A Dynamic Separator Algorithm

Deganit Armon John Reif*

Department of Computer Science, Duke University

Durham, NC 27708-0129

Abstract Our work is based on the pioneering work in sphere separators done by Miller, Teng, Vavasis *et al*, [8, 12], who gave efficient static (fixed input) algorithms for finding sphere separators of size $s(n) = O(n^{\frac{d-1}{d}})$ for a set of points in $\mathbf{R}^d$.

We present dynamic algorithms which maintain separators for a dynamically changing graph. Our algorithms answer queries and process insertions and deletions to the input set,

If the total input size and number of queries is n, our algorithm is *polylog*, that is, it takes $(logn)^{O(1)}$ expected sequential time per request to process worst case queries and worst case changes to the input set. This is the first known polylog randomized dynamic algorithm for separators of a large class of graphs known as *overlap graphs* [12], which include planar graphs and k-neighborhood graphs. We maintain a separator in expected time $O(\log n)$ and we maintain a separator tree in expected time $O(\log^3 n)$. Moreover, our algorithm uses only linear space.

*Supported by National Science Foundation Grant Number NSF-IRI-91-00681 and Army Research Office contract DAAH-04-96-1-0448. A preliminary version of this paper appeared as A Dynamic Separator Algorithm, in the proceedings of the 3rd Annual WADS, Montreal, Quebec, August 1993, pp 107-118.

We also give a general technique for transforming a class of expected time randomized incremental algorithms that use random sampling to incremental algorithms with high likelihood time bounds. In particular, we show how we can maintain separators in time $O(\log^3 n)$ with high likelihood.

Our results 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 solving linear systems and monoid path problems.

1 Introduction

The notion of a *separator* in a graph was introduced in the context of designing efficient divide-and-conquer algorithms for graph problems. A graph separator is a set of graph vertices or edges whose removal from the graph partitions it into two or more separate (i.e. unconnected) subgraphs. A *good* separator, for the purposes of divide-and-conquer, is both small in size and achieves a partition of the graph into subgraphs of roughly equal size. Clearly, not all graphs have good separators. For example, cliques do not have a separator set at all. There are, however, some important classes of graphs that have good separators that can be efficiently computed [6, 5, 3]. These include the classes of planar and almost-planar graphs. Miller, Teng, *et al* [7, 12] introduced *sphere separators*, which are useful in finding good separators for a much wider class of graphs called *overlap graphs.*

Sphere separators are geometric entities, and overlap graphs are geometrically defined, derived from sets of points, and special neighborhoods of these points, in space. This makes them useful when solving problems in computational geometry, such as nearest neighbor queries ([4, 12]), in numerical analysis, such as nested dissection problems with large scale meshes in two and three dimensions [7], and in path algebra problems [10, 11].

The basis for all these algorithms is a *separator-based search structure* [12] – a recursive structure which partitions the input graph or set of points using sphere separators. This structure is a binary tree; information about the separator is stored at the root, and the two partitioned subsets are recursively stored in the subtrees. The separators themselves are found using a randomized algorithm that uses sampling.

In this paper we describe a method for maintaining such a separator structure for dynamically changing input.

1.1 Dynamizing Static Randomized Algorithms

Dynamic (or incremental) algorithms update their solution to a problem when the input is dynamically modified. Usually it is not efficient to recompute the solution "from scratch", so the input is stored in specialized data structures that can be updated at small cost. Dynamic algorithms are very useful in interactive applications, including network optimization, VLSI, and computer graphics. Many dynamic data structures have been devised to deal with problems in computational geometry [9, 1].

