Searching Dynamic Point Sets in Spaces with Bounded Doubling Dimension*

Richard Cole
cole@cs.nyu.edu

Lee-Ad Gottlieb
adi@cs.nyu.edu

Courant Institute
New York University
New York, NY 10033

ABSTRACT

We present a new data structure that facilitates approximate nearest neighbor searches on a dynamic set of points in a metric space that has a bounded doubling dimension. Our data structure has linear size and supports insertions and deletions in $O(\log n)$ time, and finds a $(1+\epsilon)$-approximate nearest neighbor in time $O(\log n) + (1/\epsilon)^{O(1)}$. The search and update times hide multiplicative factors that depend on the doubling dimension; the space does not. These performance times are independent of the aspect ratio (or *spread*) of the points.

Categories and Subject Descriptors: F.2.2 [Nonnumerical Algorithms and Problems]:Sorting and searching, computations on discrete structures; E.1 [Data Structures]:Graphs and networks, trees.

General Terms: Algorithms.

Keywords: Approximate nearest neighbor search.

1. INTRODUCTION

Nearest neighbor. Approximate nearest neighbor search (ANN) is one of the basic operations computed on data sets comprising numeric vectors, i.e. points. The problem asks to preprocess a set X of points in a certain metric space M, so that given a new query point $q \in M$, a point near to q in X can be located efficiently. This problem has applications in data mining, database queries and related fields.

For high dimensional metrics, ANN may require significant computation time. This is due to the inherent complexity of the metric, and has been dubbed the "curse of dimensionality." Hence it is natural to study ANN techniques for point sets which are effectively lower dimensional, although inhabiting a high dimensional space.

A recent successful approach has been to consider point sets that have a small *doubling dimension*: Let the space within radius r of a point be called the *ball* centered at that point. If point set X has doubling dimension λ, then all points of X that are *covered* by a ball of radius r can be covered by 2^λ balls of radius $\frac{r}{2}$. A metric is *doubling* if its dimension is $O(1)$. While a low Euclidean dimension implies a low doubling dimension (Euclidean metrics of dimension d have doubling dimension $O(d)$ [12]), low doubling dimension is more general than low Euclidean dimension. For example, exact nearest neighbor in metric spaces with low doubling dimension may require $\Theta(n)$ computations (this follows easily from [13]), while for d-dimensional Euclidean space Clarkson [8] has given an $O(n^{\lceil d/2 \rceil (1+\epsilon)})$ data structure that answers exact nearest neighbor queries in $O(\log n)$ time (with constant factors in the bounds depending on d and ϵ, $\epsilon > 0$). By contrast, for approximate nearest neighbor one can achieve the same results for low doubling dimension as are possible for low Euclidean dimension.

Krauthgamer and Lee [15] applied navigating nets to the ANN problem. For a set of points X, a subset $Y \subseteq X$ is an ϵ-net if it satisfies the following conditions: (i) For every $x, y \in Y$, the distance between x and y is $d(x,y) \geq \epsilon$, and (ii) every point of X is within distance ϵ of some $y \in Y$. These conditions require that the points of an ϵ-net be spaced out, yet nevertheless cover all points of X.

A navigating net is composed of levels of ϵ-nets. Consider the maximum and minimum interpoint distances in X, d_{min} and d_{max}. For all scales $r \in \Gamma$, where $\Gamma = \{2^i : \lfloor \log d_{min} \rfloor \leq i \leq \lceil \log d_{max} \rceil\}$, we require that the net level Y_r be an r-net of the lower level $Y_{r/2}$. (For ease of notation, we set $d'_{min} = 2^{\lfloor \log d_{min} \rfloor}$ and $d'_{max} = 2^{\lceil \log d_{max} \rceil}$.) The bottom level of the navigating net, $Y_{d'_{min}}$, contains all points of S. The top level of the navigating net, $Y_{d'_{max}}$, contains a single point. [15] showed how to store a navigating net implicitly. They then created a directed acyclic graph D by adding a directed edge from each point $y \in Y_r$ to all points of $Y_{r/2}$ that are within distance γr of y (for some appropriate constant γ). Now, each point in the navigating net may be reached from $O(1)$ points a level above.

An approximate nearest neighbor search on the navigating net is accomplished as follows. Beginning at the single top point and descending to a stopping level specified below, the algorithm keeps track of all points of the current scale r that are within distance γr of the query point, and also of the closest point encountered thus far. This can be done efficiently since (i) the doubling and packing properties guarantee that there are no more than $O(1)$ points of Y_r at distance γr of the query point, and (ii) all pertinent points of $Y_{r/2}$

*Authors supported in part by NSF grants CCR 0105678 and IIS 0414763.

STOC'06, May21–23, 2006, Seattle, Washington, USA.

(those that are within distance $\gamma r/2$ of the query point), are necessarily connected to the pertinent points of Y_r. Eventually, the algorithm either reaches the bottom level (and has necessarily encountered the nearest neighbor), or reaches a level $Y_{r'}$ in which no points are within distance $\gamma r'$ of the query point. In the latter event, only $(1/\epsilon)^{O(1)}$ more work is necessary to find an approximate nearest neighbor by descending $O(\log(1/\epsilon))$ more levels, as explained in [15].

A core difficulty with this approach is its dependence on the height (or number of levels) of D. This in turn is determined by the aspect ratio, or *spread*, of the points, that is $\Phi = d_{max}/d_{min}$. The height of a navigating net is directly proportional to $\log \Phi$, which is acceptable if $\Phi = \text{poly}(|X|)$, but problematic if Φ is asymptotically greater. Both the query and construction times used in [15] are given in terms of the aspect ratio – these operations require $O(\log \Phi)$ time. (The paper claims a slightly worse bound for insertions and deletions, but actually $O(\log \Phi)$ suffices.)

Related work. Beygelzimer et al. [5] showed how to improve on the space required by [15]. Krauthgamer and Lee [16] gave an alternate static data structure for ANN under the assumption of constant doubling dimension, obtaining results that are independent of the aspect ratio. For constant doubling dimension, their static data structure uses space $O(n^2)$ and answers queries in $O(\log^2 n)$ time, while making $O(\log n)$ calls to the distance oracle.

Improving on this result, Har-Peled and Mendel [13] constructed a linear space ring–separator structure that provides an n^c-approximate nearest neighbor to the query point in $O(\log n)$ time. While an n^c-approximation is very coarse, it provides enough information to "jump in" to a net-tree structure at a position that is within $O(\log n)$ levels of the desired approximation (as discussed there). This clever observation completely circumvents the reliance on aspect ratio. However, the construction of [13] does not support insertions or deletions into the point set. Using the dynamization technique of Bentley and Saxe [4], this structure may be extended to allow for insertions, thereby producing a semi-dynamic structure, but then an insertion would require $\Theta(\log^2 n)$ time. It is not clear how to maintain the structure of [13] either dynamically or even semi-dynamically in $O(\log n)$ time given the special order that they impose on points when building their tree structure.

Other related research on nearest neighbor searches has focused on various assumptions concerning the metric space. Clarkson [9] made assumptions concerning the probability distribution from which X and q are drawn, and developed two randomized data structures for exact nearest neighbor. However, the query time is super-logarithmic, and the structures do not support insertion or deletion of points. Karger and Ruhl [14] introduced the notion of growth-constrained metrics (elsewhere called the KR-dimension) which is a weaker notion than that of the doubling dimension. They presented an $O(n \log n)$ space data structure on which a randomized algorithm finds an approximate nearest neighbor in $O(\log n)$ time, and which allows insertions and deletions in $O(\log n \log \log n)$ time. A survey of proximity searches in metric space appeared in [7].

Our results. We present a linear size data structure that supports insertions and deletions in $O(\log n)$ time, and finds a $(1+\epsilon)$-approximate nearest neighbor in time $O(\log n) + (1/\epsilon)^{O(1)}$. These bounds are independent of the aspect ratio.

