii, balls can only contain their central point and doubling them can only add one more, for maximum expansion constant of 2.
  - For larger radii, balls always contain the entire group and doubling them adds at most both other groups, for a maximum expansion constant of 3.
- Thus, the expansion constant is at most 4 for this dataset, and yet the grid tree is forced to expand a node into all 2^D of its possible children.

Further, we can go on to construct an arbitrarily complex dataset by replacing each point in the above with a similar group of 2^D points for $\epsilon' \ll 2^{-D}\epsilon$. This creates trees with endless streams of monster splits.

REFERENCES

[1] Amendola, L. and Tsujikawa, S., *Dark energy: theory and observations*. Cambridge University Press, 2010.

[2] Appel, A., "An efficient program for many-body simulations," *SIAM Journal on Scientific and Statistical Computing*, vol. 6, no. 1, pp. 85–103, 1985.

[3] Asuncion, A. and Newman, D., "UCI machine learning repository," 2007.

[4] Barnes, J. and Hut, P., "A hierarchical $O(N \log N)$ force-calculation algorithm," *Nature*, vol. 324, 1986.

[5] Bentley, J. L., "Multidimensional divide-and-conquer," *Communications of the ACM*, vol. 23, no. 4, pp. 214–229, 1980.

[6] Beygelzimer, A., Kakade, S., and Langford, J., "Cover trees for nearest neighbor," in *ICML*, pp. 97–104, 2006.

[7] Blackard, J. A., *Comparison of Neural Networks and Discriminant Analysis in Predicting Forest Cover Types*. PhD dissertation, Colorado State University, Department of Forest Sciences, 1998.

[8] Boyd, J. P., "The uselessness of the fast Gauss transform for summing Gaussian radial basis function series," *Journal of Computational Physics*, vol. 229, no. 4, pp. 1311–1326, 2010.

[9] Brinkhoff, T., Kriegel, H.-P., and Seeger, B., "Efficient processing of spatial joins using R-trees," *SIGMOD Rec.*, vol. 22, pp. 237–246, June 1993.

[10] Callahan, P. B. and Kosaraju, S. R., "A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields," *Journal of the ACM (JACM)*, vol. 42, no. 1, pp. 67–90, 1995.

[11] Callahan, P., *Dealing with Higher Dimensions: The Well-Separated Pair Decomposition and its Applications*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, 1995.

[12] Cipra, B. A., "The best of the 20th century: Editors name top 10 algorithms," *SIAM news*, vol. 33, no. 4, pp. 1–2, 2000.

[13] Clarkson, K. L., "Nearest neighbor queries in metric spaces," *Discrete & Computational Geometry*, vol. 22, no. 1, pp. 63–93, 1999.

[14] Clarkson, K. L., "Nearest-neighbor searching and metric space dimensions," *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*, pp. 15–59, 2006.

[15] Dobkin, D. and Lipton, R. J., "Multidimensional searching problems," *SIAM Journal on Computing*, vol. 5, no. 2, pp. 181–186, 1976.

[16] Elgammal, A., Duraiswami, R., Harwood, D., and Davis, L. S., "Background and foreground modeling using nonparametric kernel density estimation for visual surveillance," *Proceedings of the IEEE*, vol. 90, no. 7, pp. 1151–1163, 2002.

[17] Epanechnikov, V. A., "Non-parametric estimation of a multivariate probability density," *Theory of Probability & Its Applications*, vol. 14, no. 1, pp. 153–158, 1969.

[18] Ertöz, L., Steinbach, M., and Kumar, V., "Finding clusters of different sizes, shapes, and densities in noisy, high dimensional data," in *SIAM international conference on data mining*, vol. 47, 2003.

[19] Esselink, K., "The order of Appel's algorithm," *Information Processing Letters*, vol. 41, no. 3, pp. 141–147, 1992.

[20] Fischer, J., Mayer, C., and Söding, J., "Prediction of protein functional residues from sequence by probability density estimation," *Bioinformatics*, vol. 24, no. 5, pp. 613–620, 2008.

[21] Fix, E. and Hodges, J. L., "Discriminatory analysis, Nonparametric regression: consistency properties," tech. rep., USAF School of Aviation Medicine, 1951.

[22] Frey, B. and Dueck, D., "Clustering by passing messages between data points," *Science*, vol. 315, no. 5814, pp. 972–976, 2007.

[23] Friedman, J. H., Bentley, J. L., and Finkel, R. A., "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software (TOMS)*, vol. 3, no. 3, pp. 209–226, 1977.

[24] Gaede, V. and Günther, O., "Multidimensional access methods," *ACM Comput. Surv.*, vol. 30, pp. 170–231, June 1998.

[25] Gray, A. and Moore, A., "N-body problems in statistical learning," in *NIPS*, 2000.