In this section, we describe a useful technique for transforming randomized algorithms that use sampling, so that they can cope with dynamically changing inputs. This technique may be applied to a wide range of randomized algorithms. Throughout the rest of this paper we will refer to algorithms with fixed input as *static algorithms.*

Randomized algorithms that use sampling calculate a solution by using information from a small subset of the input. Given an input set of size n, a sample Σ of size σ is used to construct a data structure for the input set. Many efficient algorithms use such random sampling. Consider an algorithm A that uses a sample Σ to construct a data structure which will store the input set. Let the time taken by A be $T(n)$. Now consider an additional point p which is presented to the algorithm to be inserted into the data structure. If a new data structure is constructed for the $n+1$ points, its shape would depend only on the sample Σ. The probability that p is in Σ, and therefore used to construct the new data structure is only $\frac{\sigma}{n+1}$. Thus with probability $1 - \frac{\sigma}{n+1}$ the same data structure is produced regardless of whether p is in the input set.

By performing a single Bernoulli trial, we can decide whether or not the output is affected by the insertion of p. If the answer is “yes”, we recompute the data structure for the new data set by calling A. The probability of recalculating is $\frac{\sigma}{n+1}$. If the answer is “no”, the new point is added to a location in the data structure without otherwise changing it. Since the cost of invoking A is at most $T(n+1)$, the cost of processing a new input point p is at most $\frac{\sigma}{n+1}T(n+1)$.

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

- After a sequence of any number of updates to its input, the dynamic data structure output by the algorithm will have the same probability distribution as a data structure output by the static algorithm, had the updated input been presented to it.

With each new insertion, with probability at most $\frac{\sigma}{n}$ the entire data structure may be completely rebuilt by the static algorithm, using unbiased independent random sampling. After each step, the induction still holds. Section 3 gives more details of this approach.

Deletion of points from the data set are handled similarly, but here care must be taken, depending on the separator structure. If the data points are stored only in the leaves of the structure, deletion can be handled in an identical fashion to insertion. By performing a single Bernoulli trial, we can decide whether or not the data structure is affected by the deletion of p. If the answer is "yes", we recompute the data structure for the new data set by calling A. The probability of recalculating is $\frac{\sigma}{n}$. If the answer is "no", the new point is deleted from the data structure without otherwise changing it. Since the cost of invoking A is at most $T(n-1)$, the cost of deleting a point p is at most $\frac{\sigma}{n}T(n-1)$.

This method of deleting a point is clearly more time consuming than simply removing it from the leaf containing it, but the idea is to make sure that the induction hypothesis stated above holds, i.e., that the data structure after deletion will come from the same probability distribution as a data structure output by the static algorithms given the data set without point p.

If points are stored at intermediate nodes in the data structure, deletion is slightly more involved (see details in section 3.3). However, the time bounds remain the same.

In reality, static algorithms are costly. The static algorithms for finding sphere separators [12] use recursive randomized sampling. When constructing a separator structure [4, 12], there is a cost involved in building the tree even if the new input has no effect on the separator at the root. Nonetheless, this general idea is the basis for designing the dynamic algorithms in this paper, and analysis in all cases is similar.

1.2 Description of Our Model

Using a randomized algorithm, we maintain a separator based search structure for a dynamically changing set of points P in $\mathbf{R}^d$. Our model assumes an adversary, who knows our randomized, dynamic algorithm but not the random choices which it makes. The adversary presents a sequence of points *in advance* which are then presented to the algorithm, one at a time. Note that this is not a fully interactive model, as the adversary does not generate new points once the algorithm is running, and cannot take advantage of knowledge of choices the algorithm has already made. This feature of the model is important only while making deletions to the point set. Associated with each point p is one of the following requests:

1. INSERT p into the search structure.

2. DELETE p from the search structure.

3. Answer a QUERY about p.

Queries presented are to determine if the query point is inside or outside the separator, or to locate the point in the separator tree. A query point may or may not be in P, but it can be temporarily inserted, then deleted, if the application allows queries to be answered only for points from P (as in [4, 13]).

1.3 The Replicant Paradigm: Terminology

When dealing with randomized algorithms we can only speak of expected bounds which can be attained with some probability. One method of attaining the expected bounds with *high* probability is to repeat a process a number of times. In Section 4 we present a general technique for transforming expected time bounds to high likelihood time bounds through the use of multiple, independent processes. This section introduces the terminology relevant to this technique.

