DYNAMIC DATA STRUCTURES FOR RANDOMIZED ALGORITHMS THAT USE SAMPLING

by

Deganit Armon

Department of Computer Science
Duke University

Date: 12/11/97

Approved:

John H. Reif, Supervisor

Robert A. Wagner

Pankaj Kumar Agarwal

Michael L. Littman

Richard Hodel

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

1997

ABSTRACT

(Engineering—Mechanical)

DYNAMIC DATA STRUCTURES FOR RANDOMIZED ALGORITHMS THAT USE SAMPLING

by

Deganit Armon

Department of Computer Science
Duke University

Date: 12/11/97

Approved:

John H. Reif, Supervisor

Robert A. Wagner

Pankaj Kumar Agarwal

Michael L. Littman

Richard Hodel

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University

1997

Abstract

This dissertation presents a new technique for transforming algorithms that use random sampling into dynamic algorithms. We design a dynamic data structure for storing the input to the algorithm, which may be completely or partially rebuilt when the input is updated. Rebuilding is independent of the value of the new input; rather, the function used in the random sampling is used in determining when the data structure is rebuilt.

Our technique is simple, general, and can be applied to a wide range of data structures. It yields polylogarithmic expected time bounds on updates for several classes of data structures and algorithms.

A well known problem of randomized algorithms is that they do not always attain their expected time bounds. We address this problem by developing a method for transforming expected time bounds to high likelihood bounds. We do this by using multiple independent processes, which we call *replicants*, each maintaining its own dynamic data structure. We describe a general framework for determining how many replicants are necessary to ensure high likelihood time bounds, given the algorithm time bounds and sampling function.

To test our methods, we applied them to a randomized binary search tree, using two different models of our data structure. The empirical results from our testing were consistent with, and in some cases better than, the theoretically predicted bounds. Our code is modular, and the binary search tree module can be replaced by a code module for another data structure.

We applied our methods to the problem of finding sphere separators for a neighborhood system and its induced graph. Using our methods, sphere separators can be maintained with an expected update time of $O(\log n)$, and a high likelihood update

time of $O(\log^3 n)$. Moreover, a separator decomposition tree can be maintained in $O(\log^3 n)$ time per update. These are the best know results for dynamic maintenance of shpere separators.

An important application of graph separators is *nested dissection*, a widely used technique for solving sparse linear systems. We present here some new results improving the space and time bounds of parallel implementations of nested dissection.

Acknowledgements

This thesis would not have been possible without the guidance and support of my advisor, Dr. John Reif. He ignited the spark that started this work, and kept the flame going even when my family and teaching commitments threatened to take over all available time and then some. His amazing breadth of knowledge was a constant inspiration, always guiding me in interesting directions.

The Computer Science Department at the University of California, Riverside, and especially Dr. Tom Payne were very supportive during the final months of writing this dissertation, allowing me to take time off from teaching so that I could complete it. I would also like to thank Dr. Teodor Przymusinski for several motivational talks over the years. My friend Dr. Davida Fischman of California State San Bernardino provided invaluable editorial comments.

My students John Cleary and Tim Jenkins implemented and tested the basic data structures. By asking the right questions they forced me to think very carefully about what I was trying to say. As a result, the presentation in the early chapters is much clearer than it would otherwise have been.

Finally, I would like to thank my family – my husband Carmel and my daughters Orit and Efrat. They provided distractions when I needed them and stayed out of my way when I did not. I dedicate this dissertation to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An algorithm is a sequence of computational steps that transform an input set into an output set [16]. This definition assumes that the entire input set is known before the output is calculated. A *dynamic* algorithm receives input on an ongoing basis, modifying its output accordingly. Dynamic algorithms are useful if they can modify their output using significantly less time and effort than it would take to compute the new output "from scratch".

For example, the solution to a routing problem in a network may need to be modified when a node is added to or removed from the network. An efficient dynamic algorithm will find a new solution with small computational cost. Dynamic algorithms are very useful in interactive applications, such as network optimization, VLSI, computer graphics, and computational geometry.

In this thesis, we develop dynamic algorithms for an important class of algorithms: the class of randomized algorithms that use sampling. We show a general technique for creating dynamic data structures for this class of algorithms. These data structures can then be used to create dynamic versions of the algorithms. Our technique is simple, general and modular, and can be applied to the data structure of any randomized algorithm that uses sampling.

1.1 Dynamic Algorithms

In a *static* algorithm, all data points are known before processing begins, and the problem is solved once. The static algorithm uses its knowledge of the entire input when constructing its solution. *Dynamic algorithms*, on the other hand, process input as it is presented, modifying their solution to account for the new input.

Consider again the network routing problem. Given a network, a static algorithm can find a route between two nodes. What happens when a node is added to the

network? How do we find the new solution? The naive, brute-force way would be to start over, using as input a new data set, i.e., the old data set plus the new point, and rerun the static algorithm. This approach makes sense only if additions to the data are infrequent occurrences. If changes are frequent, this may not be efficient.

We can do better by taking into account the structure of the solution found so far, or the structure of the algorithm. In other words, we would like to exploit the path we took on the way to finding the original solution, retracing only as little of it as necessary to find the new solution. The savings in time to construct the updated solution could be significant.

1.1.1 Dynamic Data Structures

A key to designing an efficient dynamic algorithm is the data structure used to store the input. *Dynamic data structures* support updates to the data, i.e., insertion and deletion of data points, in a way that allows efficient searches through the data at any point, after any number of updates. *Semi-dynamic structures* support updates in the form of insertions or deletions, but not both. There are many efficient semi-dynamic data structures described in the literature. Saxe and Bentley [70, 7] describe techniques for transforming static data structures to dynamic structures that support insertions. They also examine transformation of data structures that support only deletions, and show a special case where certain subclasses of problems can be transformed into data structures that support both insertions and deletions. They also prove that it is not possible to find a *general* efficient transformation for totally dynamic data structures.

A special problem of dynamic data structures and algorithms is that deletions are harder to deal with than insertions. Overmars and van Leeuwen [52, 53] describe dynamization of data structures to support deletions if the static data structure satisfies certain assumptions, but these results do not generalize to all problems. Often, efficient specialized algorithms that support insertions to the data cannot be modified to support deletions. For example, Preparata describes an optimal semi-

dynamic convex hull algorithm that supports only insertions, which finds the new hull in $O(\log n)$ time per update [62]. If deletions are also considered, Overmars and van Leeuwen give the best known fully dynamic algorithm, which performs updates in $O(\log^2 n)$ time [51].

When designing dynamic data structures, we would like to know that the shape of the structure conforms to certain parameters, placing bounds on the time it takes to access and update it. Many updates to the data may cause the data structure to be lopsided, greatly extending the time necessary to perform updates and searches. A trivial example is a binary search tree which degenerates to a list if elements are inserted in order. Whereas a random tree has expected depth $O(\log n)$, worst-case trees have $O(n)$ depth. Simple binary search trees have problems even with random data. For example, in a binary search tree where deletions favor one side of the tree, it has been shown that if we alternate insertions and deletions $\Theta(n^2)$ times, then the trees will have an expected depth of $\Theta(\sqrt{n})$ [17, 18].

One way of bounding the update time is to make sure that the structure stays balanced after every update. Examples of data structures that employ this technique include AVL trees, B-trees, 2-3 trees, and red-black trees, all of which have worst-case $O(\log n)$ time per update and search operations. Detailed descriptions can be found in many data structures and algorithms textbooks [3, 34, 16].

An alternative solution is to use a data structure in which some operations may take longer, but their cost is amortized over other operations. Sleator and Tarjan introduced splay trees [72] and self-adjusting heaps [73], which have good amortized performance. Mulmuley [45] suggested a technique called "lazy balancing", where nodes deleted from the search structure are not actually removed but only marked. When the number of marked nodes exceeds some threshold, the whole structure is rebuilt. This kind of balancing does not guarantee the search time through the structure, but it does guarantee that long searches are only a polynomially small fraction of all searches.

1.1.2 Creating Dynamic Data Structures

Dynamic structures can be based on static structures, or they can be totally new entities. In this work we develop our dynamic data structures from static data structures by a process of dynamization. The static data structures that we consider are constructed using a random sample of the input. In the next section, we discuss randomized algorithms and algorithms that use random sampling.

1.2 Randomized Algorithms

Throughout the history of algorithms, starting with Euclid's greatest common divisor algorithm and continuing into the 1970s, algorithms have been deterministic. In a *deterministic algorithm*, each step leads to the next in a predetermined way; it is always possible to anticipate the next step that the algorithm will make. In 1976, Rabin [64] and, independently, Solovay and Strassen [74] introduced *randomized algorithms*. Randomized algorithms take steps that depend on random choices, and so cannot be anticipated in advance. The making of the random choices is part of the algorithm, and these choices are made based on random bits in the computer.

Deterministic algorithms may have bad behavior under certain distributions of the data. For example, deterministic quicksort, as introduced by Hoare [33], has complexity $O(n \log n)$ when the data is drawn from a random distribution, but has worst-case behavior of $O(n^2)$ if the data is already sorted. In a randomized version of quicksort, the pivot is selected at random. This algorithm may also exhibit worse case behavior, but it occurrence does not depend on the distribution of the data. Rather, the algorithm's performance depends on the random choices it makes. This is true of other randomized algorithms as well. In fact, running a randomized algorithm several times on the same data may result in different running times.

1.2.1 Classes of Randomized Algorithms

There are two broad classes of randomized algorithms. The first type, generally known as *Las Vegas*, always yields a correct answer but its running time is a random variable with a specified mean. Some Las Vegas algorithms take an extremely long time to run on certain data. Quicksort falls into this category, as it always sorts the input with mean running time $O(n \log n)$, but the running time could be as bad as $O(n^2)$. Many computational geometry algorithms also fall into this category [47]. Algorithms of the second type, known as *Monte Carlo*, are guaranteed to terminate but output the correct answer only with some probability. One example is Rabin's primality algorithm [16].

1.2.2 Algorithms that Use Random Sampling

There are a number of ways in which randomization can be used in an algorithm. In this thesis we will concentrate on randomized algorithms that use *random sampling* of their input. A truly random sample should give useful information about the entire input set. In particular, a solution for a random sample of the input may provide a useful first approximation for a solution for the entire set. Or, a random sample may be useful in partitioning the sampled set into subsets of roughly equal size, thus setting the stage for a divide-and-conquer algorithm.

Computational geometry problems seem particularly suited to be solved by algorithms that use random sampling. Clarkson [10, 12] has shown that random sampling is useful in developing efficient algorithms for searching arrangements, constructing order-k Voronoi diagrams, determining separations of polytopes [10], trapezoidal diagrams and line segment intersections, convex hulls, spherical intersections and diametrical pairs, and range reporting [12]. The algorithms are all Las Vegas (that is, the solution output is always correct and the time bounds are expected), and the construction is randomized incremental. Mulmuley's book [47] contains many other examples.

Randomized data structures and algorithms that use sampling lend themselves

nicely to our general dynamizing technique.

1.3 Making Randomized Algorithms Dynamic

We present a technique for dynamizing randomized algorithms that use sampling. The key idea is to rebuild the data structure from time to time, using the static algorithm. The algorithm tosses a weighted coin and, based on the result of the toss, decides whether to rebuild the structure. The weighting of the coin is a function of the size of the sample used by the static algorithm. The expected time per update is a function of the time it takes to run the static algorithm and the size of the sample it selects.

1.3.1 Description of Our Model

The algorithms considered here maintain a data structure by processing insertions and deletions of data points into this structure. The algorithms can then use the structure to answer queries.

The input to the dynamic algorithm is a sequence of data points, called the *input stream*. The stream is generated by an adversary who knows the algorithm, but not the random choices that it will make. Associated with each point p in the stream is one of the following requests:

1. INSERT p into the data structure.
2. DELETE p from the data structure.
3. Answer a QUERY about p.

In what follows, we will refer to the INSERT and DELETE operations as *updates*. QUERY operations can be simple membership queries, e.g., determining if an element is in the structure, or they can be requests for more complex information about the elements in the structure, such as whether an element belongs to a certain set, or describing a set that is defined by some or all of the elements in the structure.

Generally, dynamic algorithms are considered to have "good" time bounds if each of these operations can be completed in polylogarithmic time. A randomized dynamic algorithm is called *polylogarithmic*, or *polylog*, if the expected time to process a request is $(\log n)^{O(1)}$, where n is the size of the data set being updated. We will show in chapter 2 which data structures and sample sizes can attain a polylog update and query time.

To demonstrate our technique, we consider its application to the simple example of a binary search tree. Binary search trees hold an element in each node, and have the property that each node contains an element larger than that stored in its left child and smaller than that stored in its right child (if it has any). Consequently, every element in the left subtree is smaller than the element at the root, and every element in the right subtree is greater than the element stored at the root.

1.3.2 Balanced Search Structures

The purpose of search structures is to store data points in such a way that searches through the data are performed efficiently. If a structure is *balanced*, worst-case search time is minimized. In a tree, search time depends on the distance of the farthest leaves from the root. The time to traverse this distance is an upper bound on the time it takes to perform a single search. In a *perfectly balanced* binary search tree of n nodes, all paths from the root to a leaf have length $\log n$.

There are known techniques for maintaining balance in a search structure. Some, such as AVL trees [1], involve rebalancing the structure whenever it becomes unbalanced. In these and other similar structures, rebalancing is *data dependent*: whether or not rebalancing happens depends on the shape and content of the tree. Other techniques [9, 35, 39, 50, 52, 53], rebuild a structure after a certain number of updates to the structure. These techniques include *partial rebuilding*, a data dependent technique where rebuilding occurs when a subtree becomes unbalanced, and *global rebuilding*, a method that reconstructs the entire data structure periodically, after a fixed number of updates. The technique we develop in this thesis is different from

all of these in that rebuilding is independent of the data or the number of updates performed.

1.3.3 Static Construction

If the data to be stored in the search structure are known in advance, it is possible to construct perfectly balanced trees. One way to do this is to rearrange the data in a way that will yield a perfect structure. This requires a significant amount of preprocessing. For example, a perfectly balanced binary search tree can be constructed by finding the median of the data, storing it at the root, partitioning the data into two sets of elements smaller and larger than the median, then recursively storing the two subsets in the left and right subtrees.

If this amount of preprocessing is unacceptable, we can relax the requirement that the tree be perfectly balanced, and settle for "approximately balanced", within some parameters. We can use a randomized algorithm to build the search structure. Again using the binary tree example, this is equivalent to choosing a random element to be at the root. We note several points about this algorithm:

- Each time the algorithm is run, a different element may end up at the root. If there are n distinct data points, each one has probability $\frac{1}{n}$ of being at the root.
- It is possible for the algorithm to yield a worst-case tree, of depth n, by randomly choosing as root an extremum of the data, and doing so in each recursive call as well. It is also possible for the algorithm to yield a perfectly balanced tree, of depth $\log n$, by selecting the exact median at every stage. The probability of either occurring is very small.
- The average tree depth is $O(\log n)$ [44].

We now analyze the time required to create the binary search tree. This is identical to the work needed to sort n numbers using randomized quicksort. Once a pivot is

selected (each point equally likely to be chosen with probability $\frac{1}{n}$), the entire data set needs to be scanned to determine which subtree an item goes into, taking $\Theta(n)$ time. The total expected time to construct the tree, $T(n)$, can be expressed as a recurrence based on the time to construct each subtree:

$$T(n) = \Theta(n) + \frac{1}{n}\sum_{r=1}^{n}(T(r-1) + T(n-r))$$

$$= \Theta(n) + \frac{2}{n}\sum_{r=0}^{n-1}(T(r))$$

which is $O(n \log n)$. In the worst case, it takes quadratic time to build the tree. However, the formula above reflects *expected* time bounds, averaged over all possible cases.

1.3.4 Overview of Our Technique

In the previous section we focused on the static case, where processing takes place after the entire input is known. We now consider adding points to the input set dynamically. If we do not alter the structure of the existing tree, the new element is added as a leaf, and its place is uniquely determined by a path from the root. Assuming that the tree is balanced, the length of such a path is logarithmic in the number of elements in the tree. However, if we continue inserting elements into the tree in this manner, it might cease to be balanced; the depth of the tree would no longer be logarithmic in the number of nodes, and consequently the time to insert a new element could be greater than $O(\log n)$.

We avoid this scenario by occasionally rebuilding the tree or parts of it using the static algorithm. The probability of rebuilding the tree is $\frac{1}{n}$, which is the probability of any given point being selected to be at the root. If rebuilding occurs, the static algorithm is called and receives as input the original data set together with the new point. If rebuilding does not happen, the algorithm determines in which subtree the new point belongs. The entire process is recursively applied to the subtree.

The probability of rebuilding the tree in the example was $\frac{1}{n}$. In the more general case, if the data set is of size n, we denote by $p(n)$ the probability that an input point is included in the random sample selected by the static algorithm. The probability of rebuilding the structure in that case is $p(n)$.

Analysis

Chapter 2 contains a detailed analysis of the expected time to perform an update, given the time bounds on the static algorithm, the size of the random sample, and the size of the subtrees. Here we will perform the analysis for the simple example of the binary search tree.

As stated above, the probability of rebuilding the entire tree is $\frac{1}{n}$, and the time needed to rebuild the entire tree is $cn \log n$ where c is a constant. With the complementary probability, $\frac{n-1}{n}$, we recursively insert the new element into one of the subtrees. We further simplify the analysis here by that assuming that the tree is perfectly balanced and that the size of each subtree is $\lfloor n/2 \rfloor$.

The recurrence for the expected time bounds on inserting a new element into the tree is:

$$T(n) = \frac{1}{n} cn \log n + \frac{n-1}{n} T(\lfloor \frac{n}{2} \rfloor).$$

The solution to this recurrence is

$$T(n) = O(\log^2 n).$$

Of course, there are more efficient ways of inserting elements into a balanced binary tree, most involving rotation of subtrees. However, this analysis is for illustrative purposes only, and the technique yields good results for other algorithms, as described in later chapters.

1.3.5 Advantages of Our Technique

Our technique creates algorithms that have an important advantage over other, more specialized, dynamic algorithms: the identical procedure can be applied to the data structure to perform deletions, yielding the same time bounds as for insertions. This is in contrast to some of the more refined dynamic algorithms that can perform insertions efficiently but take longer to perform deletions.

Other advantages of our technique are its simplicity, modularity, and general applicability. We implemented and tested the dynamic randomized binary search tree described above, and with little effort, a different static, randomized algorithm that uses sampling can replace the static tree building module. The update modules remain the same, with the possible exception of changing the rebuilding probabilities.

This assertion about the generality of the technique is true whenever the data structure is used for storing data and answering membership queries. If more complex information needs to be extracted from the data structure, there may be some additional adjusting of the data structure after each update, as we see in Chapter 7.

Our technique does not always give the best possible asymptotic time bounds on performing updates, but it can be used to dynamize a static data structure quickly and easily. The example we use to illustrate the technique – the randomized binary search tree – has update bounds which are a logarithmic factor slower than other known methods for maintaining such a tree. In Chapter 5, we describe the problem of dynamically maintaining sphere separators for a point set, a problem for which our technique yields the best known dynamic time bounds.

1.4 High Likelihood Bounds

The dynamic algorithms that we create using our technique perform updates within the specified time bounds only with a certain probability. In other words, the algorithm can only guarantee *expected* time bounds. We can calculate the probability of being within these expected bounds by using Markov's inequality, which tells us that the probability of exceeding the time bounds by a factor of k is less than $\frac{1}{k}$ [44].

Often, expected time bounds are not good enough. As seen in empirical testing (see Chapter 4), our algorithm gives good mean update times, but the range exhibited is very wide. We describe a method for improving our algorithm so that there is a high likelihood that it attains its expected bounds.

We say that an algorithm attains a time bound *with high likelihood* if probability of attaining this bound is $1 - n^{\alpha}$ for some constant $\alpha > 1$, where n is the input size. According to Adleman and Manders [2], a randomized algorithm with success probability higher than $1 - 2^{-k}$ for some large constant k has a lower probability of failure than the hardware itself.

In Chapter 3 we describe a method for transforming the expected bounds of our dynamic algorithms into high likelihood time bounds. The main idea of our method is to perform the algorithm many times in parallel. Since we are dealing with sequential algorithms, the parallel execution does not involve parallel processors but rather a dovetailing of processes. This replication slows down the computation as a trade-off for attaining a bound with high likelihood.

Each process, which we call a *replicant*, maintains its own copy of the data structure. Given enough replicants, there is a high likelihood that at least one of the replicated structures is up-to-date with respect to the input stream, and will perform the next update within the desired time bounds.

The number of replicants necessary is a function of the desired high likelihood probability. Since attaining high likelihood time bounds, as defined above, means that the algorithm attains its time bounds with probability $1 - n^{\alpha}$, the number of replicants required can be calculated as a function of α.

1.5 Application: Sphere Separators

We apply our technique to the problem of finding sphere separators for a set of points and their induced graph. A *separator* of a graph is a small subset of its vertices which, when removed from the graph, separates it into subgraphs that are not connected. *Sphere separators* are the basis for finding separators for a large class

of graphs called *overlap graphs*, which includes planar graphs and k-neighborhood graphs. The original work on sphere separators was done by Miller, Teng, Vavasis *et al*, [43, 76], who gave linear time static randomized algorithms for finding sphere separators for a set of points in $\mathbf{R}^d$.

Applying our technique to this algorithm, we obtain dynamic algorithms which maintain separators for a dynamically changing graph. Our algorithm is polylog, i.e., it takes $(\log n)^{O(1)}$ expected sequential time per request to process worst-case queries and worst-case changes to the input set. We maintain a separator in expected time $O(\log n)$ and we maintain a separator decomposition tree in expected time $O(\log^3 n)$.

We use the replicant paradigm to transform the bounds on these algorithms into high likelihood time bounds, and show that we can maintain separators in time $O(\log^3 n)$ with high likelihood.

1.5.1 Applications of Separators: Nested Dissection

Our results on separators can be applied to generate dynamic algorithms for a wide variety of combinatorial and numerical problems, whose underlying associated dynamic graph is a k-neighborhood graph, such as monoid path problems and solving large linear systems.

A method for solving large linear systems that involves the use of separators on the underlying graph of the matrix is nested dissection. *Nested dissection* (ND) solves sparse $n \times n$ linear systems efficiently by imposing an order of elimination on the variables. *Parallel nested dissection* (PND) algorithms, which perform this variable elimination in parallel, have been developed for the PRAM model and for grid architectures. In Chapter 6 we give new results improving to the known PND algorithms. Our FAST PND algorithm speeds up previous PRAM PND algorithms by nearly a logarithmic factor; our COMPACT PND algorithm significantly decreases the space bounds needed for PND implemented on a mesh connected processor array; and our GENERALIZED COMPACT PND algorithm extends those result to a more general class of graphs. These algorithms generalize to solve all-pairs minimal-cost

path problems within the same complexity.

1.6 Organization of this Thesis

This thesis is divided into chapters as follows:

Chapter 1 provided an overview of the thesis. Chapter 2 contains basic definitions and a detailed description of the technique for dynamizing static algorithms. Chapter 3 introduces the replicant paradigm and how it can be used to attain high likelihood time bounds. In Chapter 4 we provide and analyze the results obtained from testing of the basic data structure, and of the replicant model. The code used in the testing can be found in the Appendix. In Chapter 5 we apply our technique to the algorithm of finding sphere separators for a set a points and their induced graphs. In Chapter 6 we describe the use of graph separators in nested dissection, and give new results improving the implementation of parallel nested dissection. An additional application of our technique, to a problem in computational geometry, is described Chapter 7. Chapter 8 summarizes and outlines some directions for future research.

Chapter 2

Making Randomized Algorithms Dynamic

In this chapter, we define our dynamic data structure formally and in detail. We describe a technique for transforming static algorithms that use sampling, so that they can cope with a dynamically changing input set. Our technique is simple and general and can be used to transform any static algorithm that uses a sample of the input in its construction into a dynamic algorithm, providing a "quick-and-dirty" method for obtaining a dynamic version of these algorithms.

As a running example, we use a randomized binary search tree. Though the theoretical update bounds predicted for this data structure are not as good as those attained by some other data structures, its simplicity makes it an easy example with which to illustrate our technique. The empirical results in Chapter 4 were obtained from this data structure.

We applied our technique to the problem of dynamically maintaining sphere separators and separator-based search structures, and attained the best results known for these problems (see Chapter 5). We also applied them to the problem of dynamically maintaining the convex hull of a set of points in the plane, but failed to improve on known time bounds for this problem. At the end of the chapter we list some classes of data structures and algorithms for which our technique is suitable, that is, using our technique they can be transformed to dynamic algorithms with good time bounds.

2.1 Definitions and Notation

We define a *static algorithm* to be an algorithm that is off-line, that is, the entire input is presented to the algorithm at once and does not change again throughout the execution of the algorithm. Let P be the input to the algorithm, and let $n = |P|$ be the size of the input, defined as the number of data points in P.

We give time bounds for static and dynamic algorithms. For dynamic algorithms,

we give expected and high likelihood time bounds. For an input set of size n, we use the following notation for the time bounds:

$T_S(n)$ denotes the time bound for processing the input by a static algorithm,

$T_D(n)$ denotes the expected time bound to perform an update to a data set of size n by a dynamic algorithm, and

$T_H(n)$ denotes the high likelihood time bound to perform an update to a data set of size n.

Let $\mathcal{A}$ be a static algorithm that uses sampling of its input. The algorithm may use the sample to partition the input set as part of a divide-and-conquer strategy, or it may use the sample to generate an approximate solution, or a first approximation to the solution that it will then refine. Let $S \subset P$ be the subset of the input which is the sample used by $\mathcal{A}$, and let $\sigma = |S|$. Typically, σ is a function of n, the size of the input, though it could also be a random variable. We will refer to this function as $\sigma(n)$. The sample is selected using unbiased, independent, random sampling. We define $p(n)$ to be the probability of selecting a point in the sample. For a sample of size $\sigma(n)$, this probability is $p(n) = \frac{\sigma(n)}{n}$. Let $T_S(n)$ be the time it takes $\mathcal{A}$ to process an input set P of size n.