[26] Gray, A., Komarek, P., Liu, T., and Moore, A., "High-dimensional probabilistic classification for drug discovery," in *COMPSTAT 2004*, 2004.

[27] Gray, A. and Moore, A., "Nonparametric density estimation: Toward computational tractability," in *SIAM Data Mining*, 2003.

[28] Gray, A. and Moore, A., "Rapid Evaluation of Multiple Density Models," in *Artificial Intelligence and Statistics 2003*, 2003.

[29] Gray, A. and Riegel, R., "Large-scale kernel discriminant analysis with application to quasar discovery," in *Compstat*, 2006.

[30] Greengard, L. and Rokhlin, V., "A fast algorithm for particle simulations," *Journal of Computational Physics*, vol. 73, pp. 325–248, 1987.

[31] Greengard, L., *The rapid evaluation of potential fields in particle systems*. 1988.

[32] Greengard, L. and Strain, J., "The fast Gauss transform," *SIAM Journal on Scientific and Statistical Computing*, vol. 12, no. 1, pp. 79–94, 1991.

[33] Hastie, T., Tibshirani, R., and Friedman, J., *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2001.

[34] Holmes, M., Gray, A., and Isbell, C., "Ultrafast Monte Carlo for kernel estimators and generalized statistical summations," in *NIPS*, 2008.

[35] Holmes, M. P., Gray, A. G., and Isbell, C. L., "Fast kernel conditional density estimation: A dual-tree monte carlo approach," *Computational Statistics & Data Analysis*, vol. 54, no. 7, pp. 1707–1718, 2010.

[36] Holmes, M. P., Gray, A. G., Isbell, C. L., and Tech, G., "QUIC-SVD: Fast SVD using cosine trees," *Advances in Neural Information Processing Systems*, vol. 21, 2008.

[37] Indyk, P. and Motwani, R., "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pp. 604–613, ACM, 1998.

[38] Karger, D. R. and Ruhl, M., "Finding nearest neighbors in growth-restricted metrics," in *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pp. 741–750, ACM, 2002.

[39] Krauthgamer, R. and Lee, J. R., "Navigating nets: simple algorithms for proximity search," in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 798–807, Society for Industrial and Applied Mathematics, 2004.

[40] Krauthgamer, R. and Lee, J. R., "The black-box complexity of nearest-neighbor search," *Theoretical Computer Science*, vol. 348, no. 2, pp. 262–276, 2005.

[41] Lee, D. and Gray, A., "Faster Gaussian summation: Theory and experiment," in *UAI*, 2006.

[42] Lee, D. and Gray, A., "Fast high-dimensional kernel summations using the Monte Carlo multipole method," *Advances in Neural Information Processing Systems*, vol. 21, 2008.

[43] Lee, D., Gray, A., and Moore, A., "Dual-tree fast Gauss transforms," in *NIPS*, pp. 747–754, 2006.

[44] Liu, T., Moore, A., and Gray, A., "New algorithms for efficient high-dimensional nonparametric classification," *J. Mach. Learn. Res.*, vol. 7, pp. 1135–1158, 2006.

[45] Liu, T., Rosenberg, C., and Rowley, H. A., "Clustering billions of images with large scale nearest neighbor search," in *Applications of Computer Vision, 2007. WACV'07. IEEE Workshop on*, pp. 28–28, IEEE, 2007.

[46] March, W. B., Connolly, A. J., and Gray, A. G., "Fast algorithms for comprehensive n-point correlation estimates," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1478–1486, ACM, 2012.

[47] March, W. B. and others, "Optimizing the computation of n-point correlations on large-scale astronomical data," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 74, IEEE Computer Society Press, 2012.

[48] McBride, C. K. and others, "Three-point correlation functions of sdss galaxies: constraining galaxy-mass bias," *The Astrophysical Journal*, vol. 739, no. 2, p. 85, 2011.

[49] McBride, C. K. and others, "Three-point correlation functions of sdss galaxies: Luminosity and color dependence in redshift and projected space," *The Astrophysical Journal*, vol. 726, no. 1, p. 13, 2011.

[50] Mittal, A. and Paragios, N., "Motion-based background subtraction using adaptive kernel density estimation," in *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, vol. 2, pp. II–302–II–309, IEEE, 2004.

[51] Moore, A. and others, "Fast algorithms and efficient statistics: N-point correlation functions," in *Proceedings of MPA/MPE/ESO Conference Mining the Sky, July 31–August 4, Garching, Germany*, 2000.

[52] Murakami, Y. and Mizuguchi, K., "Applying the naïve Bayes classifier with kernel density estimation to the prediction of protein–protein interaction sites," *Bioinformatics*, vol. 26, no. 15, pp. 1841–1848, 2010.

[53] Nadaraya, E. A., "On estimating regression," *Theory of Probability & Its Applications*, vol. 9, no. 1, pp. 141–142, 1964.