The algorithm will maintain many independent processes, all of which carry out the same task. Each independent process and its dynamically maintained data structure is called a *replicant*[1]. 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. 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”], but does have a *memory implant* – knowledge of the current version of the data structure, as it has been built during the most recent retirement.

1.4 Organization of Our Paper

Section 2 gives preliminary definitions and related results. In section 3 we describe the process of converting a randomized algorithm based on sampling to a dynamic randomized algorithm, and give the details of the process of inserting and deleting a point from a separator structure.

[1]In the description that follows we used terminology (in italics) and quotes (in square brackets) borrowed from the movie “Blade Runner”.

Finally, in section 4 we discuss the use of replicants to transform expected time bounds to high likelihood time bounds.

2 Definitions and Related Results

2.1 Neighborhood Systems

Let $P = \{p_1 \dots p_n\}$ 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 $\mathcal{B} = \{B_1 \dots B_n\}$, 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 from [2] relates the two:

Lemma 2.1 *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 are $2^{.2075d(1+o(1))} \leq \tau_d \leq 2^{.401d(1+o(1))}$ [12].

The *k-nearest neighborhood digraph* of the points $P = \{p_1, \dots, p_n\}$ in $\mathbf{R}^d$ is a graph $G_k = (P, E)$, with vertex set P and directed edge set $E = \{(p_i, p_j) | p_j$ 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 2.1, the indegree is bounded by $k\tau_d$.

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. $|P_I| \leq \delta n$, $|P_E| \leq \delta n$, and

2. the *induced separator* of the point set is of size at most $s(n)$.

δ is called the *splitting ration* 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 paper, wherever s, δ are determined by context, we will simply call the (s, δ)-separating sphere a *good* separating sphere.

2.2.1 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 $f(n) = O(n)$ [4].

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. However, if G has a separator set of size $O(n^{\frac{d-1}{d}})$, then the total storage required is $O(n)$ [4].

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 $S \in V$ 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. each separated subgraph G_0, G_1 contains no more than δn vertices
2. S is of size at most s.

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, k)-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 be a ball centered at p of radius r then αB is a ball centered at p with radius αr. Note that k-neighborhood graphs are a special case of overlap graphs.

2.4 Algorithms for Finding Sphere Separators

Our work is based on the pioneering work in sphere separators done by Miller, Teng, *et al* [4, 8, 7, 12]. 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 $\delta^* = \frac{d+1}{d+2}$ [8].

Furthermore, Teng [12] showed a randomized, linear time algorithm based on sampling for finding such a separator. His result states that:

Lemma 2.2 [12] *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 Σ 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(Σ), 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(Σ) 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 2.2, 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 $O(\log n)$.

3 Dynamic Maintenance of Sphere Separators

For a dynamically changing finite set of points P, we give algorithms for dynamic maintenance of a sphere separator of P, and for dynamic maintenance of a separator-based search structure. 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, e.g. determine whether it is inside or outside the sphere separator of P, or search for it in the separator structure.

We show the following result:

Theorem 3.1 *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 QUERY requests can be done in $O(\log^3 n)$ incremental expected time per request.*

The following subsections detail the proof of the theorem. Section 3.1 discusses the case where a separator is maintained for the point set, and describes the algorithm for inserting and deleting points from the data set. Section 3.2 describes dynamically maintaining seoarator trees and the algorithms for inserting and deleting points from the data structure.

Throughout, we use the paradigm of replicants, as defined in Section 1.3. In this section, we focus on the case where there is only one replicant, or process. We use the static algorithm for finding a sphere separator as a subroutine in our dynamic algorithm.

3.1 Maintaining a Separator

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

The algorithm proceeds inductively. Given a point set and a separator for it, a change to the input set (INSERT ro DELETE) may cause the algorithm to calculate a new separator. If this happens, we say that the replicant is being retired for rebuilding. alternately, the separator may remain the same, despite an addition or deletion of a point. The key point is that

- After a sequence of n point insertions and m point deletions, the separator calaculated by the dynamic algorithm comes from the same probability distribution of separators for the $n - m$ point in the point set, had they been presented to the static algorithm.