If $\mathcal{A}$ is a *divide-and-conquer* algorithm, it partitions the input set into two or more subsets of roughly equal size. The algorithm then finds a solution for each of the subsets, and combines these solutions into a solution for the entire input set. The partitioning of the data is achieved by sampling the input set and using information from the sample S to create the partition. If the information from S partitions P into independent subsets of size at most δn, where $0 < \delta < 1$, then δ is called the *splitting ratio* of P.

An *approximation algorithm* calculates a near-optimal solution to a problem for its input set. In practice, near-optimality is often satisfactory. Approximation algorithms often select a subset of the input and use it to calculate an approximate

solution for the input set, or to calculate a first approximation which is then optimized by the algorithm.

A *dynamic algorithm* is an algorithm that allows changes to its input set P. Whenever a point is added to or removed from P, the dynamic algorithm modifies its solution to reflect the change in the input. We refer to changes to the input set as *updates*. There are two types of updates that we will consider: INSERT – adding a point to the input set – and DELETE – removing a point from it. The updates are presented to the algorithm in a *stream* of requests, which are processed in the order received. The time that a dynamic algorithm takes to perform an update is called the *update time* of the algorithm.

A *dynamic data structure* stores data points, and allows updates to the data. The data structures also support a SEARCH operation, for locating a data point in the structure. After any number of updates, dynamic data structures maintain their properties, such as the expected time to perform a SEARCH operation.

The process of converting static algorithms and data structures into dynamic ones is called *dynamizing*. In the following section we describe our technique for dynamizing static algorithms and the data structures that are created in the process.

2.2 Updating the Input Set

We wish to convert $\mathcal{A}$ from a static algorithm to a dynamic algorithm, while maintaining as much as possible the characteristics of $\mathcal{A}$. In particular, given a solution based on the input set, it should not be possible to determine whether it was generated by the static algorithm, or by the dynamic algorithm after a sequence of insertions and deletions.

Given an input set P of size n, we call the static algorithm $\mathcal{A}$ to generate a solution based on this input set in time $T_S(n)$. If the input set is slightly changed – a single point is added or removed – we would like to update the solution to reflect this change. A straightforward but inefficient solution would be to call $\mathcal{A}$ again with the modified input set, generating a solution in time $T_S(n+1)$ (for INSERT) or $T_S(n-1)$

(for DELETE).

Our goal is to process updates in significantly less time, and without recomputing an entire solution from scratch. Moreover, we wish the updated solution to be indistinguishable from a solution obtained by running the static algorithm $\mathcal{A}$ on the updated data set. By *indistinguishable*, we do not mean "identical". After all, we are dealing with randomized algorithms; running the static algorithm on the same data twice may produce different solutions, or different paths to the same solution. We do require the following:

> The output of the dynamic algorithm after an update to the input, and the output of the static algorithm given the updated set as input, both come from the same probability distribution of outputs.

2.2.1 A Simple Example

We begin with a simple example that will illustrate the idea behind our technique.

Let $\mathcal{A}_0$ be a static algorithm that does the following:

1. Select a random sample S of size $\sigma(n)$ from the input set P.
2. Perform some calculation on the points of S.
3. Output the result of step 2.

The important thing to notice about this algorithm is that the output depends solely on the sample selected. Now consider a new point p' presented to the algorithm. Consider the situation where all $n+1$ points are presented as input to $\mathcal{A}_0$. Would the output be different? If p' is not selected in the sample, its inclusion in the input set has no effect on the output of the algorithm!

For each input point, the probability of being included in the sample is $p(n+1) = \frac{\sigma(n+1)}{(n+1)}$. Specifically, the probability that the new point, p', is selected in the sample is $p(n+1)$. On the other hand, with probability $1-p(n+1)$, p' has no effect on the output.

This is precisely what our dynamic algorithm does when performing an INSERT operation: with probability $1 - p(n+1)$ it leaves the output unchanged, and with probability $p(n+1)$, it generates a new output. It does so by calling the static algorithm $\mathcal{A}_0$ as a subroutine, with the set $P \cup \{p'\}$ as its input.

It is important to note that when $\mathcal{A}_0$ is invoked by the dynamic algorithm, p' **may not be selected in the sample that determines the output.** If the dynamic algorithm produces a new output, this output is not dependent on p' unless p' is selected in S, something that only happens with probability $p(n+1)$. As a result, the *value* of p' does not affect the new output any more than any other input point. The requirement specified in section 2.2 is thus satisfied.

Analysis

How long does the dynamic algorithm take to perform this update? First, it needs to determine whether or not to invoke the static algorithm. It does so by performing a single Bernoulli trial with probability $1 - p(n+1)$ of success. This takes constant time to do. With this probability, it does nothing further. With the complementary probability, the static algorithm is invoked, taking $T_S(n+1)$ time to complete. The expected time for the dynamic algorithm to insert a new point into the data set can be calculated using the formula

$$E[T_D(n+1)] = p(n+1)T_S(n+1) + (1 - p(n+1))O(1).$$

Generalizing to all n, this can be simplified to

$$E[T_D(n)] = O(p(n)T_S(n)).$$

Processing DELETE Requests

The DELETE operation is slightly more complex, because there are two circumstances in which we invoke the static algorithm. Just as in the INSERT operation, we call $\mathcal{A}_0$ with probability $p(n) = \frac{\sigma(n)}{n}$. However, since the output depends on the

points of S, we also have to recalculate the output if the point to be deleted was included in the sample.

The practical consequence is that we need to keep track of which points are included in the sample. The analysis of the DELETE operation follows the same lines as the analysis of the INSERT operation, but the probability of calling $\mathcal{A}_0$ is the probability of one of two independent events occurring. Each of the two events happens with probability $p(n)$. From elementary probability theory $P(A \text{ or } B) = P(A) + P(B) - P(A \text{ and } B) = P(A) + P(B) - P(A)P(B)$ for independent events. The expected time to perform a DELETE operation is

$$E[T_D(n)] = (2p(n) - (p(n))^2)T_S(n) + (1 - p(n))^2 O(1).$$

This too can be simplified to

$$E[T_D(n)] = O(p(n)T_S(n)).$$

While asymptotically the INSERT and DELETE operations have the same expected time bound, the expected time to perform a DELETE is almost twice as long as the expected time to perform an INSERT.

2.2.2 Processing a Stream of Updates

A dynamic algorithm is useful in situations where we expect to perform many updates to the input set. The algorithm accepts a stream of update requests and processes them in the order received. If the update requests are generated by an adversary, this adversary can provide a worst case stream of DELETE requests for points that are in the sample. This will severely impede the performance of the algorithm because it will require calling the static algorithm with every DELETE request.

We place the following restrictions on the request stream:

- The request stream is generated by an adversary that knows the random algorithm but not the choices that it makes.

- The entire request stream is generated *a priori*, before the algorithm begins execution, but the requests are made known to the algorithm one by one, in the order generated.

The second restriction is not necessary if the stream contains only INSERT requests. As we shall see in Section 2.3.4, the restriction can be eased for DELETE requests as well, under certain conditions.

2.3 The Partition Tree

We now describe a dynamic data structure that is maintained in a manner similar to that described in the simple example above. The data structure is created in a divide-and-conquer manner, to be used with a divide-and-conquer algorithm. Let the static algorithm be $\mathcal{A}$, and let the time taken by $\mathcal{A}$ be $T_S(n)$, as in Section 2.1.

$T_S(n)$ has two components:

1. Divide: the time it takes to select S and to create a partition of P.
2. Conquer: the time it takes to combine partial solutions of the partitioned subset into a complete solution for P.

The backbone of the data structure created by $\mathcal{A}$ is a tree, which we call the *partition tree*. At the root of this tree is information about the partition of the input, based on S. The points of P are then partitioned with relation to the information in the root, and each subset is stored recursively in a subtree. The recursion may stop when a subtree contains less than some prespecified number of points. Figure 2.1 illustrates the relationship between the data set and two levels of the resulting tree.

The tree in the figure is a binary tree, but it is possible that the input is partitioned into more than two subsets at each level. The resulting tree could have less depth than the binary tree. However, if each partition contains at most a constant fraction of the data set, then the depth of the tree is still $O(\log n)$.

The divide step of the algorithm involves a top-down building of the partition tree. The conquer step is a bottom-up building of the solution. We will initially focus on

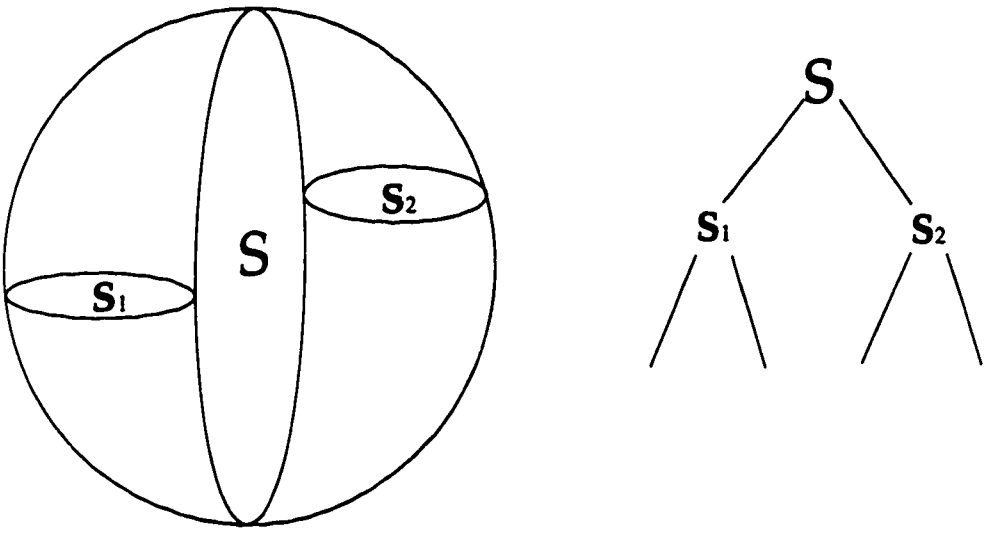

Figure 2.1: Creating the tree. Information about the partitioning is stored at the root, the partitions are recursively stored in the subtrees.

creating and maintaining the data structure (divide step). First, we describe two different versions of the partition tree.

2.3.1 Partition Tree Models

The partition tree is created using a sample S of the input. There are two ways in which S can be used to determine the partition. Either S is used to calculate a partition, or S *is* the partition. In the former case, the elements of the sample are stored, as are all other input points, in the subtrees. Since the subtrees are recursive versions of the entire tree, this means that all the points in the data set are stored in the leaves of the tree. Internal nodes contain only information about the partition, and later about the solution. We will call this model the *leaf model.*

In the latter case, the elements of S are stored in the root, and the points of $P - S$ are stored in the subtrees. Since input points are stored in internal nodes as well as in the leaves, the resulting tree is more compact, and takes up about half as much space, though the space complexity is still linear. We call this model of the partition tree the *node model.*

As will be seen later, storing the elements of S in the internal nodes of the tree leads to a problem when performing deletions from the data set. This problem does

not arise if all elements are stored in the leaves.

2.3.2 Dynamic Maintenance

The dynamic maintenance of the data structure proceeds inductively. At each step, the following condition holds:

> After a sequence of any number of updates to its input, the dynamic data structure output by the algorithm will come from the same probability distribution as a data structure output by the static algorithm, given the updated set as input.

With each new update, the entire partition tree or part of it may be completely rebuilt by the static algorithm, using unbiased, independent, random sampling. We begin by describing how to INSERT a new point into the partition tree.

Given a partition tree, we would like to add one more element to it. As in the example in Section 2.2.1, let us first consider the case where all $n+1$ points are input to the static algorithm $\mathcal{A}$.

Recall that $\mathcal{A}$ selects a random subset $S \subset P$ to determine the partition. If P contains $n+1$ elements, the probability of each one to be included in the random sample is $\frac{\sigma(n+1)}{n+1}$. The probability that the new point is included in the sample is $p(n+1) = \frac{\sigma(n+1)}{(n+1)}$. Consider the top level of the partition. With probability $1-p(n+1)$ the partitioning would be the same regardless of whether the new point is in the input set. The new point has no effect on the partition. On the other hand, with probability $p(n+1)$, a different partition would be calculated for the updated data set.

Our dynamic algorithm does precisely that: with probability $p(n+1)$ it calculates a new partition for the data set, and with the complementary probability, $1-p(n+1)$, it leaves the partition unchanged.

Note that even if the partition is left unchanged, the new point still has to be inserted into the data structure. This means that the algorithm has to determine

which subset this point belongs in, and then recursively insert it into the respective subtree. When a subtree is below a given size, it is reconstructed using the static algorithm. Figure 2.2 shows a pseudo-code description of this process.

```
DynamicInsert (tree, new_element)
    if (tree size < c)
        StaticBuild (tree elements + new_element)
    else
        Toss coin with probability of heads p(n)
        if (heads)
            StaticBuild (tree elements + new_element)
        else
            Determine subtree for new_element
            DynamicInsert (subtree, new_element)
```

Figure 2.2: Pseudo-code description of dynamic insert

Figure 2.3 shows a partition tree and a possible path taken by the algorithm. No rebuilding occurs at the root, and the point is inserted in the left subtree. Again, no rebuilding occurs, and the point is determined to belong to the right subtree. The right subtree is then completely rebuilt using the static algorithm.

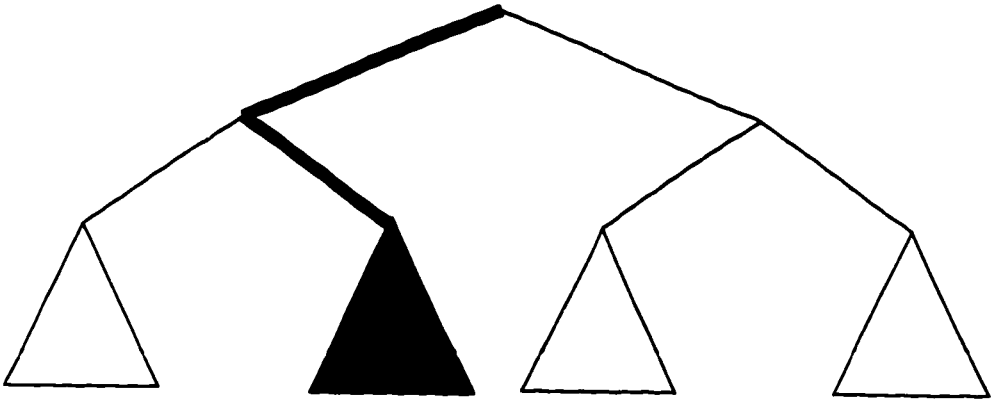

Figure 2.3: The path to subtree rebuilding: only the shaded subtree needs to be rebuilt

2.3.3 Analysis

Having described the technique we use for dynamic maintenance of the data structure, there are two things we need to show:

1. That the condition we imposed on the data structure in Section 2.3.2 holds at all times; and

2. What is the expected time to perform an update to the data set.

The Shape of the Data Structure

We show the following:

Theorem 2.1 *Let $\mathcal{A}$ be a static algorithm that creates a data structure based on a partition created by sampling of its input. Let $\mathcal{D}$ be a dynamic algorithm derived from $\mathcal{A}$ using the technique described above. Let DS_P be the data structure output by $\mathcal{A}$, given an input set P of size n. Let $P' = P \cup \{p'\}$ be an updated data set, formed by adding a new point p' to P. The data structure output by $\mathcal{A}$ given P' as input, and the data structure output by $\mathcal{D}$ given DS_P and p' as input, both come from the same probability distribution. Moreover, looking at the output data structure, it is not possible to tell which of the two algorithms produced it.*

Proof: There are two cases to consider.

1. With probability $p(n+1)$, $\mathcal{D}$ calls $\mathcal{A}$ with P' as input. In this case, the data structure produced by $\mathcal{D}$ is actually produced by $\mathcal{A}$ and the theorem is trivially true.

2. With probability $1 - p(n+1)$, $\mathcal{D}$ leaves the top level partition of DS_P intact and inserts p' in the appropriate subtree. The top level partition in DS_P is determined by points other than p'. If P' is the input to $\mathcal{A}$, the top level partition is determined by points other than p' with the same probability. Thus, it is not possible to determine which algorithm produced the top level partition. The

identical argument is then applied recursively to the point set in the subtree. With each recursive application, we show that the condition of the theorem holds for another level of the data structure. Below a certain level in the tree, the static algorithm $\mathcal{A}$ is always called, so the theorem is proved for all levels of the structure. □

Expected Update Time

As defined in Section 2.1, let $T_S(n)$ be the time necessary to construct the data structure using the static algorithm. Let $p(n)$ be the probability of a point being selected in the random sample that determines the partition. The last variable that we need for the analysis is a bound on the size of the subtrees, since at some point the algorithm may be applied to a subtree (we assume that inserting a new point into a subtree of size less than c for some constant c can be done in $O(1)$ time). Intuitively, using a sample of σ points to partition an input of size n gives an expected size of n/σ for each partition. If that is the case, then each subtree has expected size δn, where δ is the splitting ratio and is a constant < 1, typically $1/2 < \delta < 1$. In reality, the valid bounds on the size of a subtree are $n/\sigma \log \sigma$ [44, 48], though on the average each subtree indeed behaves as if it is of size n/σ [48]. While σ could be a function of n, $n/\sigma \log \sigma$ is maximized when σ is a constant. Expressing subtree size as a function of the splitting ratio δ, the bound on the subtree size becomes $\delta n \log \frac{1}{\delta}$. In the analysis that follows we assume the average size partition is of size δn.

The expected time $T_D(n)$ to perform an update of the data structure is given by the recurrence

$$T_D(n+1) = p(n+1)T_S(n+1) + (1 - p(n+1))T_D(\delta n + 1).$$

Generalizing this to all n and expanding the recurrence a bit, we get

$$T_D(n) = p(n)T_S(n) + (1 - p(n))p(\delta n)T_S(\delta n) + (1 - p(n))(1 - p(\delta n))T_D(\delta^2 n)$$

from which we get the following equation for the expected update time:

$$T_D(n) = p(n)T_S(n) + \sum_{i=1}^{\log_{1/\delta} n} p(\delta^i n)T_S(\delta^i n) \prod_{j=0}^{i-1}(1 - p(\delta^i n))$$

where we assume $p(1) = T_S(1) = 1$.

Looking at this equation, we observe that $\prod_{j=0}^{i-1}(1-p(\delta^i n))$ is a product of numbers smaller than 1, and can therefore be omitted without compromising time bounds[1].

Omitting the product term, we get the following, much nicer bound on the time to perform an update:

$$T_D(n) \leq \sum_{i=0}^{\log_{1/\delta} n} p(\delta^i n)T_S(\delta^i n). \tag{2.1}$$

We can now construct a table of expected update times for various values of $p(n)$ and $T_S(n)$, using equation 2.1. While it may seem that we should also show values for different values of δ, recall that δ is a constant $1/2 \leq \delta < 1$, and its main effect is on the constant inside the big-Oh notation. Table 2.1 shows update times for some common sample sizes and static algorithm times.

Reducing the Probability of Sampling

By decreasing the probability $p(n)$ of rebuilding the data structure, we can achieve what appear to be better time bounds. For example, if the static algorithm is $O(n \log n)$, and we only rebuild a tree with probability $\frac{1}{n \log n}$, we can get update time bounds of $O(\log n)$. However, the data structures created by this algorithm may not maintain their shape and properties. In particular, after a number of updates the data structures may no longer appear as if they were generated by the

[1] As n grows, the product converges. Terms in the product are smaller for larger values of δ and $p(n)$. If we set those at their lowest possible values, $\delta = 0.5$ and $p(n) = 1/n$, the product converges to slightly less than 0.289. For larger values of $p(n)$, it may be possible to slightly tighten the bounds by taking this product term into account.

$p(n)$	$T_S(n)$		
	n	$n \log n$	n^2
$\frac{1}{n}$	$O(\log n)$	$O(\log^2 n)$	$O(n)$
$\frac{\log n}{n}$	$O(\log^2 n)$	$O(\log^3 n)$	$O(n \log n)$
$\frac{n^\epsilon}{n}$	$O(n^\epsilon)$	$O(n^\epsilon \log n)$	$O(n^{1+\epsilon})$

Table 2.1: Time to perform an update to a data structure. $T_S(n)$ is the time to construct the structure using a static algorithm. $p(n)$ is the probability of a point being selected in the sample used to partition the data set.

static algorithm. In fact, if the data structures no longer maintain their properties, the analysis of the update bounds may no longer be valid.

2.3.4 Deletion

Deleting an element from the partition tree is performed much in the same way as inserting into it. However, now we must take care, depending on the model of the partition tree.

Leaf Model

All updates to the data set in the leaf model are performed in an identical fashion. Deletions proceed just like insertions, as described above. With probability $p(n)$, the partition tree is completely rebuilt. Otherwise, the dynamic algorithm is applied to the subtree containing the point to be deleted.

A very important point to note is that this happens *even if the data point being deleted was used to determine the partition.* The reason we can do this is that once the partition has been computed (by the static algorithm), the data points that were used to determine it are placed back in the data set and into one of the subtrees based on the partition. Figure 2.4 shows a pseudo-code description of performing the delete operation on a tree using the leaf model.

```
DynamicLeafDelete (tree, element)
    if (tree size < c)
        StaticLeafBuild (tree elements - element)
    else
        Toss coin with probability of heads p(n)
        if (heads)
            StaticLeafBuild (tree elements - element)
        else
            Determine subtree of element
            DynamicLeafDelete (subtree, element)
```

Figure 2.4: Pseudo-code description of dynamic delete on the leaf model

Since updates are performed in an identical fashion, the time to perform deletion on the leaf model is the same as the time to perform insertions, as shown in Table 2.1.

Node Model

Deleting a point from the node model is complicated by the fact that the elements are stored in internal nodes of the tree.

In this model, rebuilding actually happens under two different circumstances. One is the same as in the leaf model and in insertions, i.e., performing a single Bernoulli trial with probability of rebuilding $p(n)$, the probability that a point is included in the random sample. The other situation where rebuilding happens is if the data point is part of the partition, and stored at the root of the (sub)tree from which the node is to be deleted. Figure 2.5 shows a pseudo-code description of performing the delete operation on a tree using the node model.

The probability of rebuilding at each subtree becomes $2p(n)-(p(n))^2$, as explained in Section 2.2.1. While this slows down the update time by a factor of almost 2, it does not have an effect on the asymptotic time bounds, and they are the same as in Table 2.1.

```
DynamicNodeDelete (tree, element)
    If (tree size < c)
        StaticNodeBuild (tree - element)
    else
        If (element in root)
            StaticNodeBuild (tree - element)
        else
            Toss coin with probability of heads p(n)
            If (heads)
                StaticNodeBuild (tree - element)
            else
                Determine subtree containing element
                DynamicNodeDelete (subtree, element)
```

Figure 2.5: Pseudo-code description of dynamic delete on the leaf model

2.3.5 Processing a Request Stream

When processing a stream of update requests, the node model suffers from the same problem as the example in Section 2.2.1. An adversary that generates requests online can repeatedly make DELETE requests for points selected in the sample, forcing a call to the static algorithm with each DELETE operation. To avoid this, we place restrictions on the request stream, requiring that the stream be generated in advance, before the algorithm begins. The requests are still presented to the algorithm one at a time, and are processed in the order received.

Alternately, we can assume that the data structure appears to the adversary as a "black box", to which queries can be presented. If the adversary has no knowledge of the contents of internal nodes in the tree, he cannot force restructuring by requesting to delete elements from the root.

These restrictions are not necessary for the leaf model. Since DELETE operations are performed in an identical fashion to INSERT operations, knowledge of the choices made by the algorithm does not give the adversary an edge when generating the request stream. DELETE requests for points in the sample are no more likely to cause a call to the static algorithm than any other point in the input set.

2.4 Dynamic Divide-and-Conquer

So far, we have focused on the creation of the partition tree, which is only the first step in a divide-and-conquer strategy. The second step is computing solutions in the leaves of the tree, then moving up the tree, from the leaves towards the root, combining subtree solutions into a global solution.

This is how the static algorithm proceeds. What about the dynamic algorithm? Again, when an update is performed, we either invoke the static algorithm or move to a subtree. Once the subtree is correctly rebuilt, we need to merge the subtree solution with the other subtree(s) to create the global solution. Let the time needed by the merge step be $M(n)$.

We get the following recurrence for the expected time to perform a dynamic update:

$$T_D(n) = p(n)T_S(n) + (1 - p(n))T_D(\delta n) + M(n).$$

Solving this recurrence, we get the following equation for the expected time to perform the update:

$$T_D(n) = p(n)T_S(n) + \sum_{i=1}^{\log_{1/\delta} n} [p(\delta^i n)T_S(\delta^i n) + M(\delta^i n)] \prod_{j=0}^{i-1} (1 - p(\delta^i n))$$

where $p(1) = T_S(1) = M(1) = 1$.

Again, as in Section 2.3.3, we can omit the product term, getting

$$T_D(n) \leq \sum_{i=0}^{\log_{1/\delta} n} [p(\delta^i n)T_S(\delta^i n) + M(\delta^i n)]. \tag{2.2}$$

This time bound is often dominated by $M(n)$. If the time to combine the solutions is linear in the number of data points, then the time to perform an update is $\Omega(n)$.

2.5 Who Benefits?

This result seems somewhat discouraging, because it suggests that we cannot hope to get good dynamic time bounds for divide-and-conquer algorithms with our technique. This observation is true when the merge step involves examining all the data points. But not all divide-and-conquer algorithms require this in the merge step. It is possible that many of the data points are eliminated from consideration, so the merge involves only a small fraction of the points. If the merge step takes $O(\log n)$ time, the dynamic algorithm may have polylog time bounds on each update.