The main novelty in our solution is to devise a method for searching on D in $O(\log n) + (1/\epsilon)^{O(1)}$ time, independent of the spread. The basic idea is to extract a spanning tree S of D and to use S to guide the search. In general, the search will be confined to a constant number of subtrees of S. At each step, the search identifies a node s, whose weight splits the weight of the largest of the subtrees by a constant fraction. If the query point is far from the point represented by node s, we can rule out all of s's descendants, and thereby significantly reduce the number of points under consideration. Otherwise the query point is close to s, and we can rule out points far away from s. Ruling out the points far away from s is the interesting case, for as we shall see, this results in identifying a new constant-sized set of subtrees in which the search will continue.

To enable rapid identification of nodes s in S, a centroid path decomposition of S is maintained, with each centroid path being stored in a suitable weighted search structure (we use the biased skip list of [2]). The challenge we face is to maintain D and the associated spanning tree S under point insertion and deletion. Note that the construction of [15] requires creating or updating $\Theta(\log \Phi)$ neighbor lists following a single insertion, which for large Φ is $\omega(\log n)$ neighbor lists. We devise a technique that requires only $O(1)$ operations on the DAG for a single update.

A straightforward application of our data structure is the extraction of a well separated pairs decomposition in linear time. We omit the details in this paper.

We conclude the introduction by contrasting our data structure with the logarithmic time solution in $\Re^d$ due to Arya et al. [1]. Their solution uses a tree-based hierarchical partitioning of $\Re^d$ and seeks the $O(1)$ leaf regions nearest the query point by means of a DFS search. Thanks to the space partitioning, only $O(\log n)$ regions need be examined. In doubling dimension metrics, it is not clear how to build, or even that there exists, such a hierarchical space partition. Instead, the notion of friends (nearby regions or balls of equal size) is used to control the extent of the search. However, maintaining an unbalanced search tree so that the subtrees of friends can be effectively searched, while achieving logarithmic time, appears to need an approach quite distinct from that of [1].

We demonstrate in Section 2 how to maintain the navigating net and its corresponding DAG D and spanning tree S. In Section 3 we show how to use D and S to find a nearest neighbor. In Section 4 we address some issues that arise when deleting a point, and in Section 5 we discuss how our structure can be implemented in truly linear space.

2. NAVIGATING NET MAINTENANCE AND THE SPANNING TREE

Recall that the navigating net utilized in [15] was a collection of Y_r-nets. The Y_r points were at minimum distance r from each other (the *packing property*), and each $Y_{r/2}$ point was *contained* within a radius of r of some point in Y_r (the *covering property*). Navigation in the nets was supported by neighbor lists, where a neighbor list of $x \in Y_r$ is the set of all points $z \in Y_{r/2}$ that are distance at most γr away from x. Note that the neighbor lists form a DAG over the set of (*point, level*) pairs. These neighbor lists allowed the search procedure to descend a single level, while keeping track of the points relevant to the search.

The navigating net we use is also a collection of Y_r-nets (represented explicitly), but we define a Y_r-net as a collection of balls of radius r, whose centers are some minimum distance apart. A ball's *volume* includes just those locations strictly inside its circumference. Rather than use values of r that grow by factors of 2, we have r grow by factors of b. To emphasize this, we denote the nets as Y_{b^i}, where i is as small as needed. Later, we show that we require $b \geq 5$, so we will set $b = 5$. We also have a leaf level net Y_0 which includes all the points, and a root level net Y_∞ which contains a single ball.

Each ball $x \in Y_{b^k}$ is partitioned into five rings, each of width $\frac{b^k}{5}$. (The purpose for this will become clear in Section 2.1.2.) Each ring includes the locations lying on its inner circle, but not those locations lying on its outer circle. The rings are labeled one to five from outermost to innermost.

Each ball $x \in Y_{b^k}$ is centered at some point $p = c(x)$; when convenient, we may refer to x as $\langle p, b^k \rangle$. Note that a single point p may be the center of multiple balls, as long as they are in different levels.

We define $d(x, y)$, the distance between two balls $x \in Y_{b^k}$ and $y \in Y_{b^m}$, to be the distance between their centers. We will say that x is in a ring of y if $m > k$ and x's center is in that ring. x is *within* a ring of y if it is in that ring or is in a different ring that is closer to the center of y. We will say that y *contains* the center of x (or contains x, for short) if $m > k$ and $d(x, y) < b^m$; in addition, if $m = k + 1$, then y is a parent of x, and x a child of y. x could have multiple parents.

Finally, we introduce two relaxations in the definition of Y_{b^k}-nets: (i) We relax the covering property, the condition that each ball of Y_{b^k} (for a non-root level net Y_{b^k}) is contained in some ball of $Y_{b^{k+1}}$. Instead, we require only that each ball of Y_{b^k} be contained in some higher level ball; that is, for $x \in Y_{b^k}$ there exists some $y \in Y_{b^m}$ $(m > k)$ such that $d(x, y) \leq b^m$. Note that this definition allows for *jumps*, where a Y_{b^k} ball is contained in a much higher level ball, and in no balls at intermediate levels. y and x are also called a parent-child pair, but here x will have just one parent. Because we will stipulate that all inserted points have a corresponding ball in the Y_0 level, to achieve linear space it appears necessary to incorporate jumps into our definition. Note however, that just as in [15], each ball of Y_{b^k}, $b^k \neq 0$, has a child with the same center, although that child may be at level $Y_{b^{k-2}}$ or lower. (ii) We relax the packing property, so that balls of Y_{b^k} must be separated by a distance of at least b^{k-1} (rather than b^k).

Our search structure is also a DAG with the balls as vertices, and edges given by the parent-child pairs. In addition, each ball will have pointers to its parents and *friends*. The friends of $x \in Y_{b^k}$ are all the balls of Y_{b^k} that are within distance $2b^k$ of x. (Note that the packing property implies that a ball has $O(1)$ friends.)

More formally, the navigating net satisfies the following conditions:

- Y_0 (the *leaf level*) contains n balls, one ball centered at each of the n points. Y_∞ (the *root level*) contains a single ball.
- *Packing property:* For every $x, y \in Y_{b^k}$, $d(x, y) \geq b^{k-1}$.
- *Covering property:* With the exception of the root ball, for every ball $x \in Y_{b^k}$ there is an $m > k$ such that x is contained within a ball $y \in Y_{b^m}$.

As mentioned before, the third condition allows for jumps. The existence of a jump implies a degree of separation from other balls, which is captured by the following invariant:

INVARIANT 1. *If there is a jump from $y \in Y_{b^m}$ down to $w \in Y_{b^k}$, then the following jump conditions hold:*

(i) w and y are centered at the same point, $c(w) = c(y)$.

(ii) Ring 5 of y must be disjoint from rings 2–5 of each ball of level Y_{b^m} or lower which is not a descendant of w. That is, if $z \in Y_{b^j}$, $j \leq m$, and z is not a descendant of w, $d(z, y) \geq \frac{1}{5}b^m + \frac{4}{5}b^j$.

A jump from $y \in Y_{b^m}$ down to $w \in Y_{b^k}$ may be *split* by inserting a ball $x = \langle c(y), b^l \rangle$, $m > l > k$, that breaks the jump into two. Note that x obeys the packing property with respect to any ball $z \in Y_{b^l}$, since $d(x, z) = d(y, z) \geq \frac{1}{5}b^m + \frac{4}{5}b^l \geq \frac{1}{5}b^{l+1} + \frac{4}{5}b^l > b^l > b^{l-1}$.

Next, we describe the close-containment property. We will show that this property is key to allowing updates in $O(1)$ time (Theorem 1) and in executing the ANN search (Lemma 5).

INVARIANT 2. *(Close-containment property) If a ball has ancestors in some level, then all its rings are strictly contained within the second ring of one of those ancestors. That is, if $x \in Y_{b^k}$ has ancestors in Y_{b^m}, then x is strictly within distance $\frac{4}{5}b^m - b^k$ of at least one of them.*

In Section 2.1, we describe how to insert points into our structure. In order to maintain close-containment in the presence of insertions, we will ensure that if a Y_{b^m} ball y acquires a new distant descendant x, then if not already present, a ball closer to x than y is added to Y_{b^m}, as will be discussed in Section 2.1.2. In Section 2.1.1, we show how to maintain Invariant 1.