Specifically, if after a sequence of i updates the set of points in the separator structure is P_i, the separator output by the dynamic algorithm should come from the same probability distribution as the separator output by a static algorithm, given P_i as input.

Note that 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.

3.1.1 Insertion

We now analyze the time needed to insert a single point into the point set, subject to the condition above.

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 Σ which is used to determine the separating sphere. If $|\Sigma| = \sigma(n)$, the probability that the new point had been included in Σ 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 2.2 on the entire set of n points. That is, we perform SPHERE-SEPARATOR(Σ) on the set $P \cup \{p\}$.

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

Lemma 3.1 *Given a set P of $n-1$ points in R^d and a δ-splitting sphere separator S, 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 S the point lies in, and add it to that set (again in constant time). The expected update time is thus

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

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

▮

3.1.2 Deletion

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

Each point in the structure which was used in the sample when finding the separator is tagged. When deleting a tagged point, the separator needs to be rebuilt. If we maintain the entire search structure, a point is tagged with the highest level number in which it was used to find a separator – its highest *involvement level.* Points not selected are not tagged. When a point is presented for deletion, it is located in the separator structure down to its involvement level, then the subtree from that level down is rebuilt. The proofs of the next two lemmas are similar to the proofs of lemmas 3.1 and 3.3.

Lemma 3.2 *Given a set P of n points in R^d and a δ-splitting sphere separator S, the expected time to delete a point p from P is $O(\log n)$.*

3.2 Maintaining a Sphere Separator Tree

If the entire separator structure is maintained, then p is added recursively to the left subtree or right subtree, depending on its location with respect to the separating sphere. If the entire separator tree is maintained, the algorithm keeps track of all samples used to find separators at all levels of the tree. Thus a single point may be tagged up to $O(\log n)$ times. The algorithm proceeds inductively. 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. If this is the case, we say that the replicant structure is being *retired* for rebuilding. After every step, the following induction hypothesis holds:

Similarly, if the entire separator structure is maintained,

Lemma 3.3 *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 $O(\log^3 n)$.*

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:

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

$$\leq \frac{\sigma(n)}{n} O(n \log n) + T(\delta n) = O(\sigma(n) \log n) + T(\delta n) = O(log^2 n) + T(\delta n) = O(\log^3 n).$$

▮

3.3 Deletion

Lemma 3.4 *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 $O(\log^3 n)$.*

4 High Likelihood Time Bounds

In the previous section, we showed what the *expected* time bounds were for our dynamic algorithms. In this section, we'll show how to transform these expected time bounds to *high likelihood time bounds*, using the replicant paradigm. We define *high likelihood* to be $1 - \frac{1}{n^\alpha}$ for some $\alpha \geq 0$ (where n is the input size). There are known techniques for transforming static (non-incremental) algorithms from expected time bounds to high likelihood bounds. The main idea is to repeat the execution of the algorithm a sufficient number of times to guarantee high likelihood time bounds.

For example, due to the randomized nature of the separator algorithm [4], 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 [4] 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 the separator.

Rather than using this method, we use multiple, independent processes, called replicants. Each replicant maintains its own dynamic data structure as described in Section 3, which is either *active* – up-to-date and available to answer queries, or in *retirement* – being rebuilt. When the replicant is active, its data structure satisfies the induction hypothesis specified in Section 3. Our goal is to have, at all times and with high likelihood, enough active replicants that can answer queries about their data structure, such as whether a point is stored in the structure, or where it is with respect to the separator. By dovetailing the work of the replicants, we slow down the algorithm by a factor of the number of replicants. This slowdown is the price we pay for obtaining high likelihood bounds.

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 the data structure is rebuilt. Otherwise, when a replicant is retired, it is of

no further use to the algorithm.

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

The next two subsections deal with these issues.

4.1 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 and answers questions about the data structures and the points stored in it. For example, if the replicants are maintaining a separator for a point set, an update takes expected time $O(\log n)$, and a query can be answered in $O(1)$time.

