A Linear Time Euclidean Spanner on Imprecise Points

Jiemin Zeng* Jie Gao†

Abstract

An s-spanner on a set S of n points in $\mathbb{R}^d$ is a graph on S where for every two points $p, q \in S$, there exists a path between them in G whose length is less than or equal to $s \cdot |pq|$ where $|pq|$ is the Euclidean distance between p and q. In this paper, we consider the construction of a Euclidean spanner for imprecise points where we take advantage of prior, inexact knowledge of our input. In particular, in the first phase, we preprocess n d-dimensional balls with radius r that are approximations of the input points with the guarantee that each input point lies within its respective ball. In the second phase, the specific points are revealed and we quickly compute a spanner using data from the preprocessing phase. We can compute (or update) the $(1+\varepsilon)$-spanner in time $O(n(r+\frac{1}{\varepsilon})^d \log(r+\frac{1}{\varepsilon}))$ after $O(n(r+\frac{1}{\varepsilon})^d \log \alpha)$ preprocessing time where α is the ratio between the furthest and the closest pair of points. Our algorithm does not have any restrictions on the distribution of the points. It is the first such algorithm with linear (update) running time.

1 Introduction

While in classical computational geometry all input values are assumed to be accurate, in the real world this assumption does not always hold. Aside from measurement errors or human errors that are always inherent to any data, there are a few scenarios in emerging applications in which only approximate locations are known and from time to time more accurate positions are given. GPS units are generally power hungry and are typically not kept on all the time to save energy. It is common practice to only turn on GPS periodically and, in between the snapshots, interpolate or extrapolate the device's positions based on the speed of the user. In another application, a user's location data might be perturbed in an attempt to protect user privacy. Here, a random nearby location, instead of the user's true location, is reported to location based services, as in [20]. These are the major motivating applications for this paper.

*Department of Computer Science, Stony Brook University, jiezeng@cs.stonybrook.edu

†Department of Computer Science, Stony Brook University, jgao@cs.stonybrook.edu

Models on Imprecise Points The ways to model imprecise input can vary greatly based on the interpretation of the nature of uncertainty the data may contain. There are models that are tailored to address input data imprecision or limitations of data representation, where the exact values may never be known [11, 13, 15, 18]. Other approaches involve computing the boundary cases based on the bounds of the input [15] and computing a distribution of solutions when given a probability distribution of the input set [13]. Yet another case assumes a superset of the input is known [4].

A popular model assumes that some knowledge of the location of each input point is known ahead of time. Initially, instead of a point, a region where each input point may lie is known. The shape (lines [8], circles/balls [1, 6, 12, 14], fat regions [3, 19], etc.) of the region can vary as well as constraints such as the amount of overlapping regions allowed. Algorithms that operate in this model have two phases. The first is a preprocessing phase where the input regions are processed in such a way as to aid the second phase. The second phase, also called the query phase, occurs when the precise locations of the input points are known and the result is computed usually using some structure from the preprocessing phase.

Spanners and Proximity Queries In this paper we are interested in the maintenance of a Euclidean spanner and proximity queries using the spanner. Given a set P of n points in $\mathbb{R}^d$, a Euclidean $(1+\varepsilon)$-spanner defines a graph G (often a sparse graph with a linear number of edges) on the points, each edge weighted by the Euclidean length, such that the shortest path between any two points u, v in the graph is at most $1+\varepsilon$ times the Euclidean distance of u, v. Thus one can consider the spanner as a sparse backbone that approximates all of the pairwise distances. For this reason, a spanner can be used to answer many proximity queries such as closest pair, approximate nearest neighbor (given a point q, find a point $p \in P$ that is at most $(1+\varepsilon) \cdot d$ away from q, where d is the distance from q to its closest point in P), and approximate clustering.

Spanners have been studied for many years. Numerous algorithms have been developed to compute a spanner, see the survey [7] or the recent book [16]. Almost all previous work assume that the precise positions of the points are known. We present our results below

and compare with the only previous work on spanners for imprecise inputs.

Our Results In the model our algorithm operates in, an imprecise point p is defined to be a disk or a d-dimensional ball centered at a point $\dot{p}$ with radius r. The assumption is that initially, the only information of an input point is $\dot{p}$ and r and that later, the precise location of the point $\hat{p}$ (located somewhere within the corresponding ball) is revealed. Our algorithm preprocesses a set $S = \{p_1, p_2, \ldots, p_n\}$ of n imprecise points in $O(n(r + \frac{1}{\varepsilon})^d \log \alpha)$ time such that when $\hat{S} = \{\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n\}$ is available, we can compute a $(1+\varepsilon)$-spanner in $O(n(r + \frac{1}{\varepsilon})^d \log(r + \frac{1}{\varepsilon}))$ time where d is the dimension of the input. Without loss of generality we scale our input such that the distance between the closest pair of centers is 1 and thus α is the diameter of the centers of the imprecise points. It is important to note that our algorithm will accept input sets with overlapping points of any depth. In the applications that we are interested in, r is typically a small constant.

Related Work Previous work on algorithms for imprecise points can be classified by objective (convex hull [8], triangulation [12, 19], Delaunay triangulation [3, 6, 14], etc) and shape of imprecise regions (lines [8], circles/balls [1, 6, 12, 14], fat regions [3, 19], etc.). Most of the algorithms aim for linear or almost linear running time for the second phase, when the precise locations are given. Notice that a linear running time is the best we can hope for, as one would need linear time to simply scan the input.