A class of problems with a constant time merge step is the class of *decomposable search* problems. A search problem on an input set P is called decomposable if for every partition (P', P'') of P, the answer to a query on P an be obtained in constant time from the answers to queries on P' and P'' [7, 9, 39]. For example membership queries are decomposable: an element is a member of P if it is a member of P' OR a member of P''. Finding the largest element in a set is also decomposable: the largest element of P is $max(max(P'), max(P''))$. Other examples of decomposable search problems include nearest neighbor queries [39] and range searching [7]. A problem which is not decomposable is the problem of determining whether a point is inside the convex hull of a point set. Since the merge step while answering queries can be done in constant time, dynamizing a decomposable search problem can be done with an additional log factor over the dynamic maintenance of the data structure. Since, according to Table 2.1, expected update times are at least polylog, updating the solution to the search problem can be done within the same time bounds as updating the data structure.

Another class of algorithms that would do well with our dynamizing technique is the class of approximation algorithms, where the approximate solution depends only on a carefully selected sample of the data. The static algorithm does not give an optimal solution, but rather one that is "good", where goodness is defined to be within some parameters. In Chapter 5 we describe how we can use our technique to create a dynamic algorithm for maintaining separators for an important class of

graphs called overlap graphs.

Combining all the results in this chapter, we can formulate the following theorem:

Theorem 2.2 *Our technique for dynamizing randomized algorithms that use sampling creates dynamic algorithms and data structures with polylogarithmic update times for the following classes of data structures and algorithms:*

1. *Balanced trees that can be statically created in $O(n \log n)$ time using a sample of size $O(\log n)$.*
2. *Decomposable search problems.*
3. *Approximation algorithms that calculate a solution based on a sample of size $O(\log n)$ and take $O(n \log n)$ time.*
4. *Divide-and-conquer algorithms with a polylog time merge step.*

Chapter 3

The Replicant Paradigm

When analyzing randomized algorithms, all time bounds are *expected* bounds, which can be attained with some probability. In the previous chapter, we discussed the expected time bounds for our dynamic data structure. In this chapter, we show how to transform these expected time bounds into *high likelihood time bounds.* We define *high likelihood* to be $1 - n^{-\alpha}$ for some $\alpha \geq 0$, where n is the input size.

3.1 High Likelihood Time Bounds

The expression "high likelihood time bound" can refer to one of two very different concepts:

1. What amount of time will give a correct answer with high likelihood?

2. What is the likelihood that a correct answer can be obtained within a set time?

The first of these pertains to the correctness of the result, and the second pertains to the time it takes to obtain a correct result. In this chapter, we focus on the latter, but we begin with a short discussion of the former.

3.1.1 Obtaining Correct Results with High Likelihood

There are known techniques for transforming static (non-incremental) algorithms from expected time bounds to high likelihood bounds. The general idea is to repeat the execution of the algorithm a sufficient number of times to guarantee correct results with high likelihood. The slowdown in the algorithm is a function of the number of times the execution is repeated.

A different technique utilizes two algorithms – a fast one that provides a correct answer within some expected probability, and a slow algorithm that guarantees the

correct answer with high likelihood time bounds. The fast algorithm is used so long as it produces the desired result. If it fails, the slow algorithm is called. The cost of invoking the slow algorithm is amortized over all operations. The resulting time bounds depend on the bounds of both algorithms and on the frequency of invoking the slow algorithm.

3.1.2 Attaining Time Bound with High Likelihood

The high likelihood bounds that we would like to achieve with our dynamic algorithm are of the second type. We would like to attain a given time bound with high likelihood. In other words, we would like the algorithm to complete within its time bounds with high likelihood.

To do this, we use multiple, independent processes. Each process maintains a copy of the dynamic data structure described in Chapter 2, which is either *active* – up-to-date and available to answer queries, or in *retirement* – being rebuilt. When a process is active, its data structure satisfies the induction hypothesis specified in Section 2.3.2. Our goal is to have, at all times and with high likelihood, enough active copies of the data structure. Duplication of the data structure slows down the algorithm. This slowdown is the price we pay for obtaining high likelihood bounds.

3.2 Terminology

The algorithm maintains many independent processes, all of which carry out the same task. Each independent process is called a *replicant*[1]. Each replicant dynamically maintains a data structure, performing updates and answering queries. Since the processes use random sampling, the replicants may create different data structures from the same input. Nonetheless, all the structures will be from the same probability distribution. At any given time, a replicant may be in one of two states:

[1]This terminology is borrowed from the movie "Blade Runner". *Replicants* in the movie are manufactured organisms nearly indistinguishable from humans. The most advanced prototypes have *memory implants* and are consequently unaware that they are not human. The replicants have a limited life-span, after which they are *retired*. In the description that follows we use terminology (*in italics*) and quotes ["in square brackets"] from the movie.

1. The replicant is *activated.* In this state, the information in the replicant data structure is current and can be used to process queries.

2. The replicant is in *retirement.* In this case, the structure maintained by the replicant is being rebuilt, and the information it stores may not be current. A replicant in retirement may not be used to answer queries.

Retirement for replicants in our algorithm is a temporary state. Each replicant repeatedly alternates between periods of activation and retirement. A period of activation begins at an *incept date* and ends at a *retirement date.* An activation period's length is the replicant's *longevity.* After a retirement period, a replicant has no memory of previous activation periods ["all those moments will be lost in time, like tears in the rain"]. All it has is a *memory implant* – knowledge of the current version of the data structure, as it has been built during the most recent retirement.

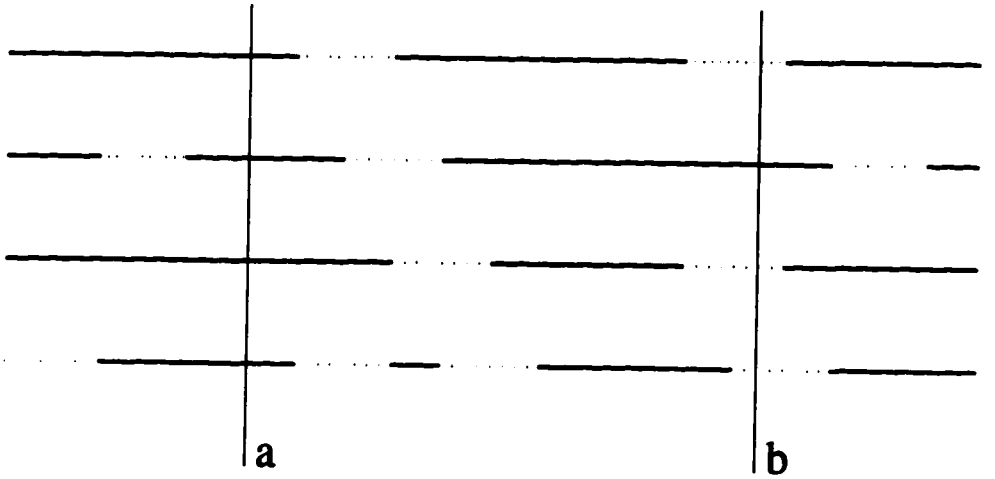

Figure 3.1: Life cycles of four replicants. Each line represents a single replicant over time. Solid lines are activated states, dotted lines represent retirement. At time a, any replicant can answer queries. At time b, all but the second replicant are retired.

Figure 3.1 illustrates the life cycles of four replicants over a period of time. Each replicant cycles between periods of activation and periods of retirement. At any point, we can determine which replicant, if any, is available to answer queries.

In order for our idea to work, we need to resolve two issues:

1. What happens when a replicant is retired for rebuilding? When that happens, other active replicants continue to process requests. We need to show that when a replicant is retired, it will be able to catch up with the other replicants, making up for the time lost when its data structure is being rebuilt. Otherwise, when a replicant is retired, it is of no further use to the algorithm.

2. How many replicants are necessary? We need to guarantee that at any point in time there will be, with high likelihood, enough active replicants that are up to date and can answer queries.

The next two sections deal with these issues.

3.3 Catching Up after Retirement

The algorithm accepts a stream of updates (INSERT or DELETE) and questions (QUERY), and the goal is for each request to be processed within a given time bound. INSERT and DELETE requests modify the data structure, while QUERY searches through the data structure and answers questions about the points stored in it.

Our goal is to determine a high likelihood bound on the time it takes to process a request. Some requests will be processed within this time bound. However, once retirement occurs, the replicant needs to stop and rebuild its data structure. As this is happening, more requests arrive and are processed by the active replicants. The retired replicant is in danger of falling farther and farther behind. This is the problem of *accelerated decrepitude*. If the replicant cannot catch up with the request stream, it is of no further use to the algorithm. Retirement happens with a positive probability, so if no catch-up occurs, after a finite number of requests no replicant will be able to process requests within the desired time bound.

The solution to this problem of accelerated decrepitude is to double the speed of processing backlog ["the light that burns twice as fast burns twice as bright"]. As each new request arrives, the replicant stores the request but otherwise ignores it. It

then proceeds to process the backlog at double speed. While this is happening new requests are still arriving, and a new backlog is accumulating. However, as we show below, the speeding up of the process ensures that the new backlog is smaller. The replicant continues to work through the backlog at double speed. After each batch is processed, there is less accumulated backlog, and eventually the replicant is caught up.

Lemma 3.1 *Let $T_S(n)$ be the time to build the data structure using the static algorithm. A replicant that is retired for rebuilding can be caught up with the request stream by processing the backlog accumulated at double speed, and it will be reactivated after no more than $6T_S(n)$ expected time.*

Proof: Let $T_H(n)$ be the time to process a request with high likelihood, and let $T_S(n)$ be the time necessary to rebuild the data structure. First, let us consider a simplified scenario, in which none of the accumulated requests require further retirement for rebuilding.

The replicant proceeds in stages through retirement toward activation. The first stage in the catch-up process is rebuilding the structure, which takes $T_S(n)$ time. During this time, $\frac{T_S(n)}{T_H(n)}$ requests arrive and are stored by the replicant. When rebuilding is done, the replicant is out of date by at most $\frac{T_S(n)}{T_H(n)}$ requests.

In the second stage, the replicant processes this backlog, performing *two* operations at a time. For each operation performed by the active replicants, the retired replicant performs two operations. The $\frac{T_S(n)}{T_H(n)}$ requests each take $\frac{T_H(n)}{2}$ time to process, so the total time to process this batch of backlog is $\frac{T_S(n)}{2}$. At the same time, new requests arrive, and are stored for further reference. Because of the quickened pace, there will be at most $\frac{T_S(n)}{2T_H(n)}$ requests accumulating. These requests will be processed in the next catch-up stage. Continuing at this pace, after i catch-up stages the backlog will be of size $\frac{T_S(n)}{2^{i-1}T_H(n)}$ and will take $\frac{T_S(n)}{2^i}$ time to process. When the backlog is 1 or less, the replicant can be reactivated. The total time in retirement is

therefore

$$\sum_{i=0} \frac{T_S(n)}{2^i} \leq 2T_S(n).$$

The simplifying assumption we made in the analysis above is that there will be no further retirement of the replicant while processing the backlog. This assumption may not always hold true. Retirement may happen with probability $\frac{\sigma(n)}{n}$.

During the first stage of retirement, $\frac{T_S(n)}{T_H(n)}$ new requests accumulate. Of those, $\frac{\sigma(n)}{n}\frac{T_S(n)}{T_H(n)}$ may require rebuilding, and each rebuilding takes $T_S(n)$ time. The remaining $\frac{(n-\sigma(n))}{n}\frac{T_S(n)}{T_H(n)}$ can be processed at double speed as above, each taking $\frac{T_H(n)}{2}$ time. Thus the total time to process the backlog accumulated during the first stage of retirement is

$$\frac{\sigma(n)}{n}\frac{T_S(n)}{T_H(n)}T_S(n) + \frac{(n-\sigma(n))}{n}\frac{T_S(n)}{T_H(n)}\frac{T_H(n)}{2} =$$

$$\frac{2\sigma(n)[T_S(n)]^2 + (n-\sigma(n))T_S(n)T_H(n)}{2nT_H(n)}.$$

The new backlog accumulating during this time is

$$\frac{1}{2n[T_H(n)]^2} \cdot 2\sigma(n)[T_S(n)]^2 + (n-\sigma(n))T_S(n)T_H(n).$$

Again, a small fraction of this will require rebuilding.

The time to complete the ith stage of retirement is

$$[B(n)]^i T_S(n),$$

where

$$B(n) = \frac{2\sigma(n)T_S(n) + (n-\sigma(n))T_H(n)}{2nT_H(n)}.$$

Thus the total time in retirement is

$$\sum_{i=0}[B(n)]^i T_S(n). \tag{3.1}$$

This summation converges when $B(n) < 1$. Given values for $T_S(n)$ and $\sigma(n)$, we can find out for which values of $T_H(n)$ the summation converges. In particular, $B(n) < 1$ when

$$T_H(n) > \frac{2\sigma(n)T_S(n)}{n+\sigma(n)}. \tag{3.2}$$

This gives us a lower bound on $T_H(n)$. Recall that $T_H(n)$ is the high likelihood time bound for processing a request. Clearly, very large values of $T_H(n)$ will yield short retirement periods, but will defeat the purpose of finding a quick way of performing updates. Values of $T_H(n)$ that are very close to the lower bound will give very long retirement periods, necessitating the use of many replicants.

If we let $T_H(n) = \frac{3\sigma(n)T_S(n)}{n+\sigma(n)}$, the base of the power term in equation 3.1 becomes

$$B(n) = \frac{2\sigma(n)T_S(n) + (n-\sigma(n))\frac{3\sigma(n)T_S(n)}{n+\sigma(n)}}{2n\frac{3\sigma(n)T_S(n)}{n+\sigma(n)}} = \frac{5n-\sigma(n)}{6n}$$

and the length of each retirement is bounded by

$$T_S(n)\sum_{i=0}\left(\frac{5n-\sigma(n)}{6n}\right)^i = T_S(n)\frac{1}{1-\frac{5n-\sigma(n)}{6n}} = T_S(n)\frac{6n}{n+\sigma(n)} < 6T_S(n).$$

□

Larger values of $T_H(n)$ will result in a shorter retirement periods. For example, for $T_H(n) = \frac{4\sigma(n)T_S(n)}{n+\sigma(n)}$, the length of a retirement period is bounded by $4T_S(n)$. Thus, we can state the following corollary to Lemma 3.1:

Corollary 3.2 *Let $T_S(n)$ be the expected time to build the data structure using the static algorithm. Let $R(n)$ be the length of the retirement period of a replicant. Then $R(n) = O(T_S(n))$.*

3.4 Number of Replicants

Now we come to the problem of determining the number of replicants necessary to ensure that with high likelihood there will be active replicants at all times. We would like the probability of all replicants being in retirement at any given time to be $n^{-\alpha}$, a very low likelihood. Maintaining more replicants requires extra time and results in a slowdown of the algorithm. We can place the following bounds on the slowdown factor by carefully choosing the number of replicants.

Lemma 3.3 *The number of replicants necessary to ensure that, with high likelihood, there are active replicants at any time, is* $O(\sigma(n)T_S(n)\log n/n)$.

Proof:

Let the time per update or request if no retirement occurs (i.e. during activation) be $T_H(n)$ with high likelihood. If retirement does occur, let the wait time until reactivation be $6T_S(n)$ with high likelihood. The expected time per update is then

$$T_D(n) = \left(1 - \frac{\sigma(n)}{n}\right) T_H(n) + \frac{\sigma(n)}{n} 6T_S(n).$$

Assume for simplicity that $\frac{n}{\sigma(n)T_S(n)} = o(1)$. This is a reasonable assumption for most applications.

As stated above, a replicant data structure needs to be rebuilt after a single update with probability $\frac{\sigma(n)}{n}$. A sequence of i updates are just i independent trials following a geometric distribution. Thus the expected *longevity* λ of a single replicant data structure is $\frac{n}{\sigma(n)}$.

We construct r independent replicants, each with its own data structure. The longevity of each replicant is $\lambda = \frac{n}{\sigma(n)}$. When a replicant is retired, the others don't wait for it to reconstruct its data structure. Rather, the reconstruction happens while activated replicants continue processing requests. As a result, we need to account for "catch-up" time before reactivation. Let the expected catch-up time for a retired

replicant be $6T_S(n)$, as described in Section 3.3. The life of a replicant consists of periods of activation of expected length λ, followed by periods of retirement of length $6T_S(n)$.

Thus at any given time, the probability that a single replicant is activated is

$$\gamma \geq \frac{\lambda}{6T_S(n)+\lambda} = \frac{n}{6\sigma(n)T_S(n)+n}.$$

Since the r replicants are independent, the probability that at least one replicant is currently activated is $1-(1-\gamma)^r$. The value of r for which this translates to a high likelihood event, i.e. an event with probability $1-n^{-\alpha}$, is

$$r \leq \frac{-\alpha \log n}{\log(1-\gamma)}.$$

By our assumption, $\frac{n}{\sigma(n)T_S(n)} = o(1)$. This means that as n increases γ approaches 0. We can get a good approximation for $\log(1-\gamma)$ from the generating function of $\ln(1-\gamma) = \sum_{n>0} \frac{\gamma^n}{n}$. This tells us that $\log(1-\gamma) = -(\gamma + O(\gamma^2))$. Consequently,

$$\begin{aligned} r &= \frac{-\alpha \log n}{\log(1-\gamma)} \\ &= \frac{\alpha \log n}{\gamma + O(\gamma^2)} \\ &\leq \frac{\alpha \log n}{\gamma} \\ &\leq \frac{\alpha \log n(6\sigma(n)T_S(n)+n)}{n} \end{aligned}$$

which means that the number of replicants r is

$$r = O\left(\frac{\sigma(n)T_S(n)\log n}{n}\right)$$

as stated in the lemma.

□

3.5 When is This Useful?

The results in the previous two sections show us that we can attain high likelihood time bounds by using multiple replicants, each with its own data structure. The replicants cycle between periods of activation and retirement, and at any given time there are, with high likelihood, replicants whose data structure is up to date with respect to the request stream. The number of replicants is a function of the probability of retirement and the length of retirement. The probability of retirement is a function of the sample size selected by the algorithm. The length of retirement is a function of the time needed by the static algorithm.

Table 3.1 gives the number of replicants necessary to attain high likelihood time bounds for some common sample sizes and static algorithm time bounds.

	$T_S(n)$		
$\sigma(n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
c	$O(\log n)$	$O(\log^2 n)$	$O(n \log n)$
$\log n$	$O(\log^2 n)$	$O(\log^3 n)$	$O(n \log^2 n)$
n^ϵ	$O(n^\epsilon \log n)$	$O(n^\epsilon \log^2 n)$	$O(n^{1+\epsilon} \log n)$

Table 3.1: Number of replicants needed to attain high likelihood time bounds for some common sample sizes and static algorithm times. $T_S(n)$ is the time to construct the structure using a static algorithm. $\sigma(n)$ is the sample size used by the static algorithm.

As can be seen from the table, using replicants would not be a practical method with which to attain time bounds for randomized algorithms that use large sample sizes. Nor would they be practical in cases where the dynamic data structure is based on a static algorithm that is $\Omega(n \log n)$.

However, if we build our dynamic data structure for randomized algorithms with time bounds of $O(n \log n)$, that use sample sizes of $O(\log n)$ or smaller, then we can attain high likelihood time bounds using replicants with only a logarithmic factor slowdown. An example of such an application can be found in Chapter 5.

3.6 Putting It All Together

Combining the results of the two previous sections, we can now determine the high likelihood time bound. The number of replicants necessary can be determined from table 3.1. This determines the slowdown factor to the algorithm. But what is it that we are slowing down? What is the high likelihood time bound on performing an update given a *single* replicant?

As mentioned in Section 3.3, we can choose the high likelihood time bound, subject to inequality 3.2. The inequality imposes a lower bound on the high likelihood time bound on performing an update by a single replicant. How does this relate to the dynamic time bounds computed in Section 2.3.2? By assigning values for $T_S(n)$ and $\sigma(n)$ in the equation, we can see that the high likelihood time bounds have the same order of magnitude as the expected dynamic bounds, for all values in the table.

Practically, selecting a higher time bound will result in fewer replicants being necessary. This means that there is an additional trade-off that has to be considered when implementing this algorithm. Asymptotically, however, we can summarize the results in this chapter as follows:

Theorem 3.4 *Given a static, randomized algorithm that uses a sample of size* $\sigma(n)$ *in its computations and has a running time of* $T_S(n)$*, we can transform it into a dynamic algorithm with high likelihood time bounds. The high likelihood time bound on performing an update is a product of the* expected *time to perform an update and a slowdown factor* r*, where* $r = O(\sigma(n)T_S(n)\log n/n)$.

Chapter 4

Empirical Results

This chapter describes how we tested our data structure, and reports the results of the testing. We tested our data structure using a simple application: a binary search tree of integers. We performed two separate sets of experiments. First, we ran programs testing the static and dynamic behavior of the data structure. Then we ran simulations of replicants. In both sets of experiments we found that the empirical results met or exceeded the theoretically predicted bounds.

4.1 Basic Data Structure

Two separate data structures were tested. In the first, which we call the *node model*, data points were stored both in the internal nodes and in the leaves of the structure. In static construction, the *median* of three random data points was picked and selected to be in the root and serve as the partitioning element. The rest of the data were then partitioned to either the left or the right subtree depending on the relationship to the point in the root. This process continued recursively until there was at most one node in each subtree.

In the second data structure, which we call the *leaf model*, nodes were stored only in the leaves. During static construction the *mean* of three random data points was selected as the pivot. In this case, however, all data points were partitioned into the left or right subtree, and no data point was stored at the root.

We selected the sample size to be the same in both models so that the we could make valid comparisons between the models.

Note that for perfectly balanced trees, the depth of a tree in the node model is only one less than the depth of a tree on the same data in the leaf model. This means that the time it takes to search for an item in either tree should be essentially the same. The main difference between the two models is that the node model takes only

half as much space as the leaf model. This is true because in a balanced binary tree the number of leaves is about half the number of the total nodes in the tree [16].

In each of the two models, we performed the following experiments:

1. Build a tree of random data.
2. Given a tree, insert one more element.
3. Given a tree, delete one element.
4. Given a tree, add 1% more elements.
5. Given a tree, delete 1% of its elements.
6. Build a tree using a sequence of insert operations.
7. Given a tree, process a stream of INSERT, DELETE and QUERY operations to it, simulating the work of a single replicant.

4.1.1 Practical Consideration

An immediate problem that arose was how to deal with duplicate data. The problem is profound in the leaf model, where duplicate data leads to infinite recursion. The problem is not as acute in the node model, since the size of the data set is reduced by at least one with each level in the tree. Nonetheless, having all duplicate data creates a very lopsided tree. We avoided these problems by disallowing duplicate data.

4.1.2 Timing the Runs

The programs were written in C++ and run on a Pentium Pro-200/256K cache, with 128Mb of EDO RAM, 167Mb of swap, and 5 Gb of disk space. We used the utility functions in the C++ *time.h* library to time specific parts of each run, so that overhead is not included in our figure. The timer gives the number of clock ticks between the time that the timer is started and the time it is stopped. We then divided this number by the number of clock ticks per second on the machine. The

results are therefore machine dependent. We repeated some of the experiments on other machines. While the times reported were different, the general behavior was replicated.

Since the shape of the data structure depends on a random sample of the input, different runs on the same data would result in different data structures being created. Different shapes of the data structure could lead to different runtimes. To get around this problem, we ran each experiment 100 times. All times reported in this chapter represent the mean runtime of 100 runs.

In some of the experiments, it was not possible to utilize this method for timing the runs, because the items of interest ran in less than a clock cycle. Instead, we counted the number of comparisons made by the program. We counted only comparisons that were specific to the general algorithm, not ones that were specific to the binary search tree or to the details of the implementation. We included decisions the program made when selecting the random sample, when determining whether to rebuild and when deciding which subtree to proceed to, but did not count comparisons made when searching or copying an array, since using an array was an implementation specific decision.

4.2 Building a Tree

Table 4.1 shows the time it took to construct a tree of n nodes, where n ranges from 1,000 to 512,000. Random data were generated by shuffling the numbers 1 through n, and stored in a file which was read by the tree building program. The tree was built by selecting or generating a pivot (for the node and leaf models, respectively), partitioning the data to the left and right subtrees, and recursively building each subtree. The left column gives the time for the node model and the right column gives times for the leaf model. These times are the means of 100 runs. Only the actual building of the tree was timed (reading in the data was considered overhead). Analysis of the data in the table shows that the time to build the tree is bounded by $O(n \log n)$. This is consistent with known properties of binary trees on random data.

n	node model	leaf model
1,000	0.0051	0.0081
2,000	0.0112	0.0172
4,000	0.0217	0.0341
8,000	0.0464	0.0717
16,000	0.0992	0.1500
32,000	0.2093	0.3133
64,000	0.4387	0.6482
128,000	0.9269	1.3456
256,000	1.9733	2.7808
512,000	4.0891	5.8685

Table 4.1: Time (in seconds) to build a table with n nodes (mean of 100 runs).

We can also see that the leaf model took almost twice as long as the node model to build the tree. This makes sense, since the tree in the leaf model has twice as many nodes as the tree in the node model, and would therefore take longer to construct.

4.3 Dynamic Maintenance

Once a tree is built, we were interested in how long it would take to perform updates to it, in the form of inserting and deleting nodes. It was not possible to time the insertion and deletion of single elements, because these operations took less than one clock tick. However, we could still extract some useful information from these runs.

Recall that an element is inserted into a (sub)tree by either rebuilding the entire (sub)tree or determining which of its two subtrees the element belongs to and repeating the process on that subtree. The higher the level of the root of the subtree, the less likely it is that the tree will be rebuilt. For each dynamic operation, we recorded at what depth in the tree rebuilding actually took place.

We also recorded the number of comparisons made by the program for a single update. This did not yield any useful information. The data were so spread out that neither the mean nor the median of a sample of 100 runs showed any consistent pattern, either within or between models.