2.1 Insertion Rules

For the insertion of the first point p_0, we create a jump from $\langle p_0, Y_\infty \rangle$ to $\langle p_0, Y_0 \rangle$. For the insertion of a subsequent point p, or rather the insertion of a leaf node $s = \langle p, 0 \rangle$ representing p, we first split an existing jump (if necessary), and then insert a new jump from a new ball t down to s. More precisely, we carry out a containment search (described in Section 3) to locate the lowest balls that contain s. Call this set of balls B, and their level Y_{b^m}. Let $x \in B$ be the ball closest to s, $x = \langle q, b^m \rangle$.

INSERTION RULE 1. *(IR1) s is contained in the fifth ring of x, or x is the root ball.*
(i) x is necessarily the top of a jump (or else its child centered at $c(x)$ would have been the lowest ball containing s). Let k be the smallest integer for which $d(s, x) < b^{k-1}$. Split this jump by creating a ball $w = \langle q, b^k \rangle$, making w a child of $x \in Y_{b^m}$ (unless $k = m$, in which case $w = x$), and assign to w a child $v = \langle q, b^{k-1} \rangle$. v becomes the parent of the bottom ball of the jump. Clearly, if $m > k + 1$, this introduces a jump at a non-leaf level.
(ii) Insert $t = \langle p, b^{k-1} \rangle$ as a child of w and parent of s.

INSERTION RULE 2. *(IR2) s is contained in rings 1–4 of non-root x.*
(i) If x is the bottom of a jump, split the jump by creating a ball $y = \langle q, b^{m+1} \rangle$, the new parent of x.
(ii) Let y be the nearest parent of x at level $Y_{b^{m+1}}$. Assign to y a child $t = \langle p, b^m \rangle$; t becomes the parent of s.

Note that all balls created by the insertion rules obey the packing property. As noted after Invariant 1, this is true for balls created by jump splits, so we need only demonstrate this for t. For IR1, s was not contained in any balls of $Y_{b^{k-1}}$ before the jump was split, so that the distance from t to any of these balls is at least $b^{k-1} > b^{k-2}$. With regard to v, k was chosen such that $d(s,x) = d(t,v) \geq b^{k-2}$, so t obeys the packing property. For IR2, x is the Y_{b^m} ball closest to t; s is contained in rings 1–4 of x, so that $d(s,x) = d(t,x) \geq \frac{b^m}{5} = b^{m-1}$, and t obeys the packing property.

Additional balls may be created by the jump rules of Section 2.1.1 and the promotion rules of Section 2.1.2.

2.1.1 Jump Maintenance

We now add three rules that preserve Invariant 1 during the execution of an insertion or a promotion. These rules will require jump splits only.

Recall that a jump from $y \in Y_{b^m}$ down to $x \in Y_{b^k}$ implies that any ball $z \in Y_{b^j}$, $j \leq m$, strictly within distance $\frac{1}{5}b^m + \frac{4}{5}b^j$ of x is a descendant of x, and is therefore strictly within distance $\frac{4}{5}b^k - b^j$ of x. We will also say that y contains a jump. We use the term ball *addition* to refer exclusively to the creation of a ball as a result of either an insertion of a new jump or a promotion (as discussed in Section 2.1.2), and not to the creation of a ball as a result of a jump split.

JUMP RULE 1. *(JR1) If $z \in Y_{b^m}$ is to be added strictly within distance $\frac{4}{5}b^m + b^{m-1}$ of one of its friends y, and y contains a jump, then* fix *the jump by creating a $Y_{b^{m-1}}$ ball centered at $c(y)$. The jump fix occurs* prior *to the addition of z.*

JUMP RULE 2. *(JR2) If a jump is inserted from $y \in Y_{b^m}$ to $x \in Y_0$, and y has a friend strictly within distance $\frac{4}{5}b^m + b^{m-1}$, then fix the jump as in JR1.*

JUMP RULE 3. *(JR3) If $x \in Y_{b^k}$ is at the bottom of a jump, and z is to be added as a friend of x, then fix the jump by creating a $Y_{b^{k+1}}$ ball centered at $c(x)$. The jump fix occurs prior to the addition of z.*

Comment. Note that for insertions, the event that causes JR3 is already addressed by IR2(i). Thus, JR3 is only needed for promotions.

2.1.2 Promotions

The insertion rules ensure that several $Y_{b^{k-1}}$ balls must be inserted as children of $y \in Y_{b^k}$ before one of them will be at distance $\frac{4}{5}b^k - b^{k-1}$ or greater from y. The multiple insertions over which this occurs gives us an opportunity to *promote* one of these balls (say $x \in Y_{b^{k-1}}$) to Y_{b^k}. x is promoted by inserting a new higher level ball centered at $c(x)$. Note that ball x remains in $Y_{b^{k-1}}$. The promotion of x allows the new ball in Y_{b^k} to become a parent for balls that are too far from y.

Such a promotion scheme could result in a difficulty, however. Suppose we were to decide that a $Y_{b^{k-1}}$ ball is promoted if it is at distance $\frac{4}{5}b^k - b^{k-1}$ or greater from its closest parent. Let $x \in Y_{b^{k-1}}$ be the first ball which is inserted at distance at least $\frac{4}{5}b^k - b^{k-1}$ from its parent, and which is subsequently promoted to create ball $x' \in Y_{b^k}$. Now, if x' is itself at distance $\frac{4}{5}b^{k+1} - b^k$ or greater from its parent, we will wish to promote x'. Thus, this scheme may result in a cascading series of promotions that cannot be done in the allotted time.

A subtler approach is called for. We use the five rings of the balls to recursively classify balls according to their need for promotion. Specifically, we define the notions of *safe* and *supersafe* balls. Promotions are used to maintain ball safety.

Definition. Safe and supersafe balls are defined as follows:

- The root ball is supersafe.
- A ball is supersafe if it is in the fifth ring of its parent.
- A ball is supersafe if it is in the fourth ring of its supersafe parent.
- A ball is safe if it is in the fourth ring of its safe parent.
- A ball is safe if it is in the third ring of its supersafe parent.

Loosely speaking, insertions of $Y_{b^{k-1}}$ balls in the second ring induce promotions into Y_{b^k}, and (earlier) insertions of $Y_{b^{k-1}}$ balls in the third ring induce promotions at some higher level. More precisely, a promotion may occur when a ball x is inserted at level $Y_{b^{k-1}}$ as a child of y. When x is in the second ring of y, x is promoted to level Y_{b^k}. When x is in the third ring of y, there may be a promotion of a different ball to a level above Y_{b^k}; the ball to be promoted, if any, is recorded at y, and is called the *obligation* of y.

The obligation for $y \in Y_{b^k}$ is instantiated when y is created by an insertion or a promotion as follows. Let $z \in Y_{b^{k+1}}$ be the nearest parent of y, if any. If y is in z's third ring, then y receives obligation $\langle y \rangle$. If y is in z's fourth ring, then y's obligation is the same as z's obligation; if z does not have an obligation, y will have no obligation. Similarly, if y is in the fifth ring of z or y is the root, then y has no obligation. (Note that if y is safe then it must have an obligation.)

When a promotion of x is about to be performed, we first check to see if x lies in the fifth ring of one of its parents (this is possible only if that parent was created subsequent to the creation of x); if so x has become supersafe, and no promotion occurs. It follows that a ball created by a promotion always obeys the packing property.

In terms of causal ordering, we view a promotion as occurring *prior* to the insertion which necessitates the promotion. This perspective simplifies the proofs of Section 2.2.

The following property is key to maintaining ball safety.

PROPERTY 1. *A newly added ball x whose center is in ring i of its parent can also overlap only rings $i \pm 1$.*

To maintain this property, it suffices to assign each ring width b^{k-1}; that is, $b \geq 5$. Accordingly, we have set $b = 5$.