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 [*"accelerated decrepitude"*]. If it 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. 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 4.1 *Let $R(n)$ 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 $6R(n)$ time.*

Proof:

Let $A(n)$ be the time to process a request with high likelihood, and let $R(n)$ 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 $R(n)$ time. During this time, $\frac{R(n)}{A(n)}$ requests arrive and are stored by the replicant. When rebuilding is done, the replicant is out of date by at most $\frac{R(n)}{A(n)}$ 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 $\frac{R(n)}{A(n)}$ requests each take $\frac{A(n)}{2}$ time to process, so the total time to process this batch of backlog is $\frac{R(n)}{2}$. At the same time, new requests arrive, and are stored for further reference. Because of the quickened pace, there will be at most $\frac{R(n)}{2A(n)}$ requests accumulating. These requests will be processed in the next catch-up stage. Continuing at this pace ["the light that burns twice as fast burns twice as bright"], after i catch-up stages the backlog will be of size $\frac{R(n)}{2^{i-1}A(n)}$ and will take $\frac{R(n)}{2^{i+1}}$ time to process. When the backlog is 1 or less, the replicant can be reactivated. The total time in retirement is therefore

$$\Sigma_{i=0}\frac{R(n)}{2^i} \leq 2R(n).$$

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 $\frac{\sigma(n)}{n}$.

Of the $\frac{R(n)}{A(n)}$ requests that accumulate in the first stage of retirement, $\frac{\sigma(n)}{n}\frac{R(n)}{A(n)}$ may require rebuilding, with each rebuilding taking $R(n)$ time. The remaining $\frac{(n-\sigma(n))}{n}\frac{R(n)}{A(n)}$ can be processed at double speed as above, each taking $\frac{A(n)}{2}$ time. Thus the total time to process the backlog accumulated during the first stage of retirement is

$$\frac{\sigma(n)}{n}\frac{R(n)}{A(n)}R(n) + \frac{(n-\sigma(n))}{n}\frac{R(n)}{A(n)}\frac{A(n)}{2} = \frac{2\sigma(n)R(n)^2 + (n-\sigma(n)R(n)A(n)}{2nA(n)}.$$

The new backlog accumulating during this time is $\frac{2\sigma(n)R(n)^2+(n-\sigma(n))R(n)A(n)}{2nA(n)^2}$. Again, a small fraction of this will require rebuilding. The time to complete the ith stage of retirement is

$$\left(\frac{2\sigma(n)R(n) + (n-\sigma(n))A(n)}{2nA(n)}\right)^i R(n).$$

Thus the total time in retirement is

$$\Sigma_{i=0}\left(\frac{2\sigma(n)R(n) + (n-\sigma(n))A(n)}{2nA(n)}\right)^i R(n).$$

This summation converges when the term raised to the i is less than 1. Given values for $R(n)$ and $\sigma(n)$, we can find out for which values of $A(n)$ the summation converges. In particular,

$$\frac{2\sigma(n)R(n) + (n-\sigma(n))A(n)}{2nA(n)} < 1$$

when

$$A(n) > \frac{2\sigma(n)R(n)}{n+\sigma(n)}.$$

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

If we let $A(n) = \frac{3\sigma(n)R(n)}{n+\sigma(n)}$, the base of the power term becomes

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

and the length of each retirement is bounded by

$$R(n) \times \frac{1}{1-\frac{5n-\sigma(n)}{6n}} = R(n)\frac{6n}{n+\sigma(n)} < 6R(n).$$

∎

Note: larger values of A will result in a shorter retirement periods. For example, for $A(n) = \frac{4\sigma(n)R(n)}{n+\sigma(n)}$, the length of a retirement period is bounded by $4R(n)$. Thus we can state the following corollary:

Corollary 1 *Let $R(n)$ be the time to build the data structure using the static algorithm. A replicant that is retired for rebuilding will be reactivated after $O(R(n))$ time.*

4.2 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 $\frac{1}{n^\alpha}$, 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 4.2 *The number of replicants necessary to ensure that, with high likelihood, there are active replicants at any time, is* $O(\frac{\sigma(n)R(n)\log n}{n})$.