We obtained more useful results by timing how long it took to perform a large

number of insert or delete operations. Instead of using an arbitrary large number, the number of operations we performed was 1% of the size of the tree. Since we expect a single update to take $O(\log^2 n)$ time, it would take $O(n \log^2 n)$ time to insert 1% again the number of nodes into the tree.

4.3.1 Performing a Single Update

Tables 4.2 and 4.3 show the mean and range of the depth in the tree where rebuilding of the subtree occurred, in each of the node and leaf models, for insertion and deletion, respectively. If rebuilding never happened, this represents the level of the leaf where the update took place.

Insertion

n	Mean (range)			
	node model		leaf model	
1,000	9.34	(1–16)	8.77	(2–14)
2,000	11.09	(5–19)	10.1	(4–18)
4,000	11.43	(2–19)	11.15	(2–18)
8,000	12.49	(3–21)	12.42	(4–18)
16,000	14.77	(9–24)	12.82	(5–19)
32,000	15.6	(8–24)	14.21	(5–21)
64,000	16.4	(10–24)	15.17	(7–24)
128,000	17.61	(8–27)	16.48	(11–22)
256,000	19.29	(10–30)	18.04	(10–26)
512,000	19.34	(9–30)	18.69	(11–26)

Table 4.2: Depth at which tree rebuilding occurred for a single insert (mean and range of 100 runs).

The mean level at which rebuilding happened was consistently higher in the leaf model than in the node model, though not significantly so. Closer examination of the data shows that in the node model rebuilding sometimes happened at deep levels. It may be that the leaf model trees are more balanced, due to the method used in finding a pivot. Since many times the new element is inserted at a leaf without any

rebuilding, an unbalanced tree will have insertions at deeper levels, contributing to a high mean depth at rebuilding/inserting.

Deletion

	Mean (range)			
n	node model		leaf model	
1,000	8.59	(3–17)	9.08	(1–20)
2,000	10.36	(4–19)	10.42	(4–17)
4,000	11.22	(5–19)	11.41	(3–19)
8,000	12.07	(4–22)	12.52	(5–20)
16,000	13.82	(7–24)	13.29	(5–21)
32,000	14.38	(7–23)	15.03	(7–23)
64,000	15.79	(9–25)	16.17	(7–27)
128,000	16.83	(7–28)	16.24	(2–26)
256,000	18.43	(10–26)	18.00	(11–26)
512,000	19.23	(11–28)	18.70	(10–27)

Table 4.3: Depth at which tree rebuilding occurred for a single deletion (mean and range of 100 runs).

When performing deletions, we encountered a new problem with the data node model. Since elements are stored in internal nodes as well as in leaves, rebuilding of a subtree has to happen if the node we want to delete is at the root of the subtree. If not, the subtree is rebuilt with probability $\frac{1}{n}$, where n is the number of nodes in the subtree. Thus the probability of rebuilding is double that of the leaf model. We expected this to make the mean level of rebuilding a bit higher. The depth of rebuilding was indeed slightly higher for deletion than for insertion. The difference between insertion and deletion in the node model was more pronounced in the next experiment, where we performed a large number of updates.

4.3.2 Performing Many Updates

We then measured the length of time it took to insert an additional 1% more nodes into a tree, and the time to delete 1% of the nodes in the tree. Table 4.4 summarizes the results of these experiments. Times are in seconds.

	Insert		Delete	
n	node model	leaf model	node model	leaf model
1,000	0.0011	0.0019	0.0017	0.0013
2,000	0.0015	0.0051	0.0045	0.0043
4,000	0.0072	0.0165	0.0091	0.0098
8,000	0.0136	0.0239	0.0203	0.0266
16,000	0.0306	0.0564	0.0526	0.0584
32,000	0.0613	0.1378	0.1103	0.1331
64,000	0.1489	0.2975	0.2623	0.2952
128,000	0.3592	0.6651	0.5999	0.6413
256,000	0.7949	1.4699	1.2763	1.4082
512,000	2.1029	3.2472	2.7770	3.4355

Table 4.4: Time (in seconds) to perform a sequence of updates to a tree (mean of 100 runs). For a tree of size n, each sequence involved $0.01n$ INSERT or DELETE operations.

Somewhat surprisingly, the leaf model still took almost twice as much time as the node model to perform a sequence of INSERT operations. While the overall tree is twice as large, the depth of the tree in the leaf model should not differ by much from the depth of the tree in the node model.

More in accordance with our expectations, the node model suffered a degradation of performance when performing the DELETE operation. In the leaf model, the time to perform a sequence of INSERT operations was comparable to the time it took to perform the sequence of DELETE operations. However, in the node model, the sequence of DELETE operations took about 50% longer to complete.

As mentioned above, we expected the time to perform this operation to be $O(n \log^2 n)$. We plotted the results from Table 4.4 in figure 4.1. The solid line represents $cn \log^2 n$ for some small constant c. As is evident from the figure, the results bear out our expectations.

4.4 Incremental Building

We then proceeded to build a tree incrementally. In other words, we built a tree using a sequence of insert operations. The assumption is that the data is given one

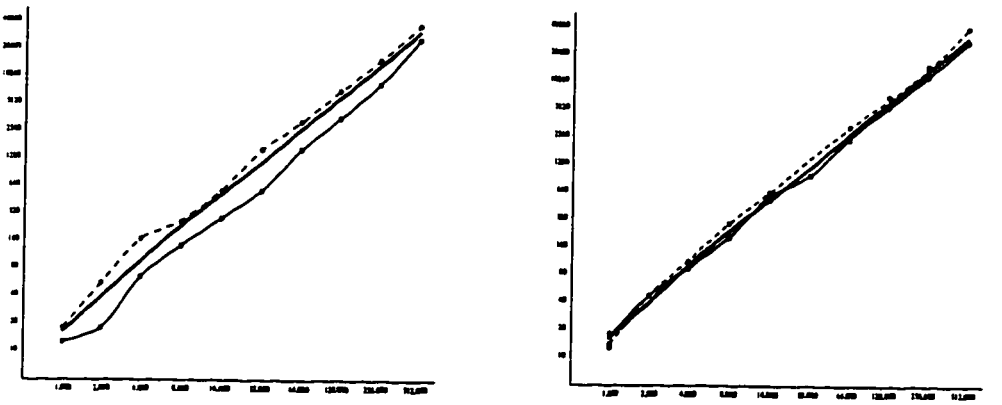

Figure 4.1: Time to perform $0.01n$ updates on a tree of size n (mean of 100 runs). INSERT operations are plotted on the left, DELETE on the right. Node model results are represented by a solid line, leaf model by a dashed line. The heavy line is $cn \log^2 n$.

point at a time rather than being known in advance. This is where we expect to see the advantage of our data structure over a regular binary search tree, especially for extreme cases where the order of the data defines a very lopsided tree.

For totally ordered data, our data structure would produce a somewhat lopsided tree, because, as the tree grows, the probability of rebuilding at the root becomes increasingly smaller. At each point, if rebuilding does not happen the new node is added to the same side of the tree, so our tree becomes right heavy. Still, while lopsided, the tree should exhibit much better performance than the $O(n^2)$ behavior of a binary search tree on ordered data. We expect slightly worse behavior than on random data, but not significantly so.

Table 4.5 shows the mean time to build a tree using a sequence of insertions. The two columns on the left show the times for random data. The two columns on the right show the run times for ordered data.

As can be seen from the table, the predictions hold true for the leaf model. It took slightly longer to create a tree of ordered data using a sequence of insert operations, but the time it took to do so is close to the time for the random data. A surprising result, which we were able to replicate several times, is that the node model actually behaved better than expected, with the tree on ordered data actually built in *less* time than the tree on random data.

n	Node Model		Leaf Model	
	random	ordered	random	ordered
1,000	0.1077	0.1048	0.1775	0.2008
2,000	0.2647	0.2474	0.4150	0.4726
4,000	0.6131	0.5792	0.9432	1.0833
8,000	1.3943	1.3018	2.1853	2.4120
16,000	3.2131	2.9869	4.9845	5.4369
32,000	7.4634	6.6676	11.5508	12.1481
64,000	17.3095	14.9053	26.4080	27.1913
128,000	39.7671	34.0909	60.1526	61.3138
256,000	90.8518	77.1845	133.7570	134.5993
512,000	203.2186	170.1584	311.3469	303.8114

Table 4.5: Time (in seconds) to build a tree by a sequence of insert operations for random and ordered data (mean of 100 runs).

4.5 Replicants

In the second phase of our experiments, we wanted to simulate replicant behavior. After examining the numbers from the earlier part of the experiment, we decided to do all the work on a tree of 128,000 elements. We built a tree using the static algorithm, and applied to the tree a stream of 6,400 requests. For the request stream, we generated a sequence of pairs, $<x, y>$, where y was a data element and x was an action to perform: SEARCH, INSERT or DELETE. We applied this stream to the same tree 100 times, each representing a replicant.

We then wanted to determine the mean time to process each of the requests, as well as the mean time to process each operation: SEARCH, INSERT or DELETE. Since the time to process a single update fell below the threshold recordable by the machine, we counted the number of comparisons performed by the algorithm, as described in Section 4.1.2.

Each of the 100 replicant runs generated a sequence of cumulative number of comparisons after processing each request. We used this sequence to determine the mean number of comparisons for each request (over 100 runs), as well as the mean number of comparisons when performing SEARCH, INSERT and DELETE. Table 4.6 shows the mean number of comparisons for each of the three operations.

Action	Number of requests	Node Model	Leaf Model
INSERT	2150	844.34	1028.98
DELETE	2118	898.05	1032.20
SEARCH	2132	57.64	59.65

Table 4.6: Number of comparisons to perform each of the three operations allowed on our data structure on a tree of 128,000 elements (mean of 100 runs, each consisting of 6,400 requests).

The numbers in the table are consistent with the results in the previous sections. Specifically,

- updates on the leaf model take longer than on the node model.
- In the leaf model, INSERT and DELETE require roughly the same number of comparisons.
- In the node model, DELETE takes longer than INSERT.
- SEARCH requires slightly fewer comparisons on the node model, consistent with the relative depth of the trees.

We then sought to determine the time per request for which a stable number of replicants can handle the input stream. Too small a time step meant that the number of viable replicants would drop off, placing a limit in the length of the input stream that the replicants can process.

Alternately, given the time bound we would like to maintain, we could empirically determine the number of replicants needed to process a request stream. Of course, for small time steps, the answer may be than no amount of replicants will allow us to do so, because the number of active replicants will be a rapidly decreasing function.

4.5.1 Methods

We took several measures to standardize the processing time of the request stream across replicants.

We started by ignoring all search requests, and focusing only on the *update* request stream. There were several reasons for doing this. First, replicants in retirement do

not process searches, as they are not up-to-date with respect to the data set. Second, we are interested in the number of active replicants available to answer queries, less so in the actual queries themselves. Third, searches take significantly less time than updates, so time bounds for updates are trivially true for queries.

Next, we made sure all active replicants are "in step". This is equivalent to making sure that update requests arrive with a fixed inter-arrival interval. This is important whenever there are more than one replicant. If there is only a single replicant, it can process a request as soon as it finishes processing the previous request. However, if there are several replicants, we would like the *active* replicants to march in lockstep. In particular, we do not want any replicant to get ahead of the request stream. At any point, the data set of all active replicant should be the same. If a replicant processed an update request faster, we let it sit idle until the next update request arrived.

The inter-arrival interval was picked externally. As explained in Section 3.6, it is appropriate to use the expected dynamic time bound (multiplied by a constant factor) as a high likelihood time bound for a single replicant. We used the mean number of comparisons per update as our baseline, and multiplied that by constant factors ranging from 0.25 to 9. We will refer to this number as the *desired time bound* (even though technically it is not a measure of time but rather a comparison count). For large intervals, it is possible that all active replicants were idle at times. We made no correction for this.

Last, we needed to account for the fact that replicants in retirement process their request stream at double speed. We did this by halving the number of comparisons whenever the replicant was lagging behind.

4.5.2 Time in Retirement

For each replicant, we now had a cumulative stream of comparison counts. To determine the mean time a replicant spent in retirement, we counted the length of consecutive subsequences where the replicant was lagging behind. The mean retirement period was inversely related to the length of the desired time bound.

The mean time in retirement did not differ significantly between models. In both models, selecting the desired bound to be 2-3 times the mean update time resulted in a mean retirement period of between 6 and 7 requests. This is in fact *better* than the length of the retirement period predicted in Section 3.3, where the length of retirement was proportional to the time bounds on the static algorithm. In practice, the length of retirement was more closely related to the time bound on the dynamic algorithm.

4.5.3 Number of Replicants

We simulated the work of 100 replicants. Given the streams of cumulative comparison count associated with each replicant and a desired bound, we counted how many of the replicant were keeping up with the request stream, i.e., how many were not in retirement. This produced a sequence of active replicant counts. We then examined these sequences to see if the number of active replicants stabilizes. For smaller desired bounds, the number of active replicant declined steadily. However, the number of active replicants was stable for desired bounds which were twice the mean update time. We determined the necessary number of replicants to be one more than the maximum number of replicants in retirement at any point. The results are summarized in Table 4.7.

There was no significant difference between the node and leaf models. Using a desired bound less than 1.5 times the mean bound resulted in the number of replicants declining steadily. As can be seen, a higher desired bound results in fewer replicants necessary. Since the bound on the update time is a function of both the desired bound and the number of replicants, the product of these two is what determines the overall high likelihood time bound. The numbers in Table 4.7 suggest that it is better to select a low desired bound (e.g. twice the expected bound) with a higher number of replicants.

The randomized binary search tree, with a static time bound of $O(n \log n)$ and constant sample size, is predicted to require $O(log^2 n)$ replicants in order to maintain

desired bound/ expected bound	node model	leaf model
1	NA	NA
1.25	NA	NA
1.5	40	44
1.75	35	39
2	31	33
2.25	30	29
2.5	27	27
2.75	25	25
3	23	24
4	20	19
5	16	17
6	15	14
7	14	13
8	13	12

Table 4.7: Number of replicants necessary to maintain a desired high likelihood time bound for updating a tree of 128,000 elements.

its desired time bounds. For a tree of 128,000 elements, $\log^2 n$ is close to 300. However, simulation of the replicant model shows that the actual number of replicants needed is much smaller. Of course, this does not really tell us anything about how the number of replicants relates to the size of the tree, since we only examined one tree size. By repeating the experiments with different tree sizes we could get results that will enable us to predict the number of replicants necessary as a function of the tree size and the desired time bound.

4.6 Future Work

The sample application that we implemented exhibited behavior that was consistent with the theoretically predicted bounds, and in some cases did even better. We are now in the process of rewriting the code so that it is even more amenable to modularization. We then plan to implement a more complex problem, such as the separator algorithm described in the next chapter.

Chapter 5

Applications to Geometric Separators

5.1 Motivation

A *separator* of a graph is a subset of its vertices that, when removed from the graph, separates the remaining vertices into two or more disconnected subgraphs of approximately equal size. Small separators are very useful in divide-and-conquer algorithms. Not all graphs have small separators, and it is interesting to determine which families of graphs have this property. It is well known that trees and planar graphs have small separators [38]. Other classes of graphs that have small separators are the class of graphs with bounded genus [27], and graphs that have a bounded excluded minor [4].

Miller, Teng, *et al* [41, 76, 42, 20] defined a much wider class of graphs, called *overlap graphs*, and introduced *sphere separators*, a geometric approach to finding small separators in this class of graphs.

In this chapter, we apply our technique to a randomized algorithm for finding sphere separators, yielding the best known time bounds for dynamic maintenance of separators and separator-based search structures. Section 5.2 gives background information about sphere separators and describes known results in this area. Section 5.3 details the application of our technique to maintaining separators for a set of points and their induced graph. In Section 5.3.2 we define a separator tree and detail its dynamic maintenance. Section 5.4.1 gives the details of applying our replicant paradigm to the dynamic sphere separator algorithm, to yield high likelihood time bounds.

5.2 Geometric Separators

Sphere separators are geometric entities, and overlap graphs are geometrically defined, derived from sets of points, and special neighborhoods of these points. This

makes them useful when solving problems in computational geometry, such as nearest neighbor queries ([22, 76]); in numerical analysis, such as nested dissection problems for large scale meshes in two and three dimensions [41]; and in path algebra problems [58].

5.2.1 Neighborhood Systems

Let $P = \{p_1 \ldots p_n\}$ be a finite set of generally positioned points in $\mathbf{R}^d$.
A *d-dimensional neighborhood system* in $\mathbf{R}^d$ is a set of balls $\mathcal{B} = \{B_1 \ldots B_n\}$, with ball B_i centered at p_i. $\mathcal{B}$ is a *k-neighborhood system* if each B_i contains at most k points from P. $\mathcal{B}$ is a *k-ply neighborhood system* if each p_i is contained in at most k balls.

Given any set of points P in $\mathbf{R}^d$, we can easily construct a k-neighborhood system by centering at each point the largest ball that will contain $k+1$ points from P. A k-ply neighborhood system is more difficult, but the following lemma relates the two:

Lemma 5.1 [14] *Each k-neighborhood system in* $\mathbf{R}^d$ *is* $\tau_d k$*-ply, where* τ_d *is the* kissing number *in d dimensions, i.e. the maximum number of non-overlapping unit balls in* $\mathbf{R}^d$ *that can be arranged so that they all touch a central unit ball.*

Note that τ_d is independent of n and depends only on the dimension d. While its exact value is not known for all d, the known bounds [76] are

$$2^{.2075d(1+o(1))} \leq \tau_d \leq 2^{.401d(1+o(1))}.$$

The *k-nearest neighborhood digraph* of the points $P = \{p_1, \ldots, p_n\}$ in $\mathbf{R}^d$ is a graph $G_k = (P, E)$ where P is the vertex set of G_k and its edge set is $E = \{(p_i, p_j) | p_j$ is one of the k nearest neighbors of $p_i\}$ (i.e., there are directed edges from each point to its k nearest neighbors). We will call the 1-nearest neighborhood graph the *nearest neighborhood graph.*

Note that since we require the points to be in general position, the k nearest neighbors of a point are unique. The outdegree of every vertex in the graph is k, and by Lemma 5.1, the indegree is bounded by $k\tau_d$.

5.2.2 Sphere Separators of Points in $\mathbf{R}^d$

A *sphere separator* of a set of points P in R^d, is a sphere S that partitions P into two sets: P_I, the points in the interior of S; and P_E, the points in its exterior.

The *induced separator set* of a neighborhood system $\mathcal{B}$ of P is the set of points whose associated balls intersect the sphere separator.

Let $s(n)$ be a positive, monotone function of n, such that $s(n) < n$. Given a finite set of points $P = \{p_1, \ldots, p_n\}$ in $\mathbf{R}^d$, and a constant $0 < \delta \leq 1$, a $(d-1)$-sphere S in $\mathbf{R}^d$ is an (s, δ)-*separator* of P if it partitions P into two subsets, P_I and P_E (the points on the interior and the exterior of S respectively) such that:

1. the *induced separator* of the point set is of size at most $s(n)$, and
2. $|P_I| \leq \delta n$, $|P_E| \leq \delta n$.

δ is called the *splitting ratio* of P.

An (s, δ)-*separator tree* of P is a binary tree. At the root is stored information about an (s, δ)-separator of P. Points in the interior of the sphere are stored in a recursive structure in the left subtree, and points on its exterior are stored in a recursive structure in the right subtree. Recursion stops when a subtree contains less than a pre-specified number of points. At each internal node of the tree, the sphere separator S' is required to be an (s, δ)-separator of the subset of points stored in the subtree. That is, if the set of points separated by the parent of the node is n', then S' is required to be an $(s(n'), \delta)$-separating sphere. In this chapter, wherever s, δ are determined by context, we will simply call the (s, δ)-separating sphere a *good* separating sphere.

Space Complexity

The separator tree contains information about the separators in its internal nodes, and the points of P in the leaves. If only the center and radius of the sphere is stored at each internal node, the space complexity for the entire tree is $O(n)$. This can be easily seen from the recurrence $f(n) = f(n_I) + f(n_E)$ where n_I and n_E are the number of points in the interior and exterior of S respectively, and $n_I + n_E = n$. If information about the induced separator set is also stored in the internal nodes, this recurrence becomes $f(n) = f(n_I) + f(n_E) + s(n)$. Thus if $s(n) \leq O(n^\beta)$, where $0 \leq \beta \leq 1$, the space is $f(n) = O(n)$ [22].

Let $G = (P, E)$ be a graph induced by P, such as a nearest neighbor graph. We say that S is a (s, δ)-*separator* of G, if it partitions the vertices of G into two subsets, P_I and P_E as above, and the number of edges that S intersects is no more than $s(n)$. P_I and P_E induce two separate subgraphs, G_I and G_E. The *separator set* of G is the subset of the vertices incident on edges which intersect the separator. The size of the separator set is no more than $2s(n)$.

An (s, δ)-*separator tree* of G is a binary tree. At the root is stored a separator set corresponding to an (s, δ)-separator of G. G_I and G_E are stored recursively in the two subtrees. Note that each vertex in the separator set is also stored in one of the subtrees, i.e, the separator tree follows the leaf model. However, if G has a separator set of size $O(n^{\frac{d-1}{d}})$, then the total storage required is $O(n)$ [22].

5.2.3 Graph Separators

Let $G = (V, E)$ be a graph or digraph. Given $0 < \delta < 1$, a *(s, δ)-separator* of G is a set of vertices $S \subset V$ such that deleting the vertices of S and their incident edges disconnects G into two subgraphs, G_0 and G_1, where we require that

1. S is of size at most s, and

2. each separated subgraph G_0, G_1 contains no more than δn vertices.

An *(s, δ)-separator tree* of a graph or digraph is a binary tree. At the root is stored information about the separator. The two separated subgraphs are stored in recursive structures in the two subtrees. Recursion stops when a subtree contains less than a pre-specified number of vertices. As before, the space required to store this tree is linear if $|S| \leq O(n^{\beta})$, where $0 \leq \beta \leq 1$.

An *α-overlap graph* is a graph induced by a neighborhood system $\mathcal{B}$ of a set of points P. The vertex set of the graph is P, and an edge exists between two vertices p_i and p_j if $\alpha B(p_i) \cap B(p_j) \neq \emptyset$ and $\alpha B(p_j) \cap B(p_i) \neq \emptyset$, where, if B is a ball centered at p of radius r then αB is a ball centered at p with radius αr. k-neighborhood graphs are a special case of overlap graphs.

5.2.4 Algorithms for Finding Sphere Separators

The pioneering work in sphere separators was done by Miller, Teng, *et al* [22, 43, 41, 76]. They showed that for any (generally positioned) set of n points $P = \{p_1, \ldots, p_n\}$ in $\mathbf{R}^d$, there is a sphere S which (s^*, δ^*)-separates P, with separator size $s^* = O(n^{\frac{d-1}{d}})$ and a splitting ratio $\delta^* = \frac{d+1}{d+2}$ [43].

Furthermore, Teng [76] showed a randomized, linear time algorithm based on sampling for finding such a separator. His result states that:

Lemma 5.2 [76] *Let $P = \{p_1, \ldots, p_n\}$ be any set of n generally positioned points in $\mathbf{R}^d$, and let ϵ be a constant $0 < \epsilon < \frac{1}{d+2}$. Let S be a random sample of points in P of size $\sigma(n) = O(\frac{d}{\epsilon^2}(\log \frac{n}{\epsilon} + \log \eta))$. Then there is a randomized, linear time algorithm SPHERE-SEPARATOR(S), with success probability of $1 - \frac{1}{\eta}$, which yields an (s, δ)-separator of P with separator size $s \leq n^{\beta}, \beta = \frac{d-1}{d} + 2\epsilon$, and splitting ratio $\delta = \frac{d+1}{d+2} + \epsilon$.*

In particular, if we choose $\eta = O(\log n)$, then with a sample of size $\sigma(n) = O(\log n)$ the algorithm will have a probability of success $1 - \frac{1}{\log n}$. Note that the

"goodness" of the sphere separator obtained by SPHERE-SEPARATOR(S) is only slightly non-optimal as compared with the existence results cited above.

To avoid time bounds which are exponential in d, the algorithm employs a method of recursive sampling. Sample sizes are carefully selected to be as stated in Lemma 5.2, and smaller samples are used to verify the "goodness" of the initial sample. For our purposes, it suffices to know that a random sample of the input points of size $\sigma(n)$ provides us a good sphere separator with probability $1 - \frac{1}{\sigma(n)}$. In what follows, we will take $\sigma(n)$ to be $O(\log n)$.

5.3 Maintaining a Separator Dynamically

In this section, we show how we can apply our data structure to algorithms for dynamic maintenance of a sphere separator of a dynamically changing finite set of points P, and for dynamic maintenance of a separator-based search structure for P. The algorithms accept a sequence of requests from an adversary. Each request is a pair <point, action>, where an action may be to INSERT or DELETE the input point from P, or answer a QUERY about the input point. A QUERY about the separator can be to determine whether the point is inside or outside the separator. A QUERY about the search structure would be a request to SEARCH for the point in the search structure.

We show the following result:

Theorem 5.3 *Let* $0 < \epsilon < \frac{1}{d+2}, \delta = \frac{d+1}{d+2} + \epsilon, \beta = \frac{d-1}{d} + 2\epsilon$, *and* $s \leq n^{\beta}$. *Dynamic maintenance of an* (s, δ)*-sphere separator of* n *points in* $\mathbf{R}^d$ *with INSERT and DELETE operations, as well as queries about the separator, can be done in* $O(\log n)$ *incremental expected time per request. Dynamic maintenance of a sphere-separator-based search tree with INSERT, DELETE and SEARCH requests can be done in* $O(\log^3 n)$ *incremental expected time per request.*

We prove the theorem in parts, each using a separate lemma for each. The details of the proof are given in the following subsections. Lemmas 5.4 and 5.5 in

Section 5.3.1 deal with the case where a separator is maintained for the point set, and describe the algorithm for inserting and deleting points from the data set. Lemma 5.6 in Section 5.3.2 proves the results for dynamically maintaining separator trees and the algorithms for inserting and deleting points from the data structure.