LEMMA 1. *All balls are safe or supersafe.*

Proof. In turn, we consider balls created by jump splits, insertions of a jump and promotions.

The new ball added by a jump split is always supersafe, since its parent has the same center.

We turn to an insertion of a new ball $t \in Y_{b^k}$, the top of a jump down to the leaf level. Assume the lemma held prior to the insertion. For IR1, t is contained in the fifth ring of w, and so it is supersafe. For IR2: (i) If t is in the fifth ring of y, then t is supersafe. (ii) If t is in the fourth ring of y, then t is supersafe if y is supersafe, and safe if y is safe. (iii) If t is in the third ring of a supersafe ball y, t will be safe. If t is in the third ring of a safe ball y, t provokes a promotion of a different ball at a higher level. But this promotion changes the whole chain of such ancestors up to and including y to be supersafe; t will therefore be safe. (iv) If t is in the second ring of y, then t is promoted and becomes supersafe, since it has the same center as its parent. (v) If t is in the first ring of y, then there must be a ball $w \in Y_{b^k}$ in the second ring of y. But then w would not be safe or supersafe, contradicting the assumption that the lemma held before the insertion.

Consider a promotion next, and assume that the lemma held prior to the promotion. A ball $s \in Y_{b^k}$ can be promoted to create $s' \in Y_{b^{k+1}}$ if s has obligation $\langle s \rangle$, or if s is newly inserted in the second ring of its closest parent $y \in Y_{b^{k+1}}$. In either event, there must be a Y_{b^k} ball in y's third ring: If s possesses obligation $\langle s \rangle$, it must be that s is in the third ring of y. If s is in the second ring, then the insertion rules imply that there must be a ball $w \in Y_{b^k}$ in the third ring of y. Since y has a ball in its third ring, y must be supersafe. Now note that $d(y, s') = d(y, s) < b^{k+1}$, and also that y must have a parent $z \in Y_{b^{k+2}}$. (If y was at the bottom of a jump, then JR3 was applied before the promotion of s, giving y a parent in $Y_{b^{k+2}}$.) If y is supersafe by being in the fourth ring of its supersafe parent z, then s' is either in the fourth or third ring of z, and is respectively supersafe or safe. If y is supersafe by having the same center as its safe parent z, then s' is in the fifth ring of z and is supersafe. □

2.2 Correctness of Invariants

We show that the rules for insertions, jumps and promotions maintain both invariants.

LEMMA 2. *Invariant 2 holds.*

Proof. Lemma 1 demonstrates that all balls are safe or supersafe, hence every ball lies in the fifth, fourth or third ring of its parent. It follows that the distance from a ball in level Y_{b^k} to its closest ancestor in Y_{b^m} is strictly less than $\frac{3}{5}b^m + \frac{3}{5}b^{m-1} + \ldots + \frac{3}{5}b^{k+1} = \frac{3}{4}b^m - \frac{3}{4}b^k = \frac{4}{5}b^m - b^k - \frac{1}{20}b^m + \frac{1}{4}b^k \leq \frac{4}{5}b^m - b^k$. □

Before proving that Invariant 1 holds, we enumerate the operations in the order that they may occur when a point is added to the point set; these involve jump splits and addition of balls. Recall that we view a promotion as preceding the insertion that necessitates it, and also that JR1 and JR3 are applied before the insertion or promotion that violate the relevant jumps. The order is: (i) JR1 and JR3 are applied prior to a promotion that violates a neighboring jump. (ii) A promotion creates a new ball. (iii) JR1 is applied prior to an insertion of a jump that violates a neighboring jump. (iv) Either IR1 or IR2 is applied; these consist of jump splits and the insertion of new jump. (v) JR2 is applied to the newly inserted jump. Of course, for any given insertion only some of these may occur.

We now show that the jump rules preserve Invariant 1, and also that a single jump fix suffices for a violated jump. We must first prove that a ball created by a jump split (and therefore, by a jump fix) does not violate another jump close by, so that the creation of such a ball does not necessitate a further jump fix.

LEMMA 3. *Let J and J' be two jumps from $y \in Y_{b^m}$ down to $w \in Y_{b^k}$ and from $y' \in Y_{b^{m'}}$ down to $w' \in Y_{b^{k'}}$, respectively. Suppose J is split by adding $x \in Y_{b^l}$, $k < l < m$, with the same center as y (and w), possibly creating jumps J_t from y to x and J_b from x to w. Then J_t, J_b and J' obey Invariant 1.*

Proof. If y is a descendant of w' or if y' is a descendant of w, then the lemma is clear. Otherwise, we consider each jump separately.

J_t obeys Invariant 1: (i) y and x are centered at the same point. (ii) Before the jump was split, any ball $z \in Y_{b^j}$, $j \leq m$, with $d(z, w) < \frac{b^m}{5} + \frac{4}{5}b^j$ was a descendant of w. As x is the parent of w, it follows that any ball $z \in Y_{b^j}$, $j \leq m$, with $d(z, x) < \frac{b^m}{5} + \frac{4}{5}b^j$ is a descendant of x.

J_b obeys Invariant 1: (i) w and x are centered at the same point. (ii) Before the jump was split, any ball $z \in Y_{b^j}$, $j \leq l < m$ with $d(z, w) < \frac{b^m}{5} + \frac{4}{5}b^j$ was a descendant of w. After the split, any ball $z \in Y_{b^j}$, $j \leq l$, with $d(z, x) < \frac{b^l}{5} + \frac{4}{5}b^j$ remains a descendant of w.

J' obeys Invariant 1: (i) w' and y' are centered at the same point. (ii) We note that before J was split, no balls violated Invariant 1 with respect to J', so we need only prove that x does not violate the jump at J'. We assume then that $l \leq m'$. If $m' \geq m$, the jump at y' implies that $d(y', y) = d(y', x) \geq \frac{b^{m'}}{5} + \frac{4}{5}b^m > \frac{b^{m'}}{5} + \frac{4}{5}b^l$. If $m' < m$, then the jump at y implies that $d(y', y) = d(y', x) \geq \frac{b^m}{5} + \frac{4}{5}b^{m'} > \frac{b^{m'}}{5} + \frac{4}{5}b^l$. □

Thus, a jump split does not violate a neighboring jump. Further, jump fixes are applied only to a newly inserted jump, or a jump that is directly violated by a newly added ball; jump fixes do not cascade. We now prove that Invariant 1 holds following a promotion or insertion. Recall first that a newly added ball $x \in Y_{b^k}$ must have a parent in $Y_{b^{k+1}}$: For insertions, this is apparent from the insertion rules. For promotions, this follows from the fact that a ball b may be promoted only if it is within the second ring of its parent; by close-containment, the level Y_{b^k} ball created by the promotion will be contained in the grandparent of b at level $Y_{b^{k+1}}$. (JR3 ensures that the grandparent must exist.)

LEMMA 4. *The jump rules preserve Invariant 1 following a promotion or insertion. Further, a jump whose top is in Y_{b^m} is only violated by balls in Y_{b^m}, and any violated jump requires a single jump fix.*

Proof. Let J be a preexisting jump from $y \in Y_{b^m}$ to $x \in Y_{b^k}$, and let $y' \in Y_{b^{m'}}$ be a new ball added either because of a promotion, or because of an insertion of a jump J' from y' down to $x' \in Y_0$. We show that the jump rules preserve Invariant 1 with respect to J. We consider three cases:

Case 1. $m' > m$. y' does not violate J, since it is at a higher level than y. For x', note that $d(y, y') = d(y, x') \geq b^m > \frac{b^m}{5}$ (or else y' could not have been inserted at level $Y_{b^{m'}}$), so that x' does not violate J.

Case 2. $m' = m$. If $d(y, y') \geq \frac{4}{5}b^m + \frac{b^m}{5} = b^m$, then neither y' or x' violate J. Otherwise, jumps J and J' are fixed by

JR1 and JR2, respectively. Call the new jumps J_f and J'_f. The packing property implies that $d(y, y') \geq b^{m-1} = \frac{b^m}{5}$, so that J_f is not violated by y' or its descendants.