[54] Nichol, R. and others, "The effect of large-scale structure on the sdss galaxy three-point correlation function," *Monthly Notices of the Royal Astronomical Society*, vol. 368, no. 4, pp. 1507–1514, 2006.

[55] Pagel, B.-U., Six, H.-W., and Winter, M., "Window query-optimal clustering of spatial objects," in *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '95, (New York, NY, USA), pp. 86–94, ACM, 1995.

[56] Peebles, P. J. E., *The large-scale structure of the universe*. Princeton university press, 1980.

[57] Pelleg, D. and Moore, A., "Accelerating exact k-means algorithms with geometric reasoning," in *KDD*, pp. 277–281, 1999.

[58] Peplow, M., "Quasars reveal cosmic magnification," *Nature*, April 27 2005.

[59] Preparata, F. P. and Shamos, M., *Computational Geometry*. Springer-Verlag, 1985.

[60] Procopiuc, O., Agarwal, P. K., Arge, L., and Vittery, J. S., "Bkd-tree: A dynamic scalable kd-tree," in *International Symposium on Spatial and Temporal Databases*, pp. 46–65, 2003.

[61] Ram, P. and Gray, A. G., "Maximum inner-product search using cone trees," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 931–939, ACM, 2012.

[62] Ram, P., Lee, D., March, W., and Gray, A. G., "Linear-time algorithms for pairwise statistical problems," *Advances in Neural Information Processing Systems*, vol. 22, pp. 1527–1535, 2009.

[63] Rao, C. R., *Linear Statistical Inference*. Wiley, 1973.

[64] Richards, G., Nichol, R., Gray, A., Brunner, R., and Lupton, R., "Efficient Photometric Selection of Quasars from the Sloan Digital Sky Survey: 100,000 $z > 3$ Quasars from Data Release One," *Astrophysical Journal Supplement Series*, vol. 155, pp. 257–269, 2004.

[65] Riegel, R., Gray, A., and Richards, G., "Massive-scale kernel discriminant analysis: Mining for quasars," in *SIAM Data Mining*, 2008.

[66] Roweis, S. T. and Saul, L. K., "Nonlinear dimensionality reduction by locally linear embedding," *Science*, vol. 290, no. 5500, pp. 2323–2326, 2000.

[67] Shamos, M. I., "Geometric complexity," in *Proceedings of seventh annual ACM symposium on Theory of computing*, pp. 224–233, ACM, 1975.

[68] Silverman, B. W., *Density Estimation for Statistics and Data Analysis*. Chapman and Hall/CRC, 1986.

[69] Silverman, B., "Algorithm AS 176: Kernel density estimation using the fast fourier transform," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 31, no. 1, pp. 93–99, 1982.

[70] Struyf, A., Hubert, M., and Rousseeuw, P., "Clustering in an object-oriented environment," *Journal of Statistical Software*, vol. 1, no. 4, pp. 1–30, 1996.

[71] Szapudi, I. and others, "Fast cosmic microwave background analyses via correlation functions," *The Astrophysical Journal Letters*, vol. 548, no. 2, p. L115, 2001.

[72] Thakar, A. R., Szalay, A., Fekete, G., and Gray, J., "The catalog archive server database management system," *Computing in Science & Engineering*, vol. 10, no. 1, pp. 30–37, 2008.

[73] Vaidya, P. M., "An $O(n \log n)$ algorithm for the all-nearest-neighbors problem," *Discrete & Computational Geometry*, vol. 4, no. 1, pp. 101–115, 1989.

[74] Vasiloglou, N., Gray, A. G., and Anderson, D. V., "Scalable semidefinite manifold learning," in *Machine Learning for Signal Processing, 2008. MLSP 2008. IEEE Workshop on*, pp. 368–373, IEEE, 2008.

[75] Vengroff, D. E., "A transparent parallel I/O environment," in *DAGS Symposium on Parallel Computation*, pp. 117–134, 1994.

[76] Wang, P., Lee, D., Gray, A., and Rehg, J., "Fast mean shift with accurate and stable convergence," in *Workshop on Artificial Intelligence and Statistics (AISTATS)*, 2007.

[77] Watson, G. S., "Smooth regression analysis," *Sankhyā: The Indian Journal of Statistics, Series A*, pp. 359–372, 1964.

[78] Yang, C., Duraiswami, R., Gumerov, N. A., and Davis, L., "Improved fast Gauss transform and efficient kernel density estimation," in *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pp. 664–671, IEEE, 2003.

[79] York, D. G. and others, "The sloan digital sky survey: Technical summary," *The Astronomical Journal*, vol. 120, no. 3, pp. 1579–1587, 2000.

[80] Zhang, J., Mamoulis, N., Papadias, D., and Tao, Y., "All-nearest-neighbors queries in spatial databases," in *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, pp. 297–306, June 2004.