5.3.1 Maintaining a Separator

Recall that the separator of a set of n points is determined by examining a random sample S of the points of size $\sigma(n) = O(\log n)$. Our algorithm keeps track of the points selected to be in S.

The algorithm proceeds inductively. Given a point set and an associated separator, an update to the input set (INSERT or DELETE) may cause the algorithm to calculate a new separator. Alternately, the separator may remain the same, despite an addition or deletion of a point. The key point is that

> After a sequence of updates to the point set, the separator output by the dynamic algorithm comes from the same probability distribution as a separator output by the static algorithm, given the updated point set.

Once we have a sphere separator for the point set, we can determine whether a query point is inside or outside the separator in $O(1)$ time.

Insertion

We now analyze the time needed to insert a single point into the point set, subject to the condition above. The analysis is a special case of the general analysis given in Section 2.3.3, and follows the same outline.

When a request for insertion arrives, the input to that stage of the algorithm is a set of $n-1$ points P, a separator for P partitioning it into P_E and P_I (points on the exterior and interior, respectively, and one additional point p'. Consider the situation where all n points are presented to the static algorithm. The only step of the static algorithm which is important for our purpose is selecting the set S which

is used to determine the separating sphere. If $|S| = \sigma(n)$, the probability that the new point had been included in S is $\frac{\sigma(n)}{n}$. Thus we perform a single Bernoulli trial, with probability of success $1 - \frac{\sigma(n)}{n}$.

Success in the Bernoulli trial means that there will be no change to the separator. The only thing the algorithm does in this case is determine whether p' is in P_I or P_E. That process is straightforward. Look at the separating sphere, and determine whether p' lies inside or outside it.

Failure in this trial means that p' needs to be included in the subset of points that will determine the separating sphere. In that case, we retire the replicant and invoke the static randomized linear time algorithm of Lemma 5.2 on the entire set of n points. That is, we perform SPHERE-SEPARATOR(S) on the set $P \cup \{p'\}$.

In reality, static algorithms are costly. The static algorithms for finding sphere separators [76] use recursive randomized sampling. When constructing a separator structure [22, 76], there is a cost involved in building the tree even if the new input has no effect on the separator at the root.

Using a sample of size $\sigma(n) = O(\log n)$ to calculate the separating sphere, we get

Lemma 5.4 *Given a set P of $n-1$ points in R^d and an associated δ-splitting sphere separator, the expected incremental time to insert a new point p' into P is $O(\log n)$.*

Proof: The probability of needing to recompute a sphere separator for $P \cup \{p'\}$ is the same as the probability of including p' in the sample used in selecting a separator, which is $\frac{\sigma(n)}{n}$. With that probability, we invoke the static, linear time algorithm for computing a sphere separator for $P \cup \{p'\}$. Otherwise, we determine (in constant time) which side of the separator the point lies in, and add it to that set (again in constant time). The expected update time is thus

$$E[T_D(n)] \leq \frac{\sigma(n)}{n} O(n) + \left(1 - \frac{\sigma(n)}{n}\right) O(1)$$

$$\leq \frac{\sigma(n)}{n} O(n) + O(1)$$

$$= O(\sigma(n)) = O(\log n).$$

□

This result is consistent with the general result stated in Section 2.3.3 and the bounds in table 2.1.

Deletion

Using very similar reasoning, the dynamic separator algorithm also supports point deletion.

As noted above, the data structure storing the separator resembles the leaf model, in that all the points are associated with either the interior or the exterior of the sphere. When deleting a point, it does not really matter if the point was used in the initial sample selected to determine the sphere, because once the sphere is calculated the sample points are treated just like all the other points. Deleting even all of the sample points should have no effect on the goodness of the sphere separator with relation to the other points. Deletion proceeds in exactly the same fashion as insertion, with the same time bounds. The proof of the following lemma is identical to the proof of lemma 5.4.

Lemma 5.5 *Given a set P of n points in R^d and an associated δ-splitting sphere separator, the expected time to delete a point p from P is $O(\log n)$.*

5.3.2 Maintaining a Separator Tree Dynamically

If the entire separator structure is maintained, then the new point p' is added recursively to the left subtree or right subtree, depending on its location with respect to

the separating sphere. The algorithm proceeds inductively as before. Given a separator search structure, the entire structure or a part of it may be completely rebuilt using the static algorithm with each new insertion or deletion. After every step, the following holds:

> After a sequence of updates to the point set, the separator tree output by the dynamic algorithm comes from the same probability distribution as a separators tree output by the static algorithm, given the updated point set.

Once we have a separator tree for the point set, a SEARCH request can be completed in $O(\log n)$ time.

Analysis of the time it takes to update the separator tree follows the same outline as the analysis of an update when maintaining a separator. The key difference is that here we need to keep recursively update subtrees, stopping the recursion when a leaf is reached or when the subtree needs to be rebuilt. Insertion and deletions are processed identically. The following lemma gives the time bound for performing an update on a separator tree for a set of points. The results are consistent with the time bounds in Section 2.3.3.

Lemma 5.6 *The expected time to perform an insertion of a new point p' into an existing sphere separator tree of a set of $n-1$ points in R^d, is $O(\log^3 n)$. The expected time to perform a deletion of a point p from an existing sphere separator tree of a set of n points in R^d is $O(\log^3 n)$.*

Proof: At the top level, the probability of recalculation of the sphere separator, which would cause us to rebuild the entire structure, is the same as the probability of including p' in the sample used in selecting a separator, which is $\frac{\sigma(n)}{n}$. With that probability, we invoke the static algorithm which constructs a new separator tree $O(n \log n)$ time. Otherwise, the time required will be the expected time to rebuild a subtree. Since the separators found using the static algorithm are guaranteed to

$\left(\frac{d+1}{d+2}+\epsilon\right)$-split the point set, the size of a subtree is at most δ, where $\delta = \frac{d+1}{d+2}+\epsilon$. Thus we get the following recurrence equation for the expected time to maintain the separator tree:

$$E[T_D(n)] \leq \frac{\sigma(n)}{n} O(n \log n) + \left(1 - \frac{\sigma(n)}{n}\right) T_D(\delta n)$$

$$\leq \frac{\sigma(n)}{n} O(n \log n) + T_D(\delta n) =$$

$$O(\sigma(n) \log n) + T_D(\delta n) = O(log^2 n) + T_D(\delta n) = O(\log^3 n).$$

□

5.4 High Likelihood Time Bounds for Maintaining Sphere Separators

The separator algorithm described in Section 5.3.1 gives "expected" results in two senses. The separator is only expected to be good with some probability, and the algorithm only attains the expected time bound with some probability.

5.4.1 Good Separator with High Likelihood

Due to the randomized nature of the separator algorithm [22], using a sample of size $O(\log n)$ means that with probability $\frac{1}{\log n}$ the sphere separator will not be "good", in that it may intersect a large number of balls. If that is the case, a slower $O(n \log n)$ algorithm for the separator is invoked to perform the correction. This generates a high likelihood time bound of $O(n \log n)$, trading time for a guarantee on the goodness of the separator. Probabilistic analysis [22] shows that if the the slower, high likelihood $O(n \log n)$ time algorithm is only invoked periodically (i.e. at the expected rate of "failure" $\frac{1}{\log n}$), then the entire algorithm is only slowed down by a constant factor; this yields a high likelihood time $O(n)$ algorithm for maintaining the separator.

Alternately, we can repeat the calculations sufficient times to ensure that the probability of failure is very small, or $n^{-\alpha}$. Since the probability of failure in k repeat calculations is $1 - \frac{1}{\log^k n}$, we can guarantee success with probability $1 - n^{-\alpha}$ by selecting $k = \alpha \log n / \log \log n$. Thus we can find a good separator with high likelihood in $O(\log^2 / \log \log n)$.

5.4.2 Good Time Bounds with High Likelihood

In order to attain high likelihood time bounds for our dynamic separator algorithm, we apply our replicant paradigm to the algorithm. Applying the results from Chapter 3, we get a high likelihood time bound $O(\log^3 n)$ for dynamically maintaining the separator. We show the following:

Theorem 5.7 *There exists a dynamic algorithm for maintaining the sphere separator of a set of dynamically changing points which works with high likelihood time bounds of* $T_H(n) = O(\log^3 n)$.

Proof: The static algorithm for generating a sphere separator has time bounds of $O(n)$. The size of the sample selected to generate the separator is $O(\log n)$. Table 3.1 shows us that the number of replicants necessary to attain high likelihood time bounds in this case is $O(\log^2 n)$. This is the slowdown penalty we pay for the high likelihood time bounds. Since each update in our dynamic algorithm takes $O(\log n)$ time, the total time, including slowdown, required by the high likelihood algorithm is $O(\log^3 n)$. □

Chapter 6

Parallel Nested Dissection

This chapter is free-standing with respect to the work in the rest of this thesis. It deals with a practical use for graph separators, a method for solving large, sparse matrix systems called *nested dissection.* Nested dissection operates on the graph of the matrix, separating it into components using graph separators. Each of the subproblems is then solved and the solutions are combined to give a solution for the entire system. Since each of the separated subgraphs is independent of the others, nested dissection is inherently parallelizable and efficient algorithms have been developed that perform parallel nested dissection (PND) on various parallel architectures. Nested dissection is useful not only in matrix computations; it has proved to be the cornerstone of path algebra problems [58], minimum cost path calculations [59] and shortest path algorithms [13].

We describe the following improvements to the known PND algorithms :

1. A fast PND algorithm for the PRAM model, which reduces the time bound by a factor of $O(\log n)$, without significantly increasing the processor bound (FAST PND).

2. A PND algorithm that works on a mesh-connected processor array, applicable to matrices representable by a grid graph, using $O(n)$ processors and taking $O(\sqrt{n})$ time, which reduces the space bounds required by the algorithm to a constant factor of the size of the input matrix, without significant increase in time or processor bounds (COMPACT PND).

3. A PND algorithm that works on a mesh-connected processor array, using $O(n)$ processors and taking $O(\sqrt{n})$ time, for the more general case of matrices representable by a graph of constant degree and separator size $\sqrt{n}$ (as required for many 2D PDE applications).

In addition to these practical results, we show, using known theoretical results about parallel matrix multiplication, that it is theoretically possible to achieve tighter bounds on the amount of work performed by PND algorithms. We also show how our algorithms generalize to solve all-pairs minimal-cost path problems, within the same complexity.

6.1 Motivation

The problem of solving very large sparse linear systems arises often in scientific computing. Much research has been devoted to finding efficient solutions to this problem. Researchers have been able to develop fast sequential algorithms for solving linear systems by exploiting their sparsity and structure. Nested dissection is one of the efficient algorithms developed, particularly for the important class of sparse linear systems that arise from finite element problems in two dimensions.

The original work on the nested dissection method for matrices representable by grid graphs is due to George [25], and was extended to general separable graphs by Lipton, Rose, and Tarjan [37]. The tightest and most comprehensive analysis of nested dissection is due to Rose and Whitten [69].

With the advent of parallel computing, research focus has shifted toward finding parallel algorithms, and especially parallel versions of known sequential algorithms (see [31] for a review and extended bibliography). In particular, considerable research has been devoted to developing parallel nested dissection (PND) algorithms. The goal of these research efforts is to find algorithms that are both fast and work efficient.

Birkhoff and George [8] predicted that a parallel implementation for a sparse matrix representable by a grid graph would be able to achieve an $O(\sqrt{n})$ parallel time bound on an n processor mesh-connected machine. Indeed, several parallel versions of the nested dissection algorithm have been suggested, taking into account different parallel models [56] and architectures [80, 49, 79]. More recently, work has been done on parallel Choleski factorization for dense [26] and sparse [28] matrices. Pan and Reif [56] presented a PRAM algorithm that uses $O(n^{3/2})$ processors, takes

$O(\log^3 n)$ time, and is applicable to the class of $O(\sqrt{n})$ separable graphs. They later generalized this result to the class of $s(n)$ separable graphs [61], giving an algorithm that uses $O(P(s(n)))$ processors (see definition of $P(n)$ in the next paragraph), and takes $O(\log^3 n)$ time. This is the best known result to date for the PRAM model.

At the heart of all these algorithms remains the process of multiplying two matrices $n \times n$ in parallel. If we wish to perform this multiplication in $O(\log n)$ parallel steps on the PRAM model, we need $P(n)$ processors, where $P(n) = n^\omega$, $2 < \omega < 3$. Currently, the best known value for ω is 2.376... [75, 15], however a practical bound for ω is at best 2.81. This is the processor bound used by several parallel nested dissection algorithms on the PRAM model.

6.1.1 Open Problems with Existing Algorithms

Time Reductions

Research efforts have concentrated mostly on improving time bounds and reducing the processor count of each version of parallel nested dissection, thereby reducing the overall work performed. Even so, a major open problem remains:

- On the PRAM model, reducing the work bound by dropping the time bound to $O(\log^2 n)$ while maintaining a processor bound of $P(n)$.

This parallel bound was shown by Pan to hold for computing dense matrix inverse [54, 55], using a reduction to the computation of the characteristic polynomial or a related form. However, while theoretically possible, this method is numerically unstable, and thus of little practical value. The problem of developing a FAST PND attaining these bounds is an open problem dating to the work of [56]. Using a pipelining technique, Pan and Reif [57] achieved a time bound of $O(\log^2 n)$ for solving minimum path problems using PND, but this speedup does not hold for linear systems. For linear systems, Gazit and Miller claim to have reduced the time bound of PND to $O(\log^2 n \log\log n))$ [24].

Storage Reductions

Despite these problems, parallel nested dissection methods can exceed the speed of iterative methods, for general 2D problems (for restricted 2D problems, such as constant degree linear PDE, parallel multigrid can be more time efficient; see [60]). However, there is an additional parameter involved when analyzing the practical aspects of an algorithm, which is the amount of storage necessary. This is particularly true when one considers the memory constraints of existing parallel machines.

Whereas iterative methods generally need only a constant factor of additional storage, the known PND implementations are not space efficient and require at least a logarithmic factor of additional storage over the input matrix. For moderate size sparse matrices, with 1,000 variables, this relative factor is already 10. Since many of the commercial massively parallel machines are severely space constrained, this is a key problem that has restricted practical implementations of PND to relatively small size matrices. For example, see [49] for a description of an implementation on the 64,000 processor Connection Machine at Thinking Machines, Inc., and see [36] for the implementation on the 16,000 Processor Massively Parallel Processing Machine (MPP) at NASA Goddard Space Center. This leads to the following problem:

- Reducing the space bounds to a constant factor of the input matrix, while maintaining the time and processor bounds.

We propose an interesting modification of PND algorithm, which we call COMPACT PND, which uses only a small constant factor of extra storage (beyond the matrix input) without compromising the asymptotic time bounds of the algorithm. This result holds for implementations on a grid- or mesh-connected machine when the graph of the matrix has bounded degree and an $O(\sqrt{n})$ separator set.

Specialized Matrix Classes

A third issue, in addition to reducing work and space bounds, is the class of matrices that can be solved using PND. On the mesh connected processor array, all existing implementations have concentrated on matrices whose underlying graphs are grids. This leaves the following open problem:

- On a mesh-connected processor array, implementing PND using $O(n)$ processors and taking $O(\sqrt{n})$ time for the important and more general case where the graph of the matrix is of constant degree and separator size $\sqrt{n}$.

This requires us to extend the previous work of Birkhoff and George [8], which is restricted to matrices with grid graphs, and also of Pan and Reif [56], which is inefficient with respect to work by a polylog factor.

6.1.2 Organization of this Chapter

Section 6.2 provides an overview of nested dissection and parallel nested dissection. Section 6.3 describes the FAST-PND PRAM algorithm. Section 6.4 describes implementations of the PND algorithm on a grid-connected architectures (meshes). In Section 6.5 we describe COMPACT PND, an algorithm for reducing the space bounds of PND implementations on meshes. In Section 6.6 we generalize COMPACT PND to linear systems with small separators. We extends PND to path problems in Section 6.7. Open problems are posed in Section 6.8.

6.2 Nested Dissection

This section provides an overview of sequential and parallel nested dissection.

6.2.1 Definitions

Consider a system of linear equations

$$Ax = b$$

where A is an $n \times n$ *symmetric positive definite* matrix, b is a vector and x is the unknown solution vector to be computed.

The graph of a matrix

We define the graph of A to be an undirected graph $G(A) = (V, E(A))$, where $V = \{1, 2 \ldots n\}$ and $E(A) = \{(i,j) | a_{i,j} \neq 0\}$. Each variable (row or column in the matrix) is a vertex in the graph, and a non-zero entry in the matrix corresponds to an edge. For example when solving n second order elliptic partial differential equations by using a finite element mesh, the graph of the matrix is a $\sqrt{n} \times \sqrt{n}$ grid. In the rest of this chapter, we will refer to the matrix and its graph interchangeably.

Without loss of generality, we assume throughout this chapter that $\sqrt{n}$ is of the form $2^d - 1$.

Solving a System of Equations

Finding a solution to a system of equations $Ax = b$ by Gaussian elimination consists of two steps. First, factor the matrix A into the product of three matrices, $A = LDL^T$ where L is lower triangular and D is diagonal. Next, solve the simplified system $LDL^Tx = b$ by solving three simpler problems: $Lz = b$, $Dy = z$ and $L^Tx = y$.

Fill-In

When A is sparse, we would like to take advantage of its sparsity when solving the system. When factoring A into LDL^T, a key issue is to limit *fill-in* - the creation of more non-zero entries in L than were in A. If L is not sparse, the backsolve step becomes more complex. The higher the fill-in, the less we can take advantage of the sparsity of A, and the more time and space will be necessary for factoring and solving the system.

A sparse matrix is represented by a graph with few edges. Elimination of a variable corresponds to the removal of a vertex and its incident edges from the graph, and adding edges between its neighbors (see Figure 6.1). When new edges are added to the graph, fill-in occurs. As more edges are added, the graph will no longer be sparse. Nested dissection is a way of ordering the variables in such a way that will limit the

amount of fill-in in the matrix, and consequently the number of edges in its graph.

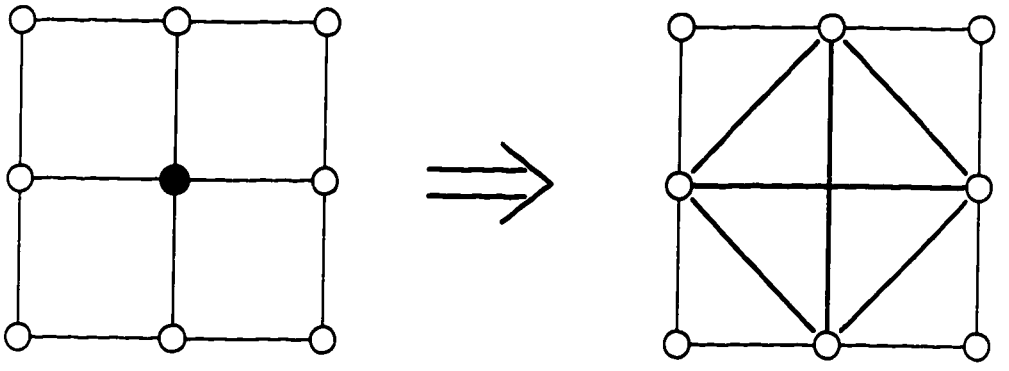

Figure 6.1: Elimination of the variable corresponding to the black vertex results in the graph on the right

6.2.2 Sequential Nested Dissection

The idea of nested dissection (ND) was first introduced by George [25]. He showed that a system of n equations whose matrix had an underlying $\sqrt{n} \times \sqrt{n}$ grid structure could be solved in $O(n^{3/2})$ sequential time using $O(n \log n)$ space.

In the factoring stage of the nested dissection algorithm, matrix A is actually factored into a product of $n-1$ lower triangular matrices, a diagonal matrix and the transposes of the triangular matrices, as follows:

$$A = L_1 \ldots L_{n-1} D L_{n-1}^T \ldots L_1.$$

This is performed in steps. First, A is factored into

$$A = L_1 A_1 L_1^T$$

where

$$A_1 = \begin{bmatrix} d_1 & 0 \\ 0 & B_1 \end{bmatrix}$$

and B_1 is an $(n-1) \times (n-1)$ matrix.

Next, A_1 is factored into

$$A_1 = L_2 A_2 L_2^T$$

where

$$A_2 = \begin{bmatrix} d_1 & 0 & 0 \\ 0 & d_2 & 0 \\ 0 & 0 & B_1 \end{bmatrix}$$

and B_1 is an $(n-2) \times (n-2)$ matrix.

In the i^{th} factorization step, A_{i-1} is factored into

$$A_{i-1} = L_i A_i L_i^T$$

where

$$A_i = \begin{bmatrix} D_i & 0 \\ 0 & B_i \end{bmatrix},$$

D_i is a diagonal $i \times i$ matrix, and B_i is an $(n-i) \times (n-i)$ matrix yet to be factored.

Finally,

$$A_{n-1} = D.$$

The k^{th} factorization step is called the *elimination of variable* x_k.

When referring to $G(A)$, the graph of A, the elimination of the variable x_k is equivalent to removing vertex k and its incident edges from the graph and adding an edge between each pair of its neighbors. For every pair of vertices u, v which are neighbors of k, we add an edge (uv) to the graph if it does not already exist. This process essentially "short-circuits" k out of the graph. The vertex is removed and its neighbors now form a *clique* – a subgraph in which there is an edge between every two vertices. Each added edge corresponds to a new non-zero entry in one of the factorization matrices – the fill-in which we seek to minimize.

George presented an ordering scheme for eliminating the variables, which bounds the fill-in in the L matrices to $O(n \log n)$. For a positive, symmetric definite matrix representable by a grid graph, this scheme involves partitioning the $\sqrt{n} \times \sqrt{n}$ grid into four $\frac{\lfloor\sqrt{n}\rfloor}{2} \times \frac{\lfloor\sqrt{n}\rfloor}{2}$ sub-grids by using a "cross" through the central row and column of the grid as a separator. Let the set of vertices in this separator cross be called S_1. If we remove the vertices of S_1 and their incident edges from the graph, we are

left with four separate sub-grids. The sub-grids are *independent*, that is, there are no edges between them, and each one is a "stand-alone" grid. Each of the four sub-grids is now partitioned into four again, using four separator crosses, one in each sub-grid. Let S_2 be the set of vertices in all four separator crosses. We continue this process of subdividing the graph until we have independent sub-grids consisting of at most one vertex. The last set of separators is S_{d-1}, and the remaining vertices of the graph make up S_d. This process partitions the vertices of the grid into d subsets. Figure 6.2 shows the 7×7 grid graph and the phases in which vertices are eliminated.

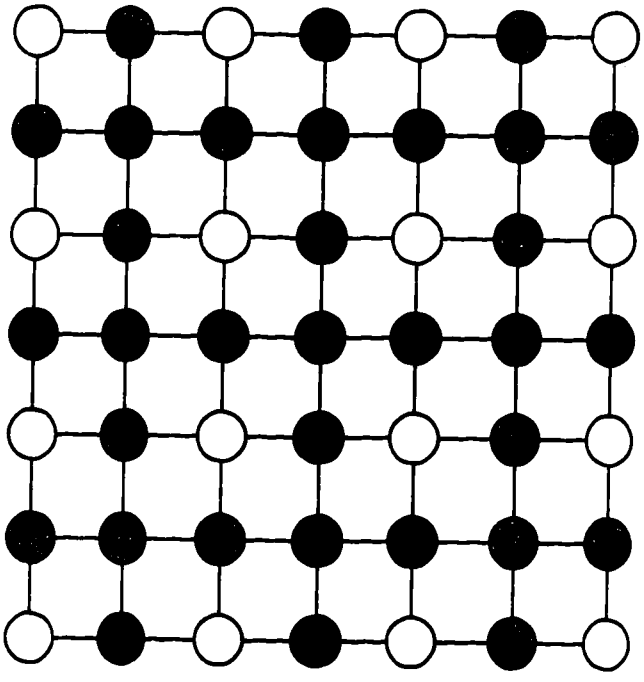

Figure 6.2: The 7x7 grid graph. Vertices eliminated in the same phase have the same color

Order of Elimination of Vertices

The elimination of variables proceeds up the separator tree, starting with the vertices at the leaves and ending with the vertices at the root. The first vertices to be eliminated are the vertices in S_d. Note that no two vertices of S_d share an edge; they are independent of each other. Because these variables are independent, the order in which they are eliminated is not important. The elimination of all the S_d vertices is

called the first *elimination phase* of the factorization. During the second elimination phase, the vertices in S_{d-1} are eliminated. In the next phase we eliminate the vertices in S_{d-2} and so on until, in the last phase, the vertices in S_1 are eliminated. Figure 6.3 shows the graph in Figure 6.2 after the first elimination phase.

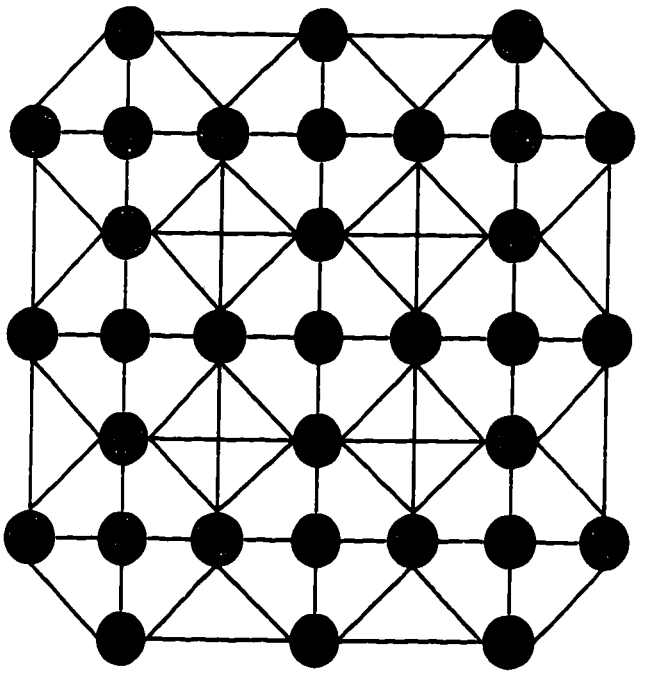

Figure 6.3: The 7x7 grid graph after the first elimination phase.