Case 3. $m' < m$. Since z', the parent of y', did not violate J, the distance from z' to y must be at least $\frac{b^m}{5} + \frac{4}{5}b^{m'+1}$. By close-containment, $d(z', x') < \frac{4}{5}b^{m'+1} - b^{m'}$; thus $d(y, y') = d(y, x') \geq d(y, z') - d(z', x') > \frac{b^m}{5} + \frac{4}{5}b^{m'+1} - \frac{4}{5}b^{m'+1} + b^{m'} = \frac{b^m}{5} + b^{m'} > \frac{b^m}{5} + \frac{4}{5}b^{m'}$.

We now show that the jump rules hold with respect to the new jump J'. Note that if we show that a ball $z \in Y_{b^k}$, $k \leq m'$, does not violate J' (so that $d(z, y') \geq \frac{b^{m'}}{5} + \frac{4}{5}b^k$) then it follows from close-containment that none of its children at level Y_{b^l}, $l < k$, violate J': $\frac{b^{m'}}{5} + \frac{4}{5}b^k - \frac{4}{5}b^k + b^l = \frac{b^{m'}}{5} + b^l > \frac{b^{m'}}{5} + \frac{4}{5}b^l$. We consider two cases with respect to J':

Case 1. There exists a ball $z \in Y_{b^{m'}}$.

If $d(z, y') \geq \frac{4}{5}b^{m'} + \frac{b^{m'}}{5} = b^{m'}$, then z does not violate J', and therefore none of its descendants violate J'.

If $d(z, y') < b^{m'}$, z violates J', and J' is fixed by JR2; call the resulting jump J'_f. We show that J'_f is not violated. Note that the distance from y' to any ball $v \in Y_{b^{m'-1}}$ is at least $\frac{b^{m'}}{5}$ (or else y' could not have been inserted in $Y_{b^{m'}}$), and so v and its descendants do not violate J'_f. If there is a jump from z down to $w \in Y_{b^l}$, $l < m' - 1$, we recall that $d(y', z) \geq b^{m'-1} = \frac{b^{m'}}{5}$, and so w and its descendants do not violate J'_f.

Case 2. There exists a jump J from $y \in Y_{b^m}$, $m > m'$ down to $x \in Y_{b^k}$, $k < m'$. The jump at y implies that the distance from y to the parent of y', say z', is at least $\frac{b^m}{5} + \frac{4}{5}b^{m'+1}$. By close-containment, $d(z', y') < \frac{4}{5}b^{m'+1} - b^{m'}$; thus $d(x, y') = d(y, y') \geq d(y, z') - d(z', y') > \frac{b^m}{5} + \frac{4}{5}b^{m'+1} - \frac{4}{5}b^{m'+1} + b^{m'} = \frac{b^m}{5} + b^{m'} > \frac{b^{m'}}{5} + \frac{4}{5}b^k$, so x and its descendants do not violate J'. □

We can now prove the following theorem.

THEOREM 1. *The insertion, promotion and jump rules require $O(1)$ operations on D for a single update.*

Proof. It is clear that a single application of an insertion, promotion, or jump rule requires $O(1)$ operations on D.

Concerning the application of jump rules following an addition of a ball or jump at level Y_{b^m} (by an insertion or promotion): It follows from Invariant 1 that JR3 is applied only once. Lemma 4 demonstrates that JR2 is applied at most once to the inserted jump, and JR1 is applied at most once to jumps whose tops are in Y_{b^m} and that are violated by the newly added ball. By the packing property, there exist at most $O(1)$ such jumps.

All that remains is to reiterate that Lemma 3 shows that jump splits do not violate neighboring jumps. □

Finally, we note that the close-containment property allows links to parents, children and friends to be maintained. Consider point $x \in Y_{b^k}$ inserted as a child of its closest parent $y \in Y_{b^{k+1}}$. All parents of x can be located, since they are strictly within distance $2b^m$ of y, and so they are friends of y. Similarly, all friends of x are children of y: If there is a jump from y down to x, then there are no balls of level Y_{b^k} strictly within distance $\frac{b^m}{5} + \frac{4}{5}b^k > 2b^k$ of y and x, and so x has no friends. If x is not the bottom of a jump, then note that by close-containment the distance from y to one of x's friends is strictly less than $\frac{4}{5}b^{k+1} - b^k + 2b^k = \frac{4}{5}b^{k+1} + b^k = b^{k+1}$, so that y is a parent of this friend.

2.3 Updating the Spanning Tree

The upkeep of D's spanning tree S is straightforward. Every ball of D is represented in S. If a ball $x \in Y_{b^l}$ is created due to the splitting of a jump from $y \in Y_{b^m}$ down to $w \in Y_{b^k}$, then a corresponding ball $x \in S$ is created as a child of the ball in S corresponding to y, and a parent of the ball corresponding to w. If $x \in D$ is created as a result of a promotion or insertion of a jump, then $x \in S$ is created as a child of the ball in S corresponding to the closest parent of x in D.

3. SEARCH ALGORITHM

In this section, we describe how to execute an $O(\log n)$-time *containment search* on S. A containment search, given a point q, seeks the closest lowest level ball x in the DAG that contains q. Given x, we defer to the algorithm of [15] to find a $(1+\epsilon)$ nearest neighbor to q in $(1/\epsilon)^{O(1)}$ more steps.

To carry out the containment search, we need the DAG D of Section 2, which records the child lists and pointers to each ball's friends. We also require a spanning tree S of D. In addition, we need to maintain a *centroid path* decomposition of S as well as store each centroid path of S in a weighted search structure, as described in Section 3.1. For our purposes, we define the centroid path of a tree to be the path starting at the root, which at each node v branches to v's 'largest' child, with ties broken arbitrarily; $s(v)$, the size of a node, is simply the number of nodes in the subtree rooted at that node. We also stipulate that each centroid path has nodes with size $2^i < s(v) \leq 2^{i+1}$ for appropriate i. In a centroid path decomposition, we recursively decompose each off-path subtree of the centroid path.

Suppose for the moment that D had a well-separated property, which means that any ball of Y_{b^k} had no friends. Then we could easily execute a containment search for point q on D in $O(\log n)$ time. The search begins at the root of S, and is defined throughout by s, the current node of interest whose ball contains q, and by the centroid path C_i that contains s. The search only considers the descendants of s.

The task is to find r, the bottommost node on C_i containing q. The search first checks if the bottom node on C_i, say s_b, contains q. If it does, then the search tests each child of s_b to see if any of them contain q. If so, the containing node is the new s, and the search proceeds on its centroid path C_j. If none of them contain q, the search terminates at s_b.

If s_b does not contain q, then r is located by means of a weighted binary search driven by the search structure for the centroid path of S. At each step, the binary search tests consecutive node $s_1, s_2 \in C_i$, where s_1 is the parent of s_2; if s_1 contains q but s_2 does not, then $r = s_1$. Otherwise the search continues in the appropriate part of C_i (everything strictly above s_1, or everything strictly below). A node's weight for the binary search is given by the sum of the node sizes of its off-path subtrees, plus 1 for the node itself. Having found r, the search tests each off-path child of r to see if any of them contain q. If so, the containing node is the new s. If not, the search terminates at r.

This algorithm runs in $O(\log n)$ time. For when a child of r is the next node of interest, as we show in Section 3.1,

the centroid path structure will have used $O(\log \frac{W}{w})$ steps to remove $W - w$ nodes from consideration (where W is the number of nodes in r's centroid path plus the combined weight of the off-path subtrees rooted at these nodes, and w is the weight of the off-path subtrees rooted at r). If r was the bottom node of its centroid path, then the search will have eliminated $W - w$ nodes from consideration in a single step.

The difficulty in using this search procedure for D is that D is not well-separated. If we find that q is not contained in r we may indeed eliminate r from contention, as close-containment ensures that none of the descendants of r contain q; but if we find that q is contained in r, we cannot eliminate r's friends from consideration. This problem would appear to break the logarithmic search, but can be evaded by noting that in the navigating net search, we needed to search no more than $O(1)$ points at each level. Translated onto the search of S, this implies that only $O(1)$ subtrees need be retained.