Proof:

Let the time per update or request if no retirement occurs (i.e. during activation) be $A(n)$ with high likelihood. If retirement does occur, let the wait time until reactivation be $6R(n)$ with high likelihood. The expected time per update is then $T(n) = (1-\frac{\sigma(n)}{n})A(n)+\frac{\sigma(n)}{n}6R(n)$.

Assume for simplicity that $\frac{n}{\sigma(n)R(n)} = o(1)$. This is true for the applications described in this paper.

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 catch-up time for a retired replicant be $6R(n)$ (as described in the previous subsection). The life of a replicant consists of periods of activation of expected length λ, followed by periods of retirement of length $6R(n)$.

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

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

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 $1 - \frac{1}{n^\alpha}$, is

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

Since $\frac{n}{\sigma(n)R(n)} = o(1)$, as n increases $\gamma \to 0$, and $\log(1-\gamma) \to -\gamma$. Consequently,

$$r = \frac{-\alpha \log n}{\log(1-\gamma)} \to \frac{\alpha \log n}{\gamma} = \frac{\alpha \log n(3\sigma(n)R(n) + n)}{n} = O(\frac{\sigma(n)R(n)\log n}{n}).$$

∎

4.3 Maintaining Sphere Separators with High Likelihood

The two results above can be combined to give the following theorem:

Theorem 4.1 *Given an incremental, randomized algorithm with expected time $T(n)$, which uses a random sample of size $\sigma(n)$ of the input, and uses a static algorithm with time bound $R(n)$ as a subroutine when rebuilding its data structure, we can transform it to an incremental algorithm with high likelihood time bounds with a slowdown factor of $O(\frac{\sigma(n)R(n)\log n}{n})$.*

For the specific case of maintaining sphere separators for a set of points in R^d, we can determine the both the high likelihood time bound $A(n)$ and the number of replicants r, given that $R(n)$ is $O(n)$, and $\sigma(n) = O(\log n)$.

In order for the retirement period to be finite, we require, from Lemma 4.1 above, that $A(n) > 2\sigma(n)R(n)/(n + \sigma(n))$. We can satisfy this condition by choosing $A(n) = 2\log n$.

Note that this choice of $A(n)$ means that the length of retirement becomes $4R(n)$. However, this has no significant effect on the number of replicants.

From Lemma 4.2 we know that the number of replicants necessary to satisfy the high likelihood condition is $O(\frac{\sigma(n)R(n)\log n}{n})$.

Since $R(n) = O(n)$, and $\sigma(n) = O(\log n)$ in our application, the number of replicants (which is the slowdown factor) is only $\log^2 n$.

Combining these two results, we get:

Corollary 2 *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* $T(n) = O(\log^3 n)$.

5 Applications and Further Work

Our results for dynamic separators can be applied to generate dynamic algorithms for a wide variety of combinatorial and numerical problems which have an underlying associated k-neighborhood graph. For example,

(i) Computational geometry problems, such as dynamic k-nearest neighbor.

(ii) Solution of linear systems and inverses of dynamic matrices, with changes both in the numeric values of entries in the matrix, as well as modifications that change the sparsity structure of the matrix.

(iii) Monoid path problems on dynamic graphs, with vertex and edge insertion and deletion, as well changes to edge labels.

References

[1] Y. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Tech Report CS–91–24*, Department of Computer Science, Brown University, 1991.

[2] 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.

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

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

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

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

[7] 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.

[8] 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.

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

[10] V. Pan and J.H Reif. Extension of the parallel nested dissection algorithm to path algebra problems. *Proc. Sixth Conference on Foundation of Software Technology and Theoretical Computer Science*, New Delhi, India, Lecture Notes in Computer Science, 241, 1986; full version: Fast and Efficient Solution of Path Algebra Problems. *Journal of Computer and Systems Sciences* 38(3):494–510, 1989.

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

[12] 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.

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