Separator Tree

It is useful to think of the vertices of $G(A)$ as placed in a data structure called a *separator tree*. At the root of the tree are the vertices of S_1. The root has four children. Each subtree contains the vertices of a sub-grid, and recursively follows the same structure, i.e., the separator vertices are in its root and the vertices of each independent sub-grid are in a subtree. Each level of the separator tree contains the vertices of one of the S_i, with the S_1 vertices at the root and the S_d vertices in the leaves. This is a process similar to that illustrated by Figure 2.1 on page 22.

We can state the following about the separator tree:

Lemma 6.1 *The depth of the separator tree, d, is at most* $\frac{1}{2}\log n$.

Proof: Let r be the set of vertices in the top row of the mesh. The set r contains $\sqrt{n}$ vertices. The first cross, S_1, has one vertex in r, splitting r into two. S_2 contributes two vertices, splitting each of the halves, and so on. The contribution of all separators and remaining sub-grids add up to the entire row. In other words, $\sum_{i=1}^{d} 2^i \leq \sqrt{n}$. Solving this inequality, we get $d < \frac{1}{2}\log n$. □

6.2.3 Parallel Implementation

The key to parallelizing the nested dissection algorithm is the realization that during factorization, vertex eliminations within each elimination phase are independent of each other, and thus can be performed in parallel. This corresponds to working on the subtrees of the separator tree, which are independent of each other. Several parallel versions of nested dissection have been suggested, some [80, 49, 79] achieving the $O(\sqrt{n})$ time bounds predicted by Birkhoff and George in [8].

6.2.4 Generalized Parallel Nested Dissection

The algorithm given by Pan and Reif in [56], operating on a more general class of graphs, works on a PRAM model using $O(n^{3/2})$ processors and takes $O(\log^3 n)$ time. The separators in this class of graphs are not as straightforward to construct as in the case of grid graphs. However, the size of the separator set is still $O(\sqrt{n})$ and each induced subgraph has at most $2n/3$ vertices. As in the grid, the graph is separated into two independent subgraphs using, as a separator, a set of vertices S_1, where $|S_1| \leq O(\sqrt{n})$. Each subgraph is separated again, and the set of vertices in all these separators makes up S_2. The process continues until all that is left of the graph are independent subsets of at most one vertex. S_d is the set of all these subsets. The depth of the separator tree is $O(\log n)$. Throughout this chapter, as in [56], we shall assume that the structure of the separator tree is known (calculated) in advance before the start of the nested dissection algorithm. This does not add to the complexity of the algorithm.

Since much of the work is done in parallel, the factorization is not done column by column as in the sequential case, but rather in blocks, where each small block is factored in turn. The smaller blocks are factored in parallel. Adopting the method used by Pan and Reif [56], we define the factorization recursively over a sequence of matrices $A_0, A_1, \ldots, A_d$. For $h = 0, 1, \ldots, d-1$, each matrix has the structure

$$A_h = \begin{bmatrix} X_h & Y_h^T \\ Y_h & Z_h \end{bmatrix}$$

where the X_h matrix is block diagonal, $Z_h = A_{h+1} + Y_h X_h^{-1} Y^T$ is the part of the matrix yet to be factored, and each block in the X_h matrices is further factored in a later stage of the algorithm.

The factorization of each A_h can be written as an LDL^T factorization:

$$A_h = \begin{bmatrix} I & 0 \\ Y_h X_h^{-1} & I \end{bmatrix} \begin{bmatrix} X_h & 0 \\ 0 & A_{h+1} \end{bmatrix} \begin{bmatrix} I & X_h^{-1} Y_h^T \\ 0 & I \end{bmatrix}$$

Note that here, D is block diagonal.

It follows that

$$A_h^{-1} = \begin{bmatrix} I & -X_h^{-1} Y_h^T \\ 0 & I \end{bmatrix} \begin{bmatrix} X_h^{-1} & 0 \\ 0 & A_{h+1}^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -Y_h X_h^{-1} & I \end{bmatrix}$$

Thus the problems of factoring A_h and of inverting it for the backsolve are reduced to factoring and inverting much smaller block diagonal matrices.

6.3 Better Time Bounds: FAST PND

In this section, we describe a $\log n$ factor improvement to the time bounds of the PRAM PND algorithm. We achieve this by grouping together several elimination phases. Looking at the separator tree, we see that at each level we break up the graph into two roughly equal parts, using a separator of size $\sqrt{n}$. This gives a tree of depth $O(\log n)$. The algorithm recursively works on each level of the tree. We can

therefore conceive of a *recursion tree*, reflecting the separator tree, which describes the work of the algorithm. The depth of this recursion tree is also $O(\log n)$. The time necessary to perform PND with this algorithm is determined by the number of recursive calls and by the time to perform each one.

Gazit and Miller [24] first suggested reducing the depth of the recursion tree by grouping together several levels of the tree and performing the work on a group of levels together. They were able to limit the depth of the tree to $O(\log \log n)$. Our idea is similar in that we, too, group levels together into *multilevels*, and perform the work on each multilevel of the tree all at once (see Figure 6.4).

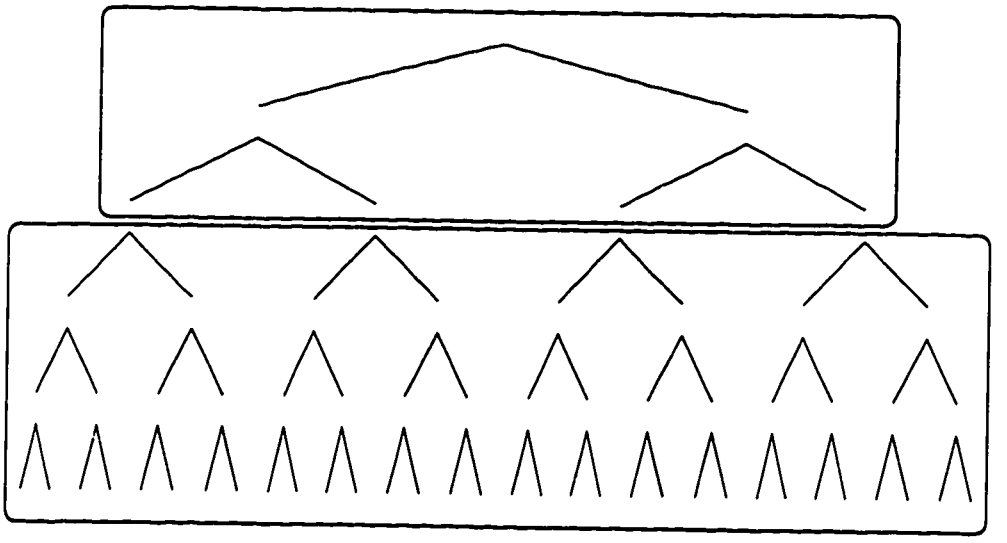

Figure 6.4: Grouping tree levels into multilevels. The work inside each rectangular box is done in parallel. The number of levels in each multilevel depends on the size of the input, and is defined recursively in terms of the size of the previous multilevel.

We introduce a new parameter ϵ, an arbitrarily small constant $0 < \epsilon < 1/2$. The number of multilevels in our recursion tree will be bounded by $\log \frac{1}{\epsilon}$, and the number of processors used will be $O(n^{\omega/2+\epsilon})$. The key idea, which allows us to bound the number of multilevels by a constant, is that *the size of each multilevel is not fixed; rather, the size of each multilevel is a function of* n, ϵ, *and the size of the multilevels above it in the tree.*

Since the structure of the separator tree is known in advance, work performed at each multilevel involves only the elimination of separator vertices from several levels.

More work is being done at each multilevel than at any single level in the original tree. However, since the subproblems are independent, we can, in the PRAM model, spread the extra work among more processors. Specifically, we require a factor of n^ϵ more processors to perform the extra work. We show the following:

Theorem 6.2 *Parallel nested dissection of a symmetric positive definite matrix can be performed on a PRAM model in $O(\log^2 n)$ time using $O(n^{\omega/2+\epsilon})$ processors, where the constant factor in the time bound is a logarithmic function of $1/\epsilon$.*

Proof: The depth of the recursion tree is initially $\log n$. We label the levels of the tree from 1 (root) to $\log n$ (leaves). Let $k_0 \ldots k_m$ be a sequence of strictly increasing integers over $[0 \ldots \log n]$, where $k_0 = 0$ and $k_m = \log n$. The levels of the tree are grouped into m multilevels, where k_j is the deepest tree level in multilevel j. We define multilevel K_j of the recursion tree to be the aggregate of levels $[k_{j-1}+1 \ldots k_j]$ of the tree.

As an example, consider the tree in figure 6.4. The tree has six levels, grouped into two multilevels. Using the terminology defined above, the k_m sequence is $k_0 = 0, k_1 = 2, k_2 = 6$. The two multilevels are $K_1 = [k_0 + 1 \ldots k_1] = [1 \ldots 2]$ and $K_2 = [k_1 + 1 \ldots k_2] = [3 \ldots 6]$.

The work in multilevel K_j consists of eliminating all vertices of the separators in these levels. The number of levels in each multilevel is a function of the size of the level above it.

To clarify the analysis that follows, let us first examine what happens in the algorithm if there is no grouping of levels. When working on a single level of the recursion tree, we eliminate the separator vertices of several subgraphs in parallel. In level i, the number of subtrees is $g_i = 2^{i-1}$. Let v_i be the number of separator vertices in each subgraph. Since the separators are of size $\sqrt{n}$, $v_i \leq \frac{\sqrt{n}}{2^{i-1}}$. The work involved in eliminating the separator vertices in level i is the work necessary to eliminate all separator vertices in each subgraph. This is dominated by the work necessary to invert g_i matrices of v_i variables each. Since inverting a matrix of n variables takes

$O(n^\omega)$ work (with $2 < \omega < 3$), the total work for eliminating the separator vertices in level i is

$$g_i O(v_i^\omega) \leq 2^{i-1} O((\sqrt{n}/2^{i-2})^\omega) \leq 2^{i-1} O(n^{w/2}).$$

When grouping levels into a multilevel, separator vertices from several levels are eliminated in parallel. Since a multilevel is defined to be the aggregate of several adjacent levels, eliminating vertices in a multilevel results in a graph identical to the graph resulting from eliminating the separator vertices in each of the levels sequentially. The number of subtrees in a multilevel is the number of subtrees in the highest level of the multilevel, and the number of separator vertices in each subtree is the total of all separator vertices in all levels of the subtree within the multilevel.

Looking at multilevel K_j, which is the aggregate of levels $[k_{j-1}+1 \ldots k_j]$, each subgraph is a subtree of subgraphs. The number of subgraphs is the number of roots in the highest level, or $g_{k_{j-1}+1} = 2^{k_{j-1}}$. The number of separator vertices in each subgraph is the total number of separator vertices in all levels of the multilevel, or

$$\sum_{i=k_{j-1}+1}^{k_j} 2^{i-k_j-1}\sqrt{\frac{n}{2^{i-1}}} = \sum_{i=0}^{k_j-k_{j-1}-1} 2^i \sqrt{\frac{n}{2^{i+k_{j-1}}}}.$$

The total work performed in multilevel K_j is bounded by the work necessary to eliminate all the separator vertices in the multilevel, multiplied by the number of subgraphs in the multilevel. This is

$$2^{k_{j-1}} O\left(\left(\sum_{i=0}^{k_j-k_{j-1}-1} 2^i \sqrt{\frac{n}{2^{i+k_{j-1}}}}\right)^\omega\right). \tag{6.1}$$

Evaluating the summation inside the power term, we get

$$\sum_{i=0}^{k_j-k_{j-1}-1} 2^i \sqrt{\frac{n}{2^{i+k_{j-1}}}} = \sqrt{\frac{n}{2^{k_{j-1}}}} \sum_{i=0}^{k_j-k_{j-1}-1} 2^{i/2} =$$

$$\sqrt{\frac{n}{2^{k_{j-1}}}}(2^{(k_j-k_{j-1})/2}-1)(\sqrt{2}+1) = (\sqrt{2}+1)\sqrt{n}2^{(k_j-2k_{j-1})/2}.$$

Inserting this back into equation 6.1, we get an equation for the total work performed in multilevel K_j, which is, for a constant c:

$$2^{k_{j-1}} c \left[\sqrt{n} 2^{(k_j - 2k_{j-1})/2}\right]^{\omega} = c n^{\omega/2} 2^{\omega k_j/2 - (\omega-1)k_{j-1}} = n^{\omega/2} 2^{\omega k_j/2 - (\omega-1)k_{j-1} + \log c}.$$

We can spread this work over $n^{\omega/2+\epsilon}$ processors for any $0 < \epsilon < 1/2$, getting

$$n^{\omega/2+\epsilon} = n^{\omega/2} 2^{\omega k_j/2 - (\omega-1)k_{j-1} + \log c}$$

and thus

$$\epsilon \log n = \omega k_j/2 - (\omega - 1)k_{j-1} + \log c.$$

Solving this equation for k_j, we get:

$$k_j = \frac{2}{\omega}(\epsilon \log n + (\omega - 1)k_{j-1} - \log c).$$

Substituting $c' = \frac{2}{\omega} \log c$, and solving this recurrence to get k_j in terms of n and ϵ, we get

$$k_j = \left(\frac{2\epsilon}{\omega - 2} \log n - \frac{c'\omega}{(\omega - 2)}\right)\left(\left(\frac{2(\omega-1)}{\omega}\right)^j - 1\right). \tag{6.2}$$

Recall that k_j is the deepest level in multilevel K_j. From this, we can determine the size of the multilevel K_j, which is $k_j - k_{j-1}$.

We can now use the information on the size of the multilevels to calculate how many multilevels there are in the recursion tree. Recall that there are m multilevels in the tree, with k_m being the deepest level of the deepest multilevel. This is also the deepest level of the recursion tree, which is level $\log n$. From equation 6.2 above we have

$$\log n = k_m = \left(\frac{2\epsilon}{\omega - 2} \log n - \frac{c'\omega}{(\omega - 2)}\right)\left(\left(\frac{2(\omega-1)}{\omega}\right)^m - 1\right).$$

Solving for m to find the number of multilevels in the tree, we get

$$\left(\frac{2(\omega-1)}{\omega}\right)^m = \frac{\log n + \frac{2\epsilon}{\omega-2}\log n - \frac{c'\omega}{\omega-2}}{\frac{2\epsilon}{\omega-2}\log n - \frac{c'\omega}{\omega-2}} = \frac{\omega - 2 + 2\epsilon - \frac{c'\omega}{\log n}}{2\epsilon - \frac{c'\omega}{\log n}}$$

and thus

$$m = \log_{\frac{2(\omega-1)}{\omega}} \left(\frac{\omega - 2 + 2\epsilon - \frac{c'\omega}{\log n}}{2\epsilon - \frac{c'\omega}{\log n}}\right).$$

Noting that $\omega < 3$ and $2\epsilon < 1$, we get

$$m < \log_{\frac{2(\omega-1)}{\omega}} \left(\frac{2 - \frac{c'\omega}{\log n}}{2\epsilon - \frac{c'\omega}{\log n}}\right) < \log_{\frac{2(\omega-1)}{\omega}} \frac{1}{\epsilon}.$$

Thus the number of multilevels in the tree is bounded by a constant.

Since we have only a constant number of multilevels, rather than a logarithmic number of levels, the time complexity for nested dissection on a PRAM can be reduced by a factor of $\log n$ to $O(\log^2 n)$. This completes the proof of the theorem. □

6.4 Mesh Model Implementations of PND

We now return to using nested dissection to solve linear systems representable by grid graphs. Of the existing parallel architectures, the natural one to use when implementing a solution for a problem involving a grid graph would be a grid- or mesh-connected processor array. Memory mapping is trivial for the input matrix, each variable being assigned to a processor on the grid. After some elimination takes place, there are idle processors evenly distributed throughout the grid. If a naive approach is used in the implementation, communication time between active processors grows with the number of idle processors in the grid. It is possible however, to redo the memory mapping after each elimination phase by conceptually "folding" the grid to maintain a compact and efficient structure. Worley and Schreiber [80] described an

implementation on a mesh connected processor array of n processors which works in $O(\sqrt{n})$ parallel time. However, the storage requirements of their algorithm grow as $\sqrt{n}$, i.e. there is a *per processor* multiplicative factor of increase in the storage requirements of the algorithm. Opsahl and Reif also present an implementation [49], based on the algorithm described in Section 6.2. The storage requirements for their implementation grows as a logarithmic factor of the input matrix size.

Unfortunately, even state-of-the-art parallel machines still have limited processor storage. If we wish to use them for solving practical problems, we need to give as much consideration to space-efficient algorithms as we do to designing time efficient ones.

In Section 6.5 we present a variant of the PND algorithm implementation on a grid architecture, which we call COMPACT PND. This algorithm maintains the $O(\sqrt{n})$ time bound but uses only a constant amount of storage per processor beyond the input, if there are n processors. More generally, given any number of processors, our algorithm uses only a constant factor more space beyond the input matrix size. First, let us establish the known space bounds.

6.4.1 Space Bounds

The PND algorithm described in Section 6.2 has two stages – factorization and backsolve. There are $O(\log n)$ factorization steps, and a new matrix is generated in each of these steps. As these matrices are later used in the backsolve stage, they must be saved in memory. In this section we show a simple modification to PND, as implemented on a mesh-connected machine, which allows us to save this $\log n$ space factor. This saving can be achieved because we do not need to save the information generated at each level of the recursion. Instead, we can recompute it.

Let m_i be the number of non-zero entries in the matrix L_i which is generated at level i of the recursion. Let $m = max(m_i)$. Thus the total space requirement for the PND algorithm on a grid graph is $\leq m \log n$.

By definition, the number of non-zero entries in a matrix L_i is the number of edges

in the underlying graph G_i. Therefore, we can limit our analysis to the number of edges in the graph generated at each step. Before proceeding with the exact analysis, we present some intuition. The original grid graph has n vertices, each with edges to at most 4 neighbors. In other words, the original matrix has at most $4n = O(n)$ non-zero entries. At each elimination phase, we remove a fraction of the vertices and their incident edges from the graph, and add edges between the neighbors of the eliminated vertices. The set of the neighbors of an eliminated vertex becomes a *clique* – a completely connected graph where there is an edge between every pair of vertices. When the only vertices in the graph are the vertices in S_1, the graph has $2\sqrt{n} - 1$ cliqued vertices, for a total of $2n - \sqrt{n} =$ edges, which is also $O(n)$. The careful analysis which follows shows that even though the degree of every vertex increases through the adding of new edges, the number of vertices in the graph is reduced and the total number of edges in the graph at any phase remains $O(n)$.

The following lemmas are introduced to establish bounds on m_i. Recall that S_k is the kth separator set and consists of the vertices eliminated in phase $d - k + 1$ of the factoring, and that $\sqrt{n}$ is of the form $2^d - 1$.

Lemma 6.3 *The size of the set S_k is $2^k(\sqrt{n}+1) - 3 \cdot 4^{k-1}$.*

Proof: S_1 is a cross of one row and one column in the mesh. Each row and column are $\sqrt{n}$ long, and one vertex is counted twice. S_i consists of 2^i rows and 2^i columns, but we need to subtract the middle vertex of each cross (of which there are 4^{i-1}), and all vertices that have already been counted in $S_j, j < i$. A simple counting argument yields the desired result. As further proof, observe that $\sum_{i=1}^{d} S_i = n$. □

Lemma 6.4 *The number of vertices in the mesh after $S_d, \ldots, S_{k+1}$ have been eliminated is at most $(\sqrt{n}+1)2^{k+1}$.*

Proof: After eliminating the vertices of $S_d, \ldots, S_{k+1}$ the vertices left in the mesh are those in $S_1, \ldots, S_k$. The number of vertices in those sets is

$$\sum_{i=1}^{k} [2^i(\sqrt{n}+1) - 3 \cdot 4^{i-1}]$$

$$= (\sqrt{n}+1)(2^{k+1}-2) - 2^{2k} < (\sqrt{n}+1)2^{k+1}.$$

□

Lemma 6.5 *After $S_d, \ldots, S_{k+1}$ have been eliminated, the number of edges incident on each vertex of S_k is at most $\frac{7\sqrt{n}}{2^k}$.*

Proof: As we eliminate sets of vertices from the mesh, we get a larger mesh with the vertices bordering on the "holes" in the mesh cliqued together. After i phases, each hole is a square, with $2^i + 1$ vertices along a side (see Figure 6.5). The "corner" vertices are not part of the clique, so we have $2^i - 1$ cliqued vertices along a side of the box, giving us cliques of $4(2^i - 1)$ vertices, each with degree $4 \cdot 2^i - 5$.

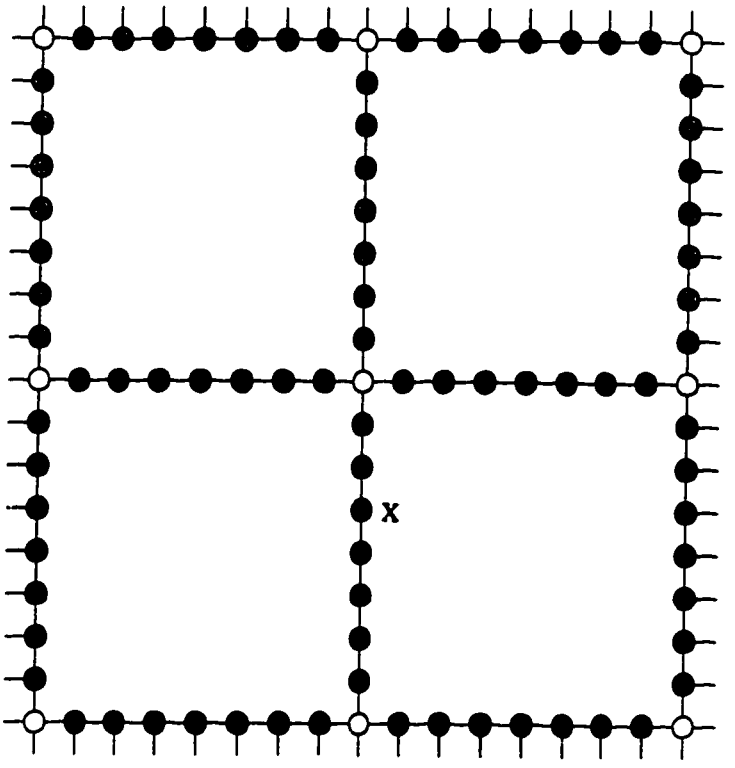

Figure 6.5: Part of an NxN grid graph after three elimination phases. The shaded vertices along the boundary of each "box" are cliqued. Neighbors of vertex X are shaded darker.

A vertex may be part of at most two cliques (if it is on the border between two boxes). The total number of vertices in both these cliques is $7(2^i - 1)$ (border is counted only once), and thus border vertices have $7 \cdot 2^i - 8$ neighbors.

If $S_d, \ldots, S_{k+1}$ have been eliminated, it means that $d-k$ elimination phases have been performed. Substituting $i = d - k = \log\sqrt{n} - k$, we find that the number of neighbors of a vertex is at most

$$7 \cdot 2^{\log\sqrt{n}-k} - 8 = \frac{7 \cdot 2^{\log\sqrt{n}}}{2^k} - 8 < \frac{7\sqrt{n}}{2^k}.$$

□

Lemma 6.6 *The number of non-zero entries in any of the L matrices is at most* $15n$.

Proof: Combining the results of the previous two lemmas, the number of edges in the graph after k elimination phases (for any k) is at most

$(\sqrt{n}+1)2^{k+1} \times \frac{7\sqrt{n}}{2^k} = 14(n+\sqrt{n}) < 15n.$ □

Since the total storage per phase is linear, and since there are $\log n$ phases, the total storage needed by an implementation which stores all intermediate results is therefore $O(n \log n)$ or $O(\log n)$ per processor.

6.5 Better Space Bounds: COMPACT PND

In this section we present a modification of the PND algorithm, which we call COMPACT PND, that requires only a small constant factor of storage over the input matrix size. In the case of an implementation on a 2D mesh connected machine and where the graph of the matrix is a 2D grid, our algorithm maintains the current known asymptotic time and processor bounds for the PND algorithm, but requires only a small constant factor of extra storage.

The issue of limiting the storage requirement associated with sequential Gaussian elimination was first raised by Eisenstadt, Schultz and Sherman [19]. They suggested,

within the context of sequential algorithms, recomputing rather than storing intermediate results as a method of saving space. We will apply this idea to the parallel implementation of the PND algorithm on a grid architecture, and prove that the space requirement is just a constant factor of the input matrix. Moreover, the time bound remains asymptotically the same.

The first stage of COMPACT PND is identical to the elimination phase of PND. It factors the matrix A into

$$A = L_0 L_1 \dots L_d D L_d^T \dots L_1^T L_0^T$$

where $d = \log n$, the L matrices are lower triangular and the D matrix block diagonal. All D and L matrices have definite bounds on the size and number of the non-zero block submatrices.

In order to solve $Ax = b$, we actually perform $A^{-1}b = x$, but instead of inverting A we invert each of the L and D matrices. We then solve

$$(L_0^T)^{-1}(L_1^T)^{-1} \dots (L_d^T)^{-1} D^T L_d^{-1} \dots L_1^{-1} L_0^{-1} b = x$$

which, if performed from right to left, is a sequence of vector-matrix multiplications.

The L_i matrices are generated in increasing order of subscripts. Thus half of the multiplications can be performed during the first stage of the COMPACT PND algorithm by:

- calculating $A_i = L_i A_{i+1} L_i^T$
- inverting L_i, and
- computing a new vector $b_{i+1} = L_i^{-1} b_i$ (where $b_0 = b$).