More formally, let a search be defined by a set V of nodes of interest, which are all friends of one node $v_h \in V$. For each node $v_i \in V$, record the centroid path C_i of S in which it is found, and the current portion of C_i being searched. Let C be the set of these centroid paths.

The search begins with the root as the only member of V. At every step, the search takes the path C_i whose remaining weight is largest, and as in the ideal well-separated case tests the bottom node of C_i and then consecutive nodes of C_i. Loosely speaking, one step of the binary search on C_i reduces the weight of the current portion of C_i by a constant fraction. (A more precise statement is found in Lemma 6 below.) When, for some i, a node r_i is found which contains q, the search from r_i continues as follows. All off-path children of r_i are tested to determine if they contain q. Let s be a child of r_i containing q, if any. Then s and its friends form the new set of nodes of interest. If there is no such node s, r_i's q-containing friends form the new set of q-containing nodes of interest. If this set is empty, the search stops at r_i.

Lemma 5 below implies that the new set of the nodes of interest are descendants of the surviving portions of the paths in C. It follows that the centroid path C_i containing one of these nodes is either the same as a $C_j \in C$, and so for the purposes of the search its weight is the same as the remaining weight of C_j, or it is a descendant of some path $C_j \in C$, in which case its weight is less than the weight of the remaining portion of C_i. Lemma 6 implies that if it took d steps to find the new nodes of interest, the weight of the heaviest path is reduced by a factor of at least $2^{\Theta(d)}$. It follows that:

THEOREM 2. *The above search procedure on S terminates in $O(\log n)$ time.*

It is left to show that taking the friends of r_i as the nodes of interest suffices. To this end, we have the following lemma.

LEMMA 5. *If the query point q is contained in a ball $y \in Y_{b^m}$, then any lower level ball $v \in Y_{b^k}$ that contains q is a descendant of y or its friends.*

Proof. Note that as y and v both contain q, $d(y, v) \leq b^m + b^k$.

We first argue that v must have an ancestor in Y_{b^m}, with proof by contradiction. Suppose that among the ancestors of v there is a jump from $z \in Y_{b^p}$ $(p \geq m+1)$ to $x \in Y_{b^l}$ $(l \leq m-1)$; note that $c(z) = c(x)$. By close-containment, we have that $d(v, x) = d(v, z) < \frac{4}{5}b^l - b^k$. It follows that $d(y, z) \leq d(y, v) + d(v, z) < b^m + b^k + \frac{4}{5}b^l - b^k = b^m + \frac{4}{5}b^l < \frac{1}{5}b^p + \frac{4}{5}b^m$; thus y violates the jump down from z.

It follows that v has an ancestor $w \in Y_{b^m}$. If w is not y, we show that w and y are friends, that is $d(w, y) \leq 2b^m$; for by close-containment, $d(v, w) < \frac{4}{5}b^m - b^k$, so that $d(y, w) \leq d(y, v) + d(v, w) \leq b^m + b^k + \frac{4}{5}b^m - b^k = \frac{9}{5}b^m < 2b^m$. □

3.1 Centroid Path Updates

S will change, but only locally, as insertions to D occur. Specifically, S may gain leaves or internal nodes. The effect of these changes on the centroid paths is limited.

A new node of weight one may be inserted in a path, a node on a path may have its weight increase by 1, and a node may leave the top of one path and join the bottom of another path one level above. We use a modified biased skip list [2] to store a centroid path and support these operations as follows. Left to right order in the skip list will correspond to bottom to top order in the centroid path.

LEMMA 6. *The modified biased skip list stores an ordered set S and supports the operations search, reweigh, insert, and delete with running times specified in terms of the following parameters. W_b and W_a are the total weights of S before and after the operation, and $W = max\{W_a, W_b\}$. The cost of a search for an item of weight w is $O(\log \frac{W}{w})$, the cost of reweighting an item from w_a to w_b is $O(\log \frac{W}{w_m})$ (where $w_m = min\{w_a, w_b\}$), the cost of inserting or deleting an item of weight 1 is $O(\log W)$, and the cost of adding or removing an item of weight w at either end is $O(\log w)$.*

In Section 3.2 we review the biased skip list and describe the modifications needed to support the end updates.

LEMMA 7. *The reweightings due to a node insertion in S take $O(\log n)$ time.*

After removing a node of weight w from the top of a centroid path C, there will be at least w insertions into the subtree rooted at C's head before another node is removed from its head. Thus we can afford $\Theta(w)$ steps to handle the node transfers, while $\Theta(\log w)$ suffices. Note that an update taking $O(\log w)$ time will be performed over the next $\log w$ insertions to the relevant subtree of S.

The one concern is to ensure that the search time is not affected when the node transfer is proceeding. But this presents no real difficulty. We can, for example, keep the node "between" paths while the updates to the centroid paths are being computed. The key point is to ensure that a search on a centroid path of weight W which terminates at a node of weight w performs $O(\log \frac{W}{w} + 1)$ queries. But the "between" path nodes induce at most 2 more queries which could be performed first, without affecting the asymptotic running time.

Comment. An attempt to use a topology tree [11] for our construction encounters two obstacles in the execution of the search: (i) The transfer to friends. This can be solved by using the (very intricate) dynamic lca query structure of Cole and Hariharan [10]. (ii) Determining which balls

have real descendants (see Section 4); it is not clear if this difficulty can be solved.

3.2 Modified Biased Skip Lists

We augment the biased skip lists of Bagchi et al. [2] to allow fast insertions and deletions at either end (*end updates*), in addition to the fast searches and reweightings described in [2]. We perform an end update by a weight w item in $O(\log w)$ time. As in Lemma 6, let W_b and W_a be the total weights of the items before and after an update, and $W = \max\{W_a, W_b\}$. As in [2], a fast search takes time $O(\log \frac{W}{w})$, where w is the weight of the sought item, and a reweighting takes time $O(\log \frac{W}{w_m})$, where w_a and w_b are the old and new weights of the item being reweighted, and $w_m = \min\{w_a, w_b\}$.

Recall that in a biased skip list the items are kept in sorted order in a doubly linked list. Each item e has an integer height $h(e)$ with $h(e) \geq \log w(e)$, where $w(e)$ is e's weight. We will say that e *oversize* if $h(e) > \lceil \log w(e) \rceil$. Further, for each height h, items of height at least h are kept in sorted order in a doubly linked list, call the h-list. The skip list is parameterized by two integer constants a, b with $1 < a \leq \lfloor b/2 \rfloor$.

Let h_{max} denote the maximum height of any item. The (a, b)-skip list obeys the following invariants.

INVARIANT 3. *For all h, $0 \leq h \leq h_{max}$, there are at most b consecutive items of height exactly h in the h-list.*

INVARIANT 4. *For each oversize item e and each h such that $\lceil \log w(e) \rceil < h \leq h(e)$ there are at least a items of height exactly $h-1$ between e and its h-list neighbor, if any. This sequence is called an oversize $(h-1)$-sequence.*

Note that if the leftmost item of height h is not oversize, there is no lower bound on the length of the leftmost $(h-1)$-sequence; a similar remark applies to the right end.

DEFINITION 1. *The sequence of consecutive items of height h in the h-list starting at the leftmost item, if non-empty, is called the left end h-sequence. The right end h-sequence is defined analogously.*

The operations search, insert, delete, reweight are all described in [2]. The operation insert(e) entails adding e at the correct location and then restoring Invariants 3 and 4 as needed by a sequence of "splits" and "joins." The delete and reweigh operations are analogous. We refer the reader to [2] for details.

Loosely speaking, [2] achieve time $O(\log \frac{W}{w})$ for an insert or delete. But we want time $O(\log w)$ for an end update. To enable this we weaken Invariants 3 and 4 as follows.