As a new L_i is generated, the matrix is stored over L_{i-1}. Since only one L matrix is available at any point in the calculation, when we next want to multiply b_d by $(L_{d-1}^T)^{-1}$, we will need to recalculate L_{d-1}. Similarly we will need to recalculate all other L_i matrices. While it may seem that the extra work will hamper the performance of the COMPACT PND algorithm, as compared with PND, a careful analysis

reveals that the asymptotic parallel time complexity of COMPACT PND remains the same.

Theorem 6.7 *The total time taken by COMPACT PND when recomputing rather than storing intermediate results is* $O(\sqrt{n})$.

Proof: The total time necessary to factor the full A matrix, which corresponds to a $\sqrt{n} \times \sqrt{n}$ grid, is $c\sqrt{n}$ for some constant c [80, 49]. When recalculating L_{d-1}, we perform all but the last factorization step, i.e. we are factoring four $\frac{\sqrt{n}}{2} \times \frac{\sqrt{n}}{2}$ grids *in parallel*, which requires $c\frac{\sqrt{n}}{2}$ parallel time. Similarly, recalculating L_i involves factoring the matrices of $\frac{\sqrt{n}}{2^{d-i}} \times \frac{\sqrt{n}}{2^{d-i}}$ grids, which takes $c\frac{\sqrt{n}}{2^{d-i}}$ parallel time. It follows that the total time needed by COMPACT PND is

$$\sum_{i=0}^{\log \sqrt{n}} c\frac{\sqrt{n}}{2^i} \leq 2c\sqrt{n}.$$

□

Despite the extra work performed, the asymptotic time bounds of COMPACT PND are the same as for PND. Moreover, the actual running time of COMPACT PND is only slowed down by a factor of less than two. The space savings of COMPACT PND, on the other hand, are substantial:

Theorem 6.8 *The COMPACT PND algorithm has the same time bounds as the PND algorithms, and requires only a constant amount of storage per processor.*

Proof: From lemma 6.6, we know that the number of non-zero entries in any matrix calculated by the algorithm is bounded by $15n$. On a mesh-connected $\sqrt{n} \times \sqrt{n}$ processor array, this amounts to constant amount of storage per processor. The time bound is the same as PND, within a factor of two, as explained above. □

6.6 GENERALIZED COMPACT PND

The previous sections concentrated on matrices whose underlying graph had a grid structure. For these matrices, we could exploit the advantages extended by using a grid architecture to solve the system of equations in parallel on the mesh. Because of the regular structure of the matrix, load balancing could be maintained throughout all stages of the algorithm. However, the nested dissection algorithm is applicable to other classes of graphs. Lipton, Rose and Tarjan [37] generalized the sequential algorithm to all matrices whose underlying graphs have "good separators", where a "good" separator is of size $O(\sqrt{n})$. Several important classes of graphs have such separators, in particular, the class of all planar and nearly planar graphs [38], and α-overlap graphs [41]. Our space saving technique used in the COMPACT PND algorithm can be applied to develop a generalized algorithm when the underlying graph of the matrix is of bounded degree and has an $O(\sqrt{n})$ separator, as is true in many two dimensional PDE problems. We will call this generalized algorithm GENERALIZED COMPACT PND.

Theorem 6.9 *Let A be a symmetric positive definite matrix whose underlying graph G is of bounded degree and has an $O(\sqrt{n})$ size separator. COMPACT PND can be implemented on a grid- or mesh-connected machine in a way that maintains the current known asymptotic time bounds for the algorithm, and moreover, requires only a constant factor of extra storage.*

Proof: In order to be able to implement the COMPACT PND algorithm in the given time and space bounds, we must show that the size of the graphs generated at any step of the algorithm is bounded, and that we can balance the load among the processors. Consider the matrix

$$A_h = \begin{bmatrix} X_h & Y_h^T \\ Y_h & Z_h \end{bmatrix}$$

described in Section 6.2. By definition, X_h is a block diagonal matrix representing the vertices of the graph which are to be eliminated, Z_h represents the part of the

matrix which is yet to be factored, and Y_h provides the connection between the two. Any unchecked growth in the size of the problem would occur only in Y_h, as X_h has bounded block size and Z_h is as yet unaffected by the algorithm. However, since the initial graph was of bounded degree, we can bound the size of the neighborhood of each eliminated vertex, and we can also bound the number of edges between this neighborhood and the rest of the graph. Consequently, we can bound the degree of each neighbor of an eliminated vertex, and thus the total size of the matrix remains bounded. □

Note that this result does not generalize to all graphs to which nested dissection can be applied. If the input graph is not of constant degree or the algorithm is running in polylog time on a PRAM, adding the space saving modification will cause a slowdown of the algorithm by a factor of $\log n$. It would be of interest to see if this GENERALIZED COMPACT PND algorithm could be modified to be space efficient without compromising time bounds.

6.7 Extension to Path Problems

Pan and Reif [58] extended their PND algorithm to apply to path algebra computations, which have a wide range of applications. One particular application of importance is the problem of finding the *all-pairs minimum cost path* in a graph. They extend the solution to apply to problems over a semi-ring $(R, \oplus, \otimes)$. The operations of a semi-ring are not necessarily invertible. Thus the inverse operation (A^{-1}) is replaced with the transitive closure operation (A^*). We will again use $P(n)$ to denote the number of processors necessary to multiply two $n \times n$ matrices over this semi-ring in $O(\log n)$ parallel steps on the PRAM model. The transitive closure operation (A^*) on an $n \times n$ dense matrix A costs $P(n)$ processors and $O(\log^2 n)$.

Their algorithm solves all-pairs-shortest-path problems within the same complexity as PND for a sparse input matrix A with small separators. For example, it finds all-pairs-shortest-paths in the mesh network in $O(\sqrt{n})$ time using a linear number of processors.

The modification of our COMPACT PND to a path problems algorithm is similar to the modified PND algorithm described in Section 6.5. First, we factor the input matrix A into:

$$A = L_0 L_1 \ldots L_d D L_d^T \ldots L_1^T L_0^T$$

where $d = \log n$ and the L_i are appropriately defined with the new operations. We then wish to solve

$$(L_0^T)^*(L_1^T)^* \ldots (L_d^T)^* D^T L_d^* \ldots L_1^* L_0^* b = x$$

which, if performed from right to left, is a sequence of vector-matrix operations. Half of these can be performed during the factorization stage, and the factorization needs to be recalculated if each L_i is to be stored in memory over L_{i-1} in order to conserve storage.

The analysis of time and storage bounds is essentially identical to that given for COMPACT PND in Section 6.5. In particular, for each scalar operation $\oplus, \otimes$ over the semi-ring $(R, \oplus, \otimes)$ executed by the COMPACT PND path problem algorithm, there is a corresponding scalar operation $+, *$ respectively executed by the COMPACT PND in Section 6.5. Also, for each $n' \times n'$ matrix operation $\oplus, \otimes$, and transitive closure executed by the COMPACT PND path problem algorithm, there is a corresponding $n' \times n'$ matrix operation $+, *$, and matrix inverse respectively executed by the COMPACT PND in Section 6.5 within a constant factor of the same parallel cost in terms of time, space and processors. Hence, the cost of solving path problems is within a constant factor of the the time, space and processor bounds of COMPACT PND for input matrices with the same class of graphs.

6.8 Open Problems

The results in this chapter describe three separate improvements to the known parallel nested dissection algorithms. Unfortunately, these appear at this time to be mutually exclusive. It would be interesting to see if any of the techniques described can be

combined to yield PND algorithms that exhibit improvements in both time and space bounds, or that apply these savings to wider classes of matrices and graphs.

Of further interest is the wider applicability of our approach of combining tree levels into multi-levels to reduce the computation by a log factor. This may be applicable to other parallel algorithms that decompose their input using some form of separator tree.

Chapter 7

Application to Convex Hull

A natural source for algorithms to which we could apply our technique is the field of computational geometry, where many problems are solved with algorithms that use random sampling to partition the data [47]. We decided to consider the problem of computing the convex hull of a set of points in the plane. This is a fundamental problem with many applications, that has been thoroughly studied [63], both statically and dynamically. On the face of it, it would seem that the simplicity of our algorithm may have something to offer over more complex solutions, and may even be able to give competitive time bounds for computing the planar convex hull of a point set. However, we will see in this chapter that the dynamic convex hull algorithm created by our technique had unacceptably long update times, due to the fact that updated subproblems take too long to merge into a solution. This is consistent with the theoretical results predicted in Section 2.4.

7.1 Convex Hull

The *convex hull* of a set of points P in the plane is the smallest convex set containing P. It is well known (and easy to prove by a reduction to sorting) that computing the convex hull of a point set requires $\Omega(n \log n)$ time. It is also known [63] that dynamic updates to the convex hull of a set of points are lower bounded by $\Omega(\log n)$. There are algorithms that attain this bound for insertions [62]. If we wish to have a fully dynamic data structure, supporting both insertions and deletions, the algorithm of Overmars and van Leeuwen [51] gives a data structure that can perform updates in $O(\log^2 n)$ time.

The reason deletions are more difficult to perform is that points inside the hull may come into play if a point on the hull is deleted. If we wish to avoid extensive

computation when that happens, we need to store all the points of the data set in a smart way. Intuitively, this means storing all the points in hull layers. If a point of the hull is deleted, the algorithm can fall back on inner layers to create the new hull.

Overmars and van Leeuwen [51] stored the point set in a data structure that enables updates to the point set in $O(\log^2 n)$ time. Their data structure follows the leaf model. The points are stored in the leaves of a balanced binary tree. Each internal nodes holds the hull of the points stored in the leaves of the subtree rooted at that node. The tree is balanced and has a height of $\log n$. Updating the tree requires locating the leaf where the point needs to be stored (or deleted from) by following a path down the tree, then following the same path back to the root, updating the hulls in the internal nodes on the way. If necessary, some tree balancing takes place as well. The work at each node on the path may take as long as $O(\log k)$ where k is the number of points in the subtree rooted at that node. Thus the total time to perform an update is $O(\log^2 n)$.

A simple application of our technique to the data structure of Overmars and van Leeuwen would not yield a better time bound nor a simpler algorithm. We needed to find a different data structure, in which to store the set of points in a way that would enable computing the hull easily. Moreover, we wanted an algorithm that would facilitate reconstructing part of the hull if only part of the data structure changed. Consider this high level description of the algorithm:

Input: A set P of n points in the plane.

Output: The convex hull of P.

Algorithm:

1. Sort the input points by their x-coordinate.

2. Divide the points into an upper and a lower hull. Without loss of generality, the rest of the algorithm deals with the construction of the upper hull. The lower hull is constructed similarly.

3. Let n be the number of points in the input. Randomly select a sample S

containing $\sigma(n) = n^\epsilon$ points of the input. These points split the plane into $\sigma(n)$ sectors; the borders of the sectors are the rays emanating from a central point and going through each of the points in the sample. The *size* of each sector is the number of points within the sector. The expected size of each sector is $n^{1-\epsilon}$.

4. Recursively compute the convex hull of each sector.

5. Combine the sector hulls into a single hull.

The crux of the algorithm is the merge step (5). As discussed in section 2.4, the time needed by the merge step may dominate the time bounds for the entire algorithm. First, let us focus on how the points are stored in the data structure, which we call the partition tree. Like the data structure of Overmars and van Leeuwen, the partition tree follows the leaf model. The data points are stored in the leaves of the tree, and the internal nodes contain information about the partition to sectors and about the hull for each sector.

7.1.1 The Partition Tree

We define a *partition tree* of the point set as follows:

The input points are stored in the leaves of a tree. The internal nodes of the tree define the recursive partitioning of the sectors. Each internal node at level i corresponds to a sector at the ith recursion level, and is the root of a subtree that contains the points included in that sector. The *size* of a subtree is the number of points stored in its leaves. The next two simple lemmas are given without proof.

Lemma 7.1 *The expected size of a subtree rooted at level i of the partition tree is $n_i = n^{(1-\epsilon)^i}$.*

Partition the points in each sector. At level i of the recursion, a sector contains n_i points and is partitioned into σ_i sub-sectors by a random sample S_i of size $\sigma_i = n_i^\epsilon$.

The root of the tree is considered to be level 0. The root has $\sigma_0 = n_0^\epsilon$ children, each corresponding to a sector. A subtree rooted at level i has $\sigma_i = n_i^\epsilon$ children, each corresponding to a sub-sector.

Lemma 7.2 *The expected number of children of a node at level i of the partition tree is $\sigma_i = n^{\epsilon(1-\epsilon)^i}$.*

Define L_i to be the number of internal nodes at level i of the tree, that is, the number of sub-sectors at the ith level of recursion.

Lemma 7.3 *The expected size of L_i is $n^{1-(1-\epsilon)^i}$.*

Proof: The number of internal nodes at a given level of the partition tree is the number of nodes at one higher level times their expected number of children, giving the recursion

$$L_i = L_{i-1}\sigma_{i-1} = L_{i-1}n^{\epsilon(1-\epsilon)^{i-1}}$$

with $L_0 = 1$ (the root). Solving this recurrence, we get

$$L_i = \prod_{j=0}^{i-1} n^{\epsilon(1-\epsilon)^j} = n^{f(\epsilon,i)} = n^{1-(1-\epsilon)^i}$$

where

$$f(\epsilon, i) = \sum_{j=0}^{i-1} \epsilon(1-\epsilon)^j = \epsilon \cdot \frac{1-(1-\epsilon)^i}{1-(1-\epsilon)} = 1-(1-\epsilon)^i$$

□

Alternate proof: The leaves of the tree contain all $n_0 = n$ input points. From Lemma 7.1, the expected size of a subtree is $n^{(1-\epsilon)^i}$. Since the subtrees define a partition, they are disjoint. Thus the expected number of subtrees is $n/n^{1-\epsilon^i} = n^{1-(1-\epsilon)^i}$. □

The recursion stops when the sector size is less than some constant c.

Lemma 7.4 *The expected depth of the partition tree is $O(\log\log n)$.*

Proof: The size of a sector at level i of the partition is $n^{(1-\epsilon)^i}$ (Lemma 7.1). To find the level when this size is polylog, solve $n^{(1-\epsilon)^k} = c$ for k. □

Partitioning a sector into sub-sectors has two components: selecting the random sample, which can be done in constant time per sample point, and determining which sub-sector each point belongs to, which can be done in constant time per input point.

Lemma 7.5 *The sectors at level i of the recursion can be partitioned in $O(n)$ time.*

Proof: By lemma 7.3, there are $n^{1-(1-\epsilon)^i}$ sectors in the ith level of the recursion. By Lemma 7.2, the sample selected in each of these sectors is of size $n^{\epsilon(1-\epsilon)^i}$. Thus the time to select all the samples in this level is

$$n^{1-(1-\epsilon)^i} \times n^{\epsilon(1-\epsilon)^i} = n^{1-(1-\epsilon)^{i+1}}.$$

At level i, there are $n^{1-(1-\epsilon)^i}$ subtrees. In each one, we select $n^{\epsilon(1-\epsilon)^i}$ sample points. The total time to select the sample points is

$$n^{1-(1-\epsilon)^i} n^{\epsilon(1-\epsilon)^i} = n^{1-(1-\epsilon)^{i+1}}$$

Sample points have to be placed in two sectors touching on their ray. All other points are in exactly one sector (if on the ray they can be assigned arbitrarily). The total time to distribute the points is thus

$$2n^{1-(1-\epsilon)^{i+1}} + (n - n^{1-(1-\epsilon)^{i+1}}) = n + n^{1-(1-\epsilon)^{i+1}} = O(n).$$

□

Combining all these results, we have:

Lemma 7.6 *The total time required to recursively partition the input point set is $O(n \log\log n)$.*

Proof: By Lemma 7.4, the depth of the partition tree is $O(\log\log n)$. By Lemma 7.5, the total work at each level is $O(n)$. The total work in all levels is therefore $O(n \log\log n)$. □

7.1.2 Merging of Sector Hulls

Once the partition tree is built, we can use it to construct a convex hull. The way to do this, is by merging the hulls of sibling subtrees. The hulls of the sectors at the leaves of the partition tree are built using any standard static convex hull algorithm. Since the number of points in each leaf is bound by a constant, the total work to construct each of the leaf hulls is also constant. The hull of a level i sector is built from the (already constructed) hulls of the sectors in level $i+1$, by merging pairs of adjacent hulls. Adjacent pairs are combined using the upper tangent algorithm of Overmars and van Leeuwen [51], which is linear in the number of points whose hulls are to be merged. This reduces the number of hulls by half. Continue merging adjacent pairs until there is only one hull.

Lemma 7.7 *The time required to construct one of the level i hulls is* $\epsilon(1-\epsilon)^i n \log n$.

Proof: Look first at how long it takes to merge the hulls on one level. In order to merge all the level $i+1$ hulls into one level i hull, we need to merge $n^{\epsilon(1-\epsilon)^i}$ subtrees, in pairs. Each of these subtrees contains $n^{(1-\epsilon)^{i+1}}$ points, and each merge take time linear in the number of points merged. We have a tree of merges, which we call the *merge tree*, which has $\log n^{\epsilon(1-\epsilon)^i}$ levels, and in each level the amount of work done is $n^{(1-\epsilon)^i}$. The work in each level of the merge tree is the same, because there are half as many merges involving twice as many points each time we go up a level in the merge tree. The total work for the merge tree is

$$n^{(1-\epsilon)^i} \cdot \log n^{\epsilon(1-\epsilon)^i} = n^{(1-\epsilon)^i} \cdot \epsilon \cdot (1-\epsilon)^i \cdot \log n$$

This is just the work involved in merging *one* of the hulls in level i. By Lemma 7.3 there are a total of $n^{1-(1-\epsilon)^i}$ hulls in level i, so the total time to merge *all* the hulls in level i is

$$n^{1-(1-\epsilon)^i} \cdot n^{(1-\epsilon)^i} \cdot \epsilon \cdot (1-\epsilon)^i \cdot \log n = \epsilon(1-\epsilon) n \log n.$$

□

Construction of the partition tree was a top-down process. Hulls are created at the leaves, and built bottom-up along the partition tree, with a merge step at every level. This leads to

Lemma 7.8 *The time required to merge the hulls at the leaves into a single hull is* $O(n \log n)$.

Proof: By lemma 7.7, the time to create all the hulls in level i is $\epsilon(1-\epsilon)n \log n$. The time to create the hull of the entire point set is the time it takes to create the hulls at the leaves plus the time to merge all the levels up to the top. There are $O(\log \log n)$ levels in the partition tree, so the total time to construct the hull from the partition tree is

$$\sum_{i=0}^{O(\log \log n)} \epsilon(1-\epsilon)^i n \log n = \epsilon n \log n \frac{1-(1-\epsilon)^{\log \log n}}{\epsilon} = O(n \log n).$$

□

7.1.3 Complexity analysis

From the results of the previous section, we have:

Lemma 7.9 *The static algorithm computes a convex hull for a set of points in the plane in time* $O(n \log n)$.

In other words, the static algorithm has the best attainable time bound for static construction the convex hull. The next section describes the use of the partition tree in constructing a dynamic convex hull algorithm.

7.2 Dynamic Maintenance

Our goal is to maintain the convex hull of a dynamically changing set P of points in the plane, processing a stream of update and query requests. Update requests ask to

INSERT or DELETE a point from P. QUERY requests ask where a point is located with respect to the hull.

Our algorithm proceeds inductively. Given a partition tree, the entire structure or a part thereof may be rebuilt with each new insertion or deletion. Rebuilding is done by calling the static algorithm for all or part of the point set.

The size of the sample used to create the partition of a level i sector is $n^{\epsilon(1-\epsilon)^i}$. At the top level of the partition tree, this is n^ϵ. Thus we build the entire tree with probability $\frac{n^\epsilon}{n}$. Otherwise, we determine which sector contains the update point. This can be done in $\epsilon \log n$ time using a binary search on the sectors. The sector found is a level 1 sector, and it is rebuilt with probability $\frac{n^{\epsilon(1-\epsilon)}}{n}$. If it is not rebuilt, determine the level 2 sub-sector that the update point belongs in, etc. This process continues, either rebuilding or descending until rebuilding happens or we reach a leaf.

Rebuilding includes not only restructuring the partition, but also computing the associated hull. Once this is done, the algorithm needs to use the information from the rebuilt subtree to readjust the hulls in all levels above where the rebuilding happened, in order to maintain the correct hull for the entire point set.

7.2.1 Time to Perform an Update

In determining the total cost of an update, we use the following notation. Let:

P_i be the probability of updating the partition at level i.

C_i be the cost of updating the partition and hull at level i.

F_i be the cost of finding which sector a point belongs to.

We get the following recursion for the total cost of an update:

$$T_D(n_i) \leq P_i C_i + (1 - P_i)(F_i + T_D(n_{i+1})).$$

In this algorithm, P_i is the probability of being included in the sample at level i, which is, by lemma 7.2, $\frac{n^{\epsilon(1-\epsilon)^i}}{n^{(1-\epsilon)^i}} = n^{-(1-\epsilon)^{i+1}}$.

The cost of finding the sector that a point belongs to is the cost of performing binary search through the sectors. Note that there is no need to search through all sub-sectors at a level, but only through the sub-sectors of the sector that the point belongs in. The cost of finding the sub-sector that a point belongs to at level i is $\log n_i^\epsilon = \epsilon(1-\epsilon)^i \log n$.

The cost of updating the partition at level i has two components: updating the sector hull and merging the sector hull up the partition tree. When performing the merge step, we have to go through a merge tree at every level of the partition tree. We do not need to construct the entire merge tree, but we do need to merge a path up each of the merge trees.

Lemma 7.10 *The time to update a level i merge tree is $\epsilon(1-\epsilon)^i n^{(1-\epsilon)^{i+1}}$.*

Proof: At each level of the merge tree we merge a pair of sub-sectors. As we proceed up the merge tree these sub-sectors become larger, doubling in size with each level. The size of a sub-sector at a leaf of the merge tree is $n^{(1-\epsilon)^{i+1}}$. The total work to merge a path up the level i merge tree is

$$n^{(1-\epsilon)^{i+1}} \sum 2^i = \epsilon(1-\epsilon)^i n^{(1-\epsilon)^{i+1}}.$$

□

But this is the work for merging only one level of the partition tree. To update the entire hull, all levels of the partition tree must be updated. At each level, there is a path merge on a merge tree. The total time to update the entire partition tree is

$$\sum_{j=1}^{i} \epsilon(1-\epsilon)^j n^{(1-\epsilon)^{j+1}}.$$

The first term in this summation is already polynomial in n. This leads to the following result:

Theorem 7.11 *The time to perform an update on a convex hull of a set of points P using the partition tree data structure is $\Omega(n^{(1-\epsilon)})$.*

Chapter 8

Conclusion

8.1 Summary

We have presented a general technique for converting static randomized algorithms that use sampling into dynamic algorithms. The key component of our technique is a dynamic data structure that is used as the basis for the dynamic algorithm. This data structure may be completely or partially rebuilt during the execution of the algorithm. However, since during many update operations no rebuilding takes place, the expected time to perform an update is significantly less than the time it takes to rebuild the entire structure.

We showed that our technique is useful in creating dynamic algorithms with polylogarithmic expected time bounds for several classes of data structures and algorithms.

In order to guarantee that the expected time bounds are attained with high likelihood, we introduced the concept of replicants, parallel processes that run the dynamic algorithm, each maintaining its own data structure. With high likelihood, at least one of the replicants attains the desired time bound.

8.2 Lessons Learned

Given any general technique, it is important to recognize when it is useful and when not. In this thesis we examined two applications of our technique, with widely differing results. Applying our dynamizing technique to the sphere separator algorithm resulted in an algorithm with good time bounds on performing an update. On the other hand, the randomized convex hull algorithm cannot be dynamized by our technique to yield a dynamic algorithm with better than linear bounds on performing an update.

The question arises, is there an inherent difference between these two problems that causes such a gap between the behaviors of their respective dynamic algorithms? In fact, there is: one problem has a single correct solution, while the other draws a solution from a set of possible solutions, all of which conform to certain parameters. For a given set of points, there is only one convex hull, but there may be a number of good separators.

8.3 Directions for Further Research

The fundamental difference between the two cases we investigated suggests a further question: can we develop a framework for defining the classes of algorithms that can benefit from our technique? We have already done so based on characteristics such as running time and sample size. Could we also decide if a static algorithm lends itself to our technique based on some other characteristics of the algorithm or of the problem it sets out to solve?

We suspected that the convex hull problem would not have good dynamic time bounds because its merge step was linear. Is this true of all other algorithms that have unique solutions?

The separator algorithm has less exacting requirements of its solution. Would other algorithms that impose parameters on their solutions in a way that makes several solutions possible be dynamizable with good time bounds?

Another class of algorithms that we have not yet looked at but which appears promising, is the class of approximation algorithms. These algorithms have the same property that a correct solution is drawn from a set of possible solutions. To see if this prediction holds true, we would start by applying our technique to an instance of this class of algorithms. We can then establish parameters for determining which approximation algorithms can be the basis for efficient dynamic approximation algorithms.

Appendix A

Code for the Dynamic Data Structure

This appendix contains the basic code for the dynamic data structure, applied to a binary search tree of integers. The programs were written in C++ and structured in independent modules that can call each other. Included here are only the modules for the basic operations – the static algorithm, updates and searches. Utility modules have been omitted.

For each of the experiments described in Chapter 4, we wrote a short programs that called the modules presented below. We then wrote Perl scripts that ran the programs many times on different data and generated statistics about the runs.

Since the code was written in this modular fashion, it should be relatively easy to adapt it to other data structures and algorithms. The static algorithm would, of course, be different; the sampling may change; and, if the algorithm involves computing some information from the data structure, the code for this would have to be added.

Each of the Sections A.2 through A.4 contain code for both the node and the leaf model. Detailed descriptions of each model can be found in Chapter 4. Section A.5 contains the code for simulating a single replicant accepting and processing a stream of requests. This program is completely independent of the data structure and should not change across applications.