INVARIANT 5. *The sequence of consecutive height h items in the h-list starting at the leftmost item, if any, contains at most $b+1$ items, and similarly for the right end.*

INVARIANT 6. *The oversize h-sequence of consecutive height h items in the h-list starting at the leftmost item, if any, contains at least $a-1$ items, and similarly for the right end.*

Before discussing the end updates, we outline the splitting and joining rules. A split takes an h-sequence of $b+1$ or $b+2$ items and creates sequences of $\lfloor b/2 \rfloor$ and $\lceil b/2 \rceil$, or of $\lfloor (b+1)/2 \rfloor$ and $\lceil (b+1)/2 \rceil$ items, respectively. A join takes h-sequences of $a-1$ or $a-2$ items and of $a+c$ items and produces an h-sequence of $2a+c$ or $2a+c-1$ items (capturing the separating item). A join immediately followed by a split takes the same pair as in the previous sentence and produces h-sequences of $\lfloor (2a+c-2)/2 \rfloor$ and $\lceil (2a+c-2)/2 \rceil$ items, or of $\lfloor (2a+c-1)/2 \rfloor$ and $\lceil (2a+c-1)/2 \rceil$ items, respectively. We need that the sizes of the new h-sequences lie in the range $[a+1, b-1]$. Choosing $a=3$, $b=10$ works.

We describe the update procedure for left end sequences; the procedure for right end sequences is entirely analogous. But first we need to specify some more constraints observed by the end h-sequences. A left end h-sequence σ can occupy one of five states: -2,-1,0,1,2. It is in state -2 if σ is oversize and holds $a-1$ items, in state -1 if it is oversize and holds a items, in state 1 if it holds b items, in state 2 if it holds $b+1$ items, and in state 0 otherwise.

We keep two stacks, one for each end of the list. The left end stack stores pointers to the left end h-sequences in state ± 2 with the stack order from bottom to top corresponding to decreasing index order. The right end stacks are organized analogously.

When an item is added to the end h-sequence in state 2, the sequence splits, putting the end h-sequence in state 0 and adding one item to the end $(h+1)$-sequence. Adding an item to a state c end sequence, $c < 2$, creates a state $c+1$ end sequence, unless $c = 0$, when the updated end sequence may also be a state 0 sequence. Removing an item from the end h-sequence in state -2 is a little more delicate. Note that the sequence is immediately followed on the right by an oversize item e of height $h+1$. e's height is reduced to h. The two leftmost h-sequences are then joined and if need be immediately split; in the former case this creates a state 0 end h-sequence and reduces the index of the end $(h+1)$-sequence, if any, by 1 or if it was in state 0 may leave its index unchanged; if there is a split the result is a state 0 end h-sequence and one more item in the left end $(h+1)$-sequence which therefore returns to its previous state.

At the ends of the skip list, the ranks of successive h-sequences are necessarily strictly increasing, but may not include intermediate indices. We refer to the indices which are not present as *missing*.

The following state distribution invariant ensures that there is no need for a cascading series of joins or splits when doing updates.

INVARIANT 7. *(i) Consecutive state 2 sequences on the stack are separated by at least one c state sequence, $c \leq 0$, which can be a missing empty sequence.*

(ii) The state 2 sequence with the lowest index is preceded by a possibly missing state c sequence, $c \leq 0$, of yet lower index.

Analogous rules apply to state -2 sequences, except that missing sequences are not included.

To insert an item of weight w at the left end, clear all h-sequences in states 2 and -2 for $h \leq \lceil \log w \rceil$ plus one more from the left end stack by performing appropriate joins and splits, and then add w with height $\lceil \log w \rceil$ to the left end of the skip list.

To delete the rightmost item, remove the bottom sequence on the right end stack, if it is in state -2, by splits and joins, and then remove the rightmost item. If this creates a state -2 sequence, then fix it by a join (and if need be by a follow up split).

LEMMA 8. *The procedure maintains Invariants 3 – 7.*

Proof. We begin by considering the effect of an insertion. Consider the states of the left end h-sequences in increasing index order, including missing (empty) h-sequences. (For simplicity, instead of having state values $c \leq 0$, we simply show the value zero, but it is not hard to see the argument continues to hold.) Such a state series has the form $\ldots 01 \ldots 1201 \ldots 12$ or $\ldots 01 \ldots 121 \ldots 101 \ldots 12$. The insertion procedure repeatedly removes the leftmost 2 state, creating a new series of the form $\ldots 01 \ldots 1011 \ldots 12$ or $\ldots 01 \ldots 1021 \ldots 101 \ldots 12$; the new series continues to satisfy Invariant 7. The final step of the insertion is to add the new item. The last state 2 sequence σ to be split, if any, had index greater than that of the inserted item. σ, if split, was replaced by a state 0 sequence σ'. As the newly inserted item has lower index than state 0 σ', Invariant 7(i) is maintained. Maintaining Invariant 7(ii) is a concern only if the newly inserted item creates a state 2 sequence τ, necessarily of lowest index; in this event, splitting τ maintains both Invariant 7(i) and (ii).

Analogous arguments apply for state -2. □

Comment. We conjecture that essentially this construction can be applied to the biased search trees of Bent et al. [3]

The next issue to mention is what happens when a reweighting or non-end update performs a split or join on an end sequence. This is performed as usual, with the corresponding sequence being removed from the stack. Invariant 7 continues to hold, as in the proof of Lemma 8.

THEOREM 3. *The modified biased skip list structure stores a set of weighted ordered items supporting searches in* $O(\log \frac{W}{w})$ *time, reweightings in* $O(\log \frac{W}{w_m})$ *time, and end updates in* $O(\log w)$ *time.*

4. DELETIONS

When a point is deleted, its corresponding leaf ball in Y_0 is marked, but no other changes are made to D. This scheme results in two issues that need to be addressed: (i) After multiple deletions, there may be b balls in the DAG but $o(b)$ points, and (ii) the containment search may return a ball that is not the ancestor of a real point. The first concern is addressed by rebuilding the data structure in the background. (It suffices, if D stores n points including deleted points, to start rebuilding after $\frac{n}{3}$ deletions, and to complete the rebuilding over the next $\frac{n}{6}$ insertions and deletions; i.e. for each update, perform 7 updates on the background structure. D will then contain at least $\frac{n}{2}$ points including at most $\frac{n}{6}$ deleted points.)

To address the second problem, we devise a scheme for returning only balls which are ancestors of real points. For each centroid path, we keep track of whether its top point has any real descendants; if so, we also record if there are real descendants in the subtrees of the bottommost node of the current centroid path, and we record which (if any) of the path's vertices have real points in their off-path subtrees. Call these vertices *real vertices*. For each centroid path, the real vertices are kept in a standard balanced tree. Note that the leftmost node of this balanced tree stores the bottommost real vertex of the centroid path.

Note that if a path loses some (but not all) of its real descendants, then each ancestral path continues to have the same real vertices, and therefore its balanced tree remains unchanged. Thus, the effect of deleting a point is to reduce a (possibly empty) series of paths from having one real vertex to having none, and for the next path up to decrease its number of real vertices to some nonzero number. All paths further up are unaffected. The cost of the updates to the associated balanced trees is $O(1)$ for each of $O(\log n)$ paths in the series, and $O(\log n)$ for the topmost path, for a total of $O(\log n)$.

A containment search for insertions proceeds as before, irrespective of which vertices are ancestors of real points. For a containment search for an ANN search, we wish to return only balls that are ancestors of real points. To this end, when the search algorithm considers a centroid path, we seek a ball on this path that has real descendants in addition to containing the query point. When the search defers to the algorithm of [15], we again only consider balls with real descendants. We can determine in $O(1)$ time whether a ball has real descendants, by locating its centroid path and the lowest node of that path that contains real descendants, and such a real descendant can be identified in $O(1)$ time by keeping a pointer to it at the top of the centroid path. It is easy to maintain this pointer as updates occur.

5. ACHIEVING TRULY LINEAR SIZE.

In this section, we discuss how our structure can be implemented in space *truly linear* in the number of points. That is, we show how to construct a search structure whose size is $O(n)$, where the notation does not hide constants that depend on the doubling dimension. (The notation does hide the cost of storing a point.) In the following discussion, by a 'constant' we mean a value independent of the doubling dimension.

We first focus on the DAG of Section 2. We prove that the aforementioned rules for insertions, promotions, and jump fixes already ensure that the number of balls in the DAG is truly linear. We then reduce the number of friends lists that need to be maintained, while changing the search time by only an additive factor.

5.1 Linear Number of Balls in the DAG

In this section we demonstrate that the rules for the construction of the DAG already guarantee that D is of truly linear size.

Recall that the insertion and promotion rules require a constant number of operations; hence, a newly added point adds only a constant number of balls in the DAG due to the insertion or promotion rules. Likewise, a jump fix requires the addition of a single ball, however a ball addition may violate multiple neighboring jumps, and necessitate multiple jump fixes; unfortunately, the resulting bound on the number of balls created will depend on the doubling dimension. Therefore, it is necessary and sufficient to demonstrate that although jump fixes may cause multiple balls to be added to the DAG after a single point insertion, the total number of balls in the DAG remains truly linear. We shall accomplish this with a simple charging argument.

Before we present the charging argument, note that by Lemma 4 a newly added ball $x \in Y_{b^m}$ only violates jumps whose tops are in Y_{b^m}. Suppose that a newly added ball x violated multiple such jumps; let $\mathcal{J}$ be the set of these jumps, and let B be the set of the tops of these jumps, with $y_i \in B$ being the top of jump $J_i \in \mathcal{J}$. Since x violates each

jump of $\mathcal{J}$, we have that $d(x, y_i) < b^m$ and $d(y_i, y_j) < 2b^m$ for all $y_i, y_j \in B$. We can prove the following lemma:

LEMMA 9. *Suppose a ball $y_i \in B$ has the same center as some ball $v \in Y_{b^{m+r}}$, for $r \geq 2$. Then no other ball of B has the same center as* any *ball $w \in Y_{b^{m+q}}$, for $q \geq 2$.*

Proof. First note that the promotion and jump split rules ensure that v is an ancestor of y_i. Now, suppose by contradiction that y_i has such an ancestor v and some y_j has such an ancestor w. If $q = r$, then we have that $d(v, w) = d(y_i, y_j) < 2b^m < \frac{b^{m+r}}{5}$, and the packing property is violated, which is a contradiction. Otherwise, $r \neq q$; we assume without loss of generality that $r > q$, and that there is a jump from v down to $Y_{b^{m+p}}$, $p < q$. Since $d(v, w) = d(y_i, y_j) < 2b^m = \frac{2}{5}b^{m+1} < \frac{1}{5}b^{m+r} + \frac{4}{5}b^{m+q}$, the jump is violated, which is a contradiction. It follows that at most one ball of B can have an ancestor in $Y_{b^{m+r}}$, for $r \geq 2$. □

We now present the charging argument. Let $y_j \in B$ be the only ball with an ancestor in $Y_{b^{m+r}}$, $s \geq 2$, if any. Charge each point $p_i = c(y_i)$, $i \neq j$ for the jump fix of $J_i \in \mathcal{J}$, but charge the jump fix of J_j to the newly added ball that led to the jump fix.

LEMMA 10. *Each point can be charged for only a constant number of jump fixes.*

Proof. When a point is inserted, a leaf ball $s \in Y_0$ representing the new point is added to the DAG. The insertion rules provide that along with a new jump from $t \in Y_{b^m}$ down to s, a constant number of balls may be inserted into the DAG. Each one of these balls can violate multiple jumps, but the charging rules stipulate that for each ball b, $c(s)$ may be charged for only one violated jump (namely, the jump whose top has an ancestor at least two levels above b). Hence, when s is inserted, $c(s)$ is only charged for a constant number of jump fixes.

Subsequent to the insertion of s, the jump from t to s may be violated by further point insertions. However, $c(s)$ can only be charged for the creation of $\langle c(s), Y_{b^{m-1}} \rangle$ and $\langle c(s), Y_{b^{m-2}} \rangle$. Any further fixes to this jump create lower balls which are charged to the newly inserted violating points. □

5.2 Efficient Storage of Friends Lists

Although we have shown that the number of balls in the DAG is truly linear, we have stipulated that each ball store a list of its friends, and in general this cannot be done if one is allotted only truly linear space. To this end, we relax the condition that each ball store its friends. Instead, only some balls will store all their friends.

Suppose that we knew λ, the doubling dimension of the space. We know then the maximum number of friends that any ball may possess; this quantity is $2^{O(\lambda)}$, or $2^{\lambda'}$ for brevity. If we keep friends lists in only $n/2^{\lambda'}$ balls of the DAG, then the resulting structure uses truly linear space. Since there are $O(n/2^{\lambda})$ balls at height λ or greater in the modified biased skip list, it suffices to allow only these balls to store friends lists.

As we do not know λ', instead we limit the length of a ball's friends list according to its height in the skip list; a ball at height $2i$ or $2i + 1$ will be allowed to store a list of length at most 2^i. Any ball that has more friends than can fit in its list is marked as *incomplete*. Again, this uses truly linear space.

The search proceeds as before, until we reach a node which is marked incomplete. In this scenario, we backtrack in the biased skip list until we reach a level at which all the nodes being examined store complete friends lists; that is, at most up to height $2\lambda'$. We then proceed with the containment search in a "brute force" manner in each subtree of interest, as in [15]. The search time increases by an additive term of $2^{O(\lambda)}$. We conclude that:

THEOREM 4. *The data structure may be implemented to require only truly linear space.*

6. ACKNOWLEDGMENTS

We thank Gerth Brodal and Bob Tarjan for helpful pointers to the literature on the dynamic weighted search problem. We also thank Sariel Har-Peled for comments on an earlier version of this paper, and both Har-Peled and Piotr Indyk for pointers to related literature.

7. REFERENCES

[1] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM*, 45(6):891–923, 1998.

[2] A. Bagchi, A. L. Buchsbaum and M. T. Goodrich. Biased skip lists. *Algorithmica*, 42(1):31–48, 2005.

[3] S. W. Bent, D. D. Sleator and R. E. Tarjan. Biased search trees. *SIAM J. Comput.*, 14(3):545–68, 1985.

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

[5] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. Manuscript.

[6] P. B. Callahan and S. R. Kosaraju. A decomposition of multi-dimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *ACM Symposium on Theory of Computing*, 1992, 546–556.

[7] E. Chavez, G. Navarro, R. Baeza-Yates, and J. L. Marroquin. Proximity searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.

[8] K. L. Clarkson. A randomized algorithm for closest-point queries. *SIAM J. Comput.*, 17:830–847, 1988.

[9] K. L. Clarkson. Nearest neighbor queries in metric spaces. *Discrete Computational Geometry*, 22(1):63–93, 1999.

[10] R. Cole and R. Hariharan. Dynamic lca queries. *SIAM J. on Comput.*, 34(4):894–923, 2005.

[11] G. N. Frederickson. A data structure for dynamically maintaining rooted trees. *J. Alg.*, 24(1):37–65, 1997.

[12] A. Gupta, R. Krauthgamer, and J. R. Lee. Bounded geometries, fractals, and low-distortion embeddings. *IEEE Symposium on Foundations of Computer Science*, 2003, 534–543.

[13] S. Har-Peled and M. Mendel. Fast construction of nets in low dimensional metrics, and their applications. *ACM Symposium on Computational Geometry*, 2005, 150–158.

[14] D. Karger and M. Ruhl. Finding nearest neighbors in growth-restricted metrics. *ACM Symposium on Theory of Computing*, 2002, 63–66.

[15] R. Krauthgamer and J. R. Lee. Navigating nets: simple algorithms for proximity search. *ACM-SIAM Symposium on Discrete Algorithms*, 2004, 798–807.

[16] R. Krauthgamer and J. R. Lee. The black-box complexity of nearest neighbor search. *Theoretical Computer Science*, 348(2–3):262–276, 2005.