A.1 Basic Definitions

This is a header file used in all the programs. It contains some constants, function declarations and the definition of the basic building block of the data structure – in this case, a binary search tree.

```
#ifndef _PROJECT_H_
```

```
#define _PROJECT_H_

#include <stdlib.h>
#include <stdio.h>                          // old habits die hard...
#include <iostream.h>

struct Node
{
   Node(){right = NULL; left = NULL;}
   ~Node() {delete right; delete left;}
   int  count;
   int number;                              //data field
   Node * right;                            //pointer
   Node * left;                             //pointer
};

Node * Build_it  (int *, int);              //prototypes
Node * Insert(Node *, int);
Node * Delete(Node *, int, int &);
int Search(Node *, int);

void load(Node *, int*, int *);             //utilities
void unique(int *, int*, int, int);
int median(int *, int, int, int);

//Some useful macros
#define random(x)       ((x) < 1 ? 0 : (rand()) % (x))
#define avg3(x,y,z)     (((x)+(y)+(z))/3)
#define greatest(x,y)   ((x) > (y) ? (x) : (y))
extern const int max;

extern int depth;                   // Depth of the tree at some point

extern unsigned int comp;           // comparison count

#define SEARCH 0
```

```
#define DELETE 1
#define INSERT 2

#define TRUE  1
#define FALSE 0
#endif _PROJECT_H_
```

A.2 Static Algorithm

This is the heart of the program, the static algorithm which may be called as a subroutine from the update modules. It creates a binary search tree according to the node or leaf model (for a detailed explanation of each see Chapter 4). It could be changed to any static randomized algorithm that uses sampling.

Node Model

```
#include "project.h"

// Creates a binary search tree from the elements of array.
// A random element is selected to be the pivot and placed
// at the root.  All elements smaller/greater than the pivot
// recursively make up the left/right subtree.

Node * Build_it(int * array, int end)
{
   comp++;
   if(0 <= end) {                          // array exists
      comp++;
      if (0 == end) {                      // array one cell
         Node *newone = new Node;
         newone->number = array[0];
         newone->count = 1;
         return newone;
      }
```

```
else {
   int  ran, piv, len;
   len = end +1;                          // cells in sub array

   if (len >= 3) {                        // median of three
      comp += 3;
      int r1, r2, r3;
      r1 = (rand() % len);
      r2 = (rand() % len);
      r3 = (rand() % len);
      ran = median(array, r1, r2, r3);
   }
   else {
      comp++;
      ran = (rand() % len);               // random index
   }
   piv = array[ran];                      // get pivot
   array[ran] = array[0];                 // move to 1st cell
   array[0] = piv;

   Node *newone = new Node;
   newone->number = array[0];
   newone->count = end + 1;

   int w=0;                               // space needed
   for (int i = 1; i < len; i++)
       if (array[i] < piv)
           w++;

   int *less = new int[w];
   int *more = new int[len - w];
   int l = -1;
   int m = -1;
   int i = 1;
```

```
            while (i <= end) {                    // look at all cells
               comp++;
               if (array[i] < piv) {
                  l++;
                  less[l]=array[i];
               }
               else {
                  m++;
                  more[m]=array[i];
               }
               i++;
            }
            newone->left = Build_it(less, l);
            delete less;
            newone->right = Build_it(more, m);
            delete more;
            return newone;
         }
      }
      return NULL;
}
```

Leaf Model

```
#include "project.h"

// Creates a binary search tree from the elements of array.
// The mean of three random element is selected to be the pivot.
// All elements smaller/greater than the pivot recursively make up
// the left/right subtree.

Node * Build_it(int * array, int end)
{
```

```
int  piv, len;                          // temp vars
int w=0;                                // space needed

comp++;
if (end >= 0) {                         // array exists
   comp++;
   if (end == 0) {                      // array one cell
      Node *newone = new Node;
      newone->number = array[0];
      newone->count = 1;
      return newone;
   }
   else {
      len = end +1;                     // cells in sub array

      if (len >= 3) {                   // mean of three
         comp += 3;
         piv = avg3(array[random(len)], array[random(len)],
                    array[random(len)]);
      }
      else {
         comp++;
          piv = greatest(array[0],array[1]);
      }
      Node *newone = new Node;
      newone->number = piv;
      newone->count =  len*2;

      for (int i = 0; i < len; i++) {
         if (array[i] < piv)
            w++;
      }
      int *less = new int[w];
      int *more = new int[len - w];
```

```
            int l = -1;
            int m = -1;
            int i = 0;

            while (i <= end) {                    // look at all cells
               comp ++;
               if (array[i] < piv) {
                  l++;
                  less[l]=array[i];
               }
               else {
                  m++;
                  more[m]=array[i];
               }
               i++;
            }
            newone->left = Build_it(less, l);
            delete less;
            newone->right = Build_it(more, m);
            delete more;
            return newone;
         }
      }
      return NULL;
}
```

A.3 Updates

This section contains the code for performing updates to the data structure. Section A.3.1 gives the code for inserting a data point into the tree. Section A.3.2 gives the code for removing a data point from the tree.

A.3.1 Insert

Node Model

```
#include "project.h"

int depth;
unsigned int comp =0;

// Inserts element newnum into the tree rooted at root.

Node * Insert(Node * root, int newnum)
{
   depth++;
   int result;

   comp++;
   if (root == NULL) {
      Node *newone = new Node;
      newone->number = newnum;
      newone->count = 1;
      return newone;
   }
   else {
      comp++;
      result = random(root->count+1);

      comp++;
      if (result == 1) {
         comp++;
         if (root->count == 1) {
            Node *newone = new Node;
            newone->number = root->number;
            newone->count = 1;

            if (newnum > root->number)
               root->left=newone;
```

```
            else
               root->right=newone;

            root->number=newnum;
            root->count +=1;
            return root;
         }
         else {
            int i = 0;
            int size = 0;
            size = root-> count;
            int * array = new int[size +1];
            load(root, array, &i);
            array[size] = newnum;

            delete root;
            root = Build_it(array, size);          //rebuid
            delete array;
            return root;
         }
      }
      else {
         root->count +=1;

         comp++;
         if (newnum < root->number)
            root->left = Insert(root->left, newnum);
         else
            root->right= Insert(root->right, newnum);
      }
      return root;
   }
}
```

Leaf Model

```
#include "project.h"

int depth;
unsigned int comp;

// Inserts element newnum into the tree rooted at root.

Node * Insert(Node * root, int newnum)
{
   depth++;
   int result;

   comp++;
   if (root == NULL) {
      Node *newone = new Node;
      newone->number = newnum;
      newone->count = 1;
      return newone;
   }
   else {
      comp++;
      result = random(root->count+1);

      comp++;
      if (root->count == 1) {
         // create the new leaf
         Node *newone = new Node;
         newone->number = newnum;
         newone->count = 1;

         // move the current node down
         Node *othernode = new Node;
         othernode->number = root->number;
         othernode->count = 1;
```

```
    //and set root's children to these two
    root->number = greatest(root->number, newnum);

    root->left = (othernode->number < newnum ?
                              othernode : newone);
    root->right = (othernode->number < newnum ?
                              newone : othernode);
    root->count++;
    return root;
 }
 comp++;
 if (result == 1) {
    int i = 0;
    int size = 0;
    int * array = new int[max];
    load(root, array, &i);
    size = i;
    array[size] = newnum;

    delete root;
    root = Build_it(array, size);              //rebuid
    delete array;
    return root;
}
else {
    root->count +=1;

    comp++;
    if (newnum < root->number)
       root->left = Insert(root->left, newnum);
    else
       root->right= Insert(root->right, newnum);
}
return root;
```

```
   }
}
```

A.3.2 Delete

Node Model

```
#include "project.h"

// Deletes element newnum from the tree rooted at root.

Node * Delete(Node * root, int newnum)
{
   int result;
   int found = 0;
   depth++;

   comp++;
   if (root == NULL)
      return NULL;
   else {
      comp++;
      result = random(root->count-1);

      comp++;
      if (root->number == newnum) {
         result = 1;
         found = 1;
      }
      comp++;
      if (result == 1) {
         comp++;
         if (root->left == NULL && root->right == NULL) {
            comp++;
            if (newnum == root->number) {
               delete root;
```

```
        return NULL;
      }
      else
        return root;
   }
   else {
      int i = 0;
      int size = root-> count;
      int * array = new int[size];
      load(root, array, &i);

      int lfound =-1;
      for (int h=0; h<size; h++) {
         if (array[h] == newnum) {
            lfound = h;
            break;
         }
      }

      if (lfound != -1) {
         int temp = array[lfound];
         array[lfound] = array[size -1];
         array[size -1] = temp;
      }
      delete root;

      comp++;
      if (lfound == -1)                    // rebuid
         root = Build_it(array, (size -1));
      else
         root = Build_it(array, (size -2));
      delete array;
      return root;
}
```

```
      }
      else {
         comp++;
         if (newnum < root->number)
            root->left = Delete(root->left, newnum);
         else
            root->right= Delete(root->right, newnum);
      }
      if (found)
         root->count -= 1;
      return root;
   }
}
```

Leaf Model

```
#include "project.h"

const int max = 1100000;

// Deletes element newnum from the tree rooted at root.

Node * Delete(Node * root, int newnum)
{
   depth++;
   int result;

   comp++;
   if (root == NULL)              // Item not found, so don't bother
      return NULL;
   else {                         // Decide whether or not to rebuild
      comp++;
      result = random(root->count-1);
      comp++;
      if (root->number == newnum && root->count == 1)
        result = 1;
```

```
comp++;
if (result == 1) {
   comp++;
   if (root->count == 1) {
      comp++;
      if (newnum == root->number) {
         delete root;
         return NULL;
      }
      else
         return root;
   }
   else {
      int i = 0;
      int *array = new int[max];
      int size;

      load(root, array, &i);
      size = i;
      int lfound =-1;
      for (int h=0; h<size; h++) {
         comp++;
         if (array[h] == newnum) {
            lfound = h;
            break;
         }
      }
      if (lfound != -1) {
         int temp = array[lfound];
         array[lfound] = array[size -1];
         array[size -1] = temp;
      }
      delete root;
```

```
            comp++;
            if (lfound == -1)                          //rebuid
               root = Build_it(array, (size -1));
            else
               root = Build_it(array, (size -2));
            delete array;
            return root;
          }
      }
      else {
         comp++;
         if (newnum < root->number) {
            root->left = Delete(root->left,newnum);
            if (found)
               root->count -= 1;
            if (root->left == NULL && root->count != 1)
               return root->right;
         }
         else {
            root->right = Delete(root->right,newnum);
            if (found)
               root->count -= 1;
            if (root->right == NULL && root->count != 1)
               return root->left;
         }
      }
      return root;
   }
}
```

A.4 Search

This section contains the code for searching through the data structure.

Node Model

```
#include "project.h"

int Search(Node *root, int data)
{
   comp++;
   if (root == NULL)
      return FALSE;
   comp++;
   if (root->number == data)
      return TRUE;
   comp++;
   if (root->number < data)
      return Search(root->right,data);
   else
      return Search(root->left,data);
   return FALSE;
}
```

Leaf Model

```
#include "project.h"

int search(Node *root, int data)
{
   comp++;
   if (root == NULL)
      return FALSE;
   comp++;
   if (root->number == data && root->count == 1) {
      comp++;
      return TRUE;
   }
   comp++;
```

```
    if (root->number < data)
       return Search(root->right,data);
    else
       return Search(root->left,data);
    return FALSE;
}
```

A.5 Replicants

This section contains the code for simulating a single replicant accepting and processing a stream of requests. We used the same code for both models. With the exception of the first part of the code where the input is read, this program should be useful for any data structure and application.

The program reads in data points and builds an initial tree. It then reads in a stream of requests – search, insert or delete – and performs them. The output is a sequence of cumulative operation counts after each request is satisfied.

```
#include <iostream.h>
#include <time.h>
#include <sys/times.h>
#include <stdlib.h>
#include <strstream.h>
#include "project.h"

const int MAX = 10000000;

int main(int argc, char *argv[])
{
   srand(time(NULL));
   int *array = new int[MAX];
   int command, data;
   int counter = 0;

   // Read in data from specified file
```

```
 if (argc != 2) {
    fprintf(stderr,"Usage: %s command_file\n",argv[0]);
    exit(1);
 }
 while (cin >> array[counter])
    counter ++;
 cin.clear();
 while (cin.eof())
    cin.ignore();

 int end = counter-1;
 Node * root;

 // Build the tree
 root = Build_it(array, end);                    //rebuid

 // Open the request stream file
 FILE *stream;
 stream = fopen(argv[1],"r");
 if (stream == NULL) {
    fprintf(stderr,"Error: Could not open %s!",argv[1]);
    exit(1);
 }

 // execute requests
 comp = 0;
 while (fscanf(stream,"%d %d",&command,&data) != EOF) {
    switch(command) {
      case SEARCH:
         search(root,data);
         break;
      case DELETE:
         root = Delete(root,data);
         break;
```

```
            case INSERT:
                root = Insert(root,data);
                break;
            }
            printf("%d ",comp);
        }
        printf("\n");
}
```

Bibliography

[1] G.M. Adel'son-Vel'skii and E.M. Landis. An information organizing algorithm. *Doklady Akad. Nauk USSR* 146:263–266, 1962.

[2] L.M. Adleman and K.L. Manders. Reducibility, randomness, and intractability. *Proc. 9th ACM Symposium on the Theory of Computing*, 151–163, 1977.

[3] A. Aho, J. Hopcroft and J. Ullman. *Data Structures and Algorithms.* Addison Wesley, 1983.

[4] N. Alon, P. Seymour and R. Thomas. A separator theorem for graphs with an excluded minor and its applications. *Proc. 22nd ACM Symposium on the Theory of Computing*, 293–299, 1990.

[5] D. Armon and J.H. Reif. Space and Time Efficient Implementations of Parallel Nested Dissection, *Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992.

[6] D. Armon and J.H. Reif. A Dynamic Separator Algorithm *Proc. 3rd Annual Workshop on Data Structures (WADS)*, 107–118, 1993.

[7] J.L. Bentley and J.B. Saxe. Decomposable searching problems I: static-to-dynamic transformation. *J. Algorithms*, 1(4):301–358, 1980.

[8] G. Birkhoff and J.A. George. Elimination by nested dissection. In *Complexity of Sequential and Parallel Numerical Algorithms* Traub JF, ed. Academic Press, New York, 1973.

[9] Y. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proc. IEEE* 80:1412–1434, 1992.

[10] K.L. Clarkson. New applications of random sampling to computational geometry. *Discrete and Computational Geometry* 2:195–222, 1987.

[11] K.L.Clarkson, K. Mehlhorn and R. Seidel. Four results on randomized incremental constructions. *proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science*, 463–474, 1992.

[12] K.L. Clarkson and P.W. Shor. Applications of random sampling in computational geometry II. *Discrete and Computational Geometry* 4:387–421, 1989.

[13] E. Cohen. Efficient parallel shortest-paths in digraphs with separator decomposition. *J. Algorithms* 21:331–357, 1996.

[14] R. Cole and M.T. Goodrich. Optimal parallel algorithms for polygon and point-set problems. *Tech Report 88–14*, Department of Computer Science, Johns Hopkins University, 1988.

[15] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions, em J Symbolic Comput 9(3):251–280, 1990.

[16] T.H. Cormen, C.E. Leiserson and R.L. Rivest Introduction to Algorithms McGraw-Hill 1990.

[17] J. Culberson and J.I. Munro. Explaining the behavior of binary search trees under prolonged updates: a model and simulations. *Computer Journal* 32:68–75, 1989.

[18] J. Culberson and J.I. Munro. Analysis of the standard deletion algorithms in exact fit domain binary search trees. *Algorithmica* 5:295–311, 1990.

[19] S.C. Eisenstadt, M.H. Scultz and A.H. Sherman. Applications of an element model for Gaussian elimination. In *Sparse Matrix Computations* J.R. Bunch and D.J. Rose, eds., Academic Press, New York, 1976.

[20] D. Eppstein, G.L. Miller and S.-H. Teng. A deterministic linear time algorithm for geometric separators and its application. *Fundamenta Informaticae* 22:309–329, 1995.

[21] G. Frederickson. Planar graph decomposition and all pair shortest paths. *JACM* 38(1):162–204, 1991.

[22] A. Frieze, G.L. Miller and S.-H. Teng. Separator based parallel divide-and-conquer in computational geometry. *Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, 420–430, 1992.

[23] H. Gazit and G.L. Miller A parallel algorithm for finding a separator in planar graphs. *Proc. 28th Annual Symposium on Foundations of Computer Science*, 238-248, 1987.

[24] H. Gazit and G.L. Miller. *Communication to J.H. Reif*, 1992.

[25] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal of Numerical Analysis* 10:345–363, 1973.

[26] A. George, M. Heath and J. Liu. Parallel Choleski factorization on a shared memory multiprocessor. *Lin Alg Appl* 77:165–187, 1986.

[27] J.R. Gilbert, J.P. Hutchinson and R.E. Tarjan. A separation theorem for graphs of bounded genus. *Journal of Algorithms* 5:391–407, 1984.

[28] J. Gilbert and R. Schreiber. Highly parallel sparse Choleski factorization. *SIAM Symposium on Sparse Matrices*, 96–98, 1989.

[29] L.J. Guibas, D.E. Knuth and M.Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica* 7:381–413, 1992.

[30] Y. Han, V.Y. Pan and J.H. Reif. Efficient parallel algorithms for computing all pair shortest paths in directed graphs. *Algorithmica* 17:399–415, 1997.

[31] M.T. Heath, E. Ng and B.W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review* 33:420–460, 1991.

[32] W.L. Hightower, J.F. Prins and J.H. Reif. Implementations of randomized sorting on large parallel machines. *Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, 158–167, 1992.

[33] C.A.R. Hoare. Quicksort *The Computer Journal* 5:10–15, 1962.

[34] E. Horowitz and S. Sahni. *Fundamentals of Data Structures* Computer Science Press.

[35] J. van Leeuwen and D. Wood. Dynamization of decomposable searching problems. *Information Processing Letters*, 10:51–56, 1980.

[36] C.E. Leiserson, J.P. Mesirov, L. Nekludova, S.M. Omohundro, J.H. Reif and W. Taylor. Solving sparse systems of linear equations on the connection machine. *Annual SIAM Conference*, A51, Boston, MA, July 1986.

[37] R.J. Lipton, D.J. Rose and R.E. Tarjan. Generalized nested dissection. *SIAM Journal of Numerical Analysis* 16:346–358,1979.

[38] R.J. Lipton and R.E. Tarjan. A separator theorem for planar graphs. *SIAM Journal of Applied Math.* 177–189, 1979.

[39] K. Mehlhorn. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry* Springer-Verlag, 1984

[40] K. Mehlhorn and M.H. Overmars. Optimal dynamization of decomposable searching problems. *Information Processing Letters*, 12:93–98, 1981.

[41] G.L. Miller, S.-H. Teng, W. Thurston and S.A. Vavasis. Automatic mesh partitioning. in *Sparse Matrix Computations: Graph Theory Issues and Algorithms* A. George, J. Gilbert and J. Liu, eds., IMA Volumes in Mathematics and it Applications 56;57–84, Springer-Verlag, 1993.

[42] G.L. Miller, S.-H. Teng, W. Thurston and S.A. Vavasis. Separators for sphere-packings and nearest neighbor graphs. *JACM*, 44(1):1–29, 1997.

[43] G.L. Miller, S.-H. Teng and S.A. Vavasis. A unified geometric approach to graph separators. *Proc. 32nd Annual Symposium on Foundations of Computer Science*, 538–547, 1991.

[44] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[45] K. Mulmuley Randomized multidimensional search trees: lazy balancing and dynamic shuffling. *Proc. 32nd Annual Symposium on Foundations of Computer Science*, 180–196, 1991.

[46] K. Mulmuley Randomized multidimensional search trees: further results in dynamic sampling. *Proc. 32nd Annual Symposium on Foundations of Computer Science*, 216–227, 1991.

[47] K. Mulmuley *Computational Geometry, An Introduction Through Randomized Algorithms.* prentice Hall, 1994.

[48] K. Mulmuley and O. Schwartzkopf. Randomized algorithms. in *Handbook of Discrete and Computational Geometry* J.E. Goodman and J. O'Rourke, eds., CRC Press, 1997.

[49] T. Opsahl and J.H. Reif. Solving sparse systems of linear equations on the massive parallel machine. In *First Symposium on Frontiers of Scientific Computing, NASA, Goddard Space Flight Center, Greenbelt, MD*, 2241–2248, September 1986.

[50] M. Overmars. The design of dynamic data structures. *Lecture Notes in Computer Science*, 156, Springer-Verlag, 1983.

[51] M.H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and Systems Sciences*, 23:166–204, 1981.

[52] M.H. Overmars and J. van Leeuwen. Two general methods for dynamizing decomposable searching problems. *Computing*, 26:155–166, 1981.

[53] M.H. Overmars and J. van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Information Processing Letters*, 12(4):168–173, 1981.

[54] V. Pan. Fast and efficient parallel algorithms for the exact inversion of integer matrices. *Proc. 5th Conference FST and TCS*, Lecture Notes in Computer Science, 504–521, 1985.

[55] V. Pan. Complexity of parallel matrix computations. *Theoretical Computer Science* 54, 65–85, 1987.

[56] V. Pan and J.H. Reif. Fast and efficient parallel solution of linear systems. *Proc. 17th ACM Symposium on the Theory of Computing*, 143–152, 1985.

[57] V. Pan and J.H. Reif. Fast and efficient algorithms for linear programming and for the linear least squares problem. *Proc. 12th International Symposium on Mathematical Programming*, MIT, Cambridge, MA, August 1985. also in *Computers and Mathematics with Applications*, 12A(12):1217–1227, 1986.

[58] V. Pan and J.H Reif. Fast and Efficient Solution of Path Algebra Problems. *Journal of Computer and Systems Sciences* 38(3):494–510, 1989.

[59] V. Pan and J.H. Reif. Acceleration of minimum cost path calculations in graphs having small separator families. Technical Report, 1989.

[60] V. Pan and J.H. Reif. On the bit complexity of discrete approximations to PDEs . *International Colloquium on Automata, Languages, and Programming*, 612–625, 1990.

[61] V. Pan and J.H. Reif. Fast and efficient parallel solution of sparse linear systems. *SIAM Journal on Computing*,22(6):1227–1250, 1993.

[62] F.P. Preparata. An optimal real time algorithm for planar convex hulls. Communications of the ACM 22:402–405, 1979.

[63] F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[64] M.O. Rabin Probabilistic algorithms. in *Algorithms and Complexity* J.F. Traub, ed., Academic Press, 21–36 1976.

[65] S. Rajasekaran and J.H. Reif. Randomized parallel computation. *Proc. Foundations of Computation Theory Conference* 364–376, 1987.

[66] J.H. Reif and S. Sen. Optimal parallel randomized algorithms for computational geometry II. *Tech Report CS–1990–7*, Department of Computer Science, Duke University, 1990.

[67] J.H. Reif and S. Sen. Optimal parallel randomized algorithms for 3-D convex hulls and related problems. *SIAM Journal on Computing*,21(3):466–485, 1992.

[68] J.H. Reif and S. Sen. Optimal randomized parallel algorithms for computational geometry. *Algorithmica*,7:91–117, 1992.

[69] D.J. Rose and G.F. Whitten. A Recursive analysis of dissection strategies. in *Sparse Matrix Computations*, Academic Press, New York, 1976.

[70] J.B. Saxe and J.L. Bentley. Transforming static data structures to dynamic structures. *Proc. 20th Annual IEEE Symposium on Foundations of Computer Science*, 148–168, 1979.

[71] R.G. Seidel and C.R. Aragon. Randomized search trees. *Algorithmica* 16:464–497, 1996.

[72] D. Sleator and R.E. Tarjan Self-adjusting binary search trees. *JACM* 32(3):652–686, 1985.

[73] D. Sleator and R.E. Tarjan Self-adjusting heaps. *SIAM Journal on Computing* 15(1):52–69, 1986.

[74] R. Solovay and V. Strassen, A fast Monte Carlo test for primality. *SIAM Journal on Computing*, 6(1):84–85, 1977. Amended in *SIAM Journal on Computing*, 7(1):118, 1978.

[75] V. Strassen, The asmptotic spectrum of tensors and the exponent of matrix multiplication, *Proc. 27th Annual IEEE Symposium on Foundations of Computer Science*, 49–54, 1986.

[76] S.-H. Teng. Points, Spheres and Separators: A Unified Geometric Approach to Graph Partitioning. *PhD thesis*, Carnegie-Mellon University, School of Computer Science, CMU-CS-91-184, 1991.

[77] P.M. Vaidya. An optimal algorithm for the all-nearest-neighbor problem. *Proc. 27th Annual Symposium on Foundations of Computer Science* 117–122, 1986.

[78] M.A. Weiss. *Data Structures and Algorithm Analysis* Benjamin Cummings, 1994.

[79] D. Wildenhain. Guide to parallel nested dissection on the MPP *Manuscript*, September, 1987.

[80] P.H. Worley and R. Schreiber. Nested dissection on a mesh-connected processor array. *Proc. ARO Workshop on New Computing Environments: Parallel, Vector and Systolic*, 1985.

Biography

Deganit Armon was born on November 6, 1962 in Jerusalem, Israel. She graduated from high school in Jerusalem, and served for 5 years in the Israeli military, attaining the rank of lieutenant. After her discharge, she came to the United States, and studied in Rochester Community College and Winona State University in Minnesota, receiving her BS in Computer Science from Winona State in 1989.

Ms. Armon came to Duke University as a James B. Duke Fellow in the fall of 1989, to begin doctoral studies in the Department of Computer Science. Two years into the program, she moved with her husband, Carmel, to Southern California, embarking on a two year killer commute back and forth to Duke.

Since January, 1994, Ms. Armon has been a lecturer in the CS department at the University of California, Riverside, where she teaches introductory programming courses, data structures and algorithms.

Her daughter Orit was born on the first day of Hanukkah, 1994. Her second daughter, Efrat, recently celebrated her first birthday.