Our construction algorithm modifies a dynamic spanner which has been well studied before [2, 9, 10, 17]. However, since previous work focuses on optimizing the insertion/deletion of single points, the time needed to update the entire spanner for all imprecise points is superlinear.

The only known previous work for spanners on imprecise points is done by Abam et. al. [1]. They provided an algorithm to construct a $(1 + \varepsilon)$-spanner from a set of n imprecise points in $O(n/\varepsilon^d)$ time after $O(n \log n)$ preprocessing time, in which the imprecise points are assumed to stay in n *pairwise disjoint* unit balls in d-space. In the preprocessing phase, the well-separated pair decomposition of the centers of the balls is computed, in time $O(n \log n)$ [5]. When the specific points are known, the spanner is constructed by creating an edge between every pair of well-separated sets. Since there are $O(n/\varepsilon^d)$ edges in the spanner, this phase takes $O(n/\varepsilon^d)$ time. They also provide a variation for inputs of different sizes, however this version runs in $O(n \log n/\varepsilon^{2d})$ time in the second phase after $O(n \log n)$ preprocessing.

In comparison with our results, we remark that the above algorithm requires that the imprecise regions are disjoint. If the closest pair of centers is defined to have distance 1, the input requires r to be at most $1/2$. The version with varying sized imprecise regions allows higher values of r but the update cost has an extra d on the exponent. Our algorithm accepts imprecise points with *overlapping* regions and allows the user a degree of control over the running time by trading efficiency with the size of the imprecise region.

2 Background

The algorithm we present constructs a $(1 + \varepsilon)$-spanner or more specifically the deformable spanner (DefSpanner) as described in [9]. A DEFSPANNER is a specific $(1+\varepsilon)$-spanner construction that is designed to be easily modified and updated. More specifically, for a set of points S in $\mathbb{R}^d$, the DEFSPANNER G is made up of a hierarchy of levels $G_{\lceil \log_2 \alpha \rceil} \subseteq G_{\lceil \log_2 \alpha \rceil - 1} \subseteq \cdots \subseteq G_i \subseteq G_{i-1} \subseteq \cdots \subseteq G_0 = S$ where the top level $G_{\lceil \log_2 \alpha \rceil}$ contains only one point and the bottom level G_0 contains all points in S.

The aspect ratio α of S is defined as the ratio of the distance between the furthest and closest pair of points in S. Since we scale all values such that the distance between the closest pair of imprecise centers is 1, α is also the diameter. Note that r is scaled as well. There is no restriction for the distance between the closest pair of precise points.

Any set G_i is a maximal subset of G_{i-1} where for any two points $p, q \in G_i$, $|pq| \geq 2^i$. Let $p^{(i)}$ denote the node p in level i given that $p \in G_i$. We say a point $p^{(i)}$ covers a point $q^{(i-1)}$ if $|pq| \leq 2^i$. While a point $q^{(i-1)}$ may be qualified to be covered by several points in the level above, we arbitrarily designate one of these points (say $p^{(i)}$) as its parent by $p^{(i)} = P(q^{(i-1)})$ and say that $q^{(i-1)}$ is the child of $p^{(i)}$. The exception is if $q \in G_i$, then $P(q^{(i-1)}) = q^{(i)}$. We denote $P^k(p^{(i)}) = P(P^{k-1}(p^{(i)}))$ to be the ancestor of $p^{(i)}$ at k levels higher and define $P^0(p^{(i)}) = p^{(i)}$. When p has no superscript, we assume that $P(p) \neq p$ or intuitively, that p is in the highest level it resides in. We also denote $C_{i-1}(p) = \{q \in G_{i-1} | P(q) = p\}$ as the set of children of $p^{(i)}$ in level $i-1$. By a packing argument (shown in [9]), $|C_{i-1}(p)| \leq 5^d$. Note that for any point p, $|P^i(p^{(0)})p| \leq 2^{i+1}$. See Figure 1(a) for an illustration of these rules.

The edges of a DEFSPANNER of a set S are determined by connecting all nodes within distance $c \cdot 2^i$ in all levels, where i is the level and c is a constant. Such pairs are called neighbors at level i. The neighbors of p at level i are denoted by $N_i(p)$. As shown in [9], $|N_i(p)| \leq (1 + 2c)^d - 1$. The total number of edges in a DEFSPANNER is less than $2(1 + 2c)^d n$. Note that two nodes can be neighbors in multiple levels and that an edge is always constructed between every parent-child

pair. The family tree metaphor is extended to define cousins as two points whose parents are neighbors. If two points are neighbors, they are also cousins as their parents must be neighbors as well.

A path between two nodes in a DefSpanner that is less than or equal to $1+\epsilon$ times their euclidean distance can be found by traversing up the hiearachy from both nodes until a mutual link is found. This is illustrated in Figure 1(b). We refer to [9] for a proof and more details.

Now we state a property of a DefSpanner that will be useful later.

Lemma 1 *For a node $p \in G_i$, let $q \in G_{i+1}$ be another node such that $|pq| \leq (c-1)2^{i+1}$. The nodes $P^j(p^{(i)})$ and $P^{j-1}(q^{(i+1)})$ must be neighbors (or the same node) for all $j \geq 1$, i.e., the ancestors of p and q must be neighbors (or converge) in all levels above i.*

Proof. It is given that $|pq| \leq (c-1)2^{i+1}$. By the definition of a DefSpanner, $|P^j(p^{(i)})p| \leq 2^{i+j+1} - 2^{i+1}$ and $|P^{j-1}(q^{(i+1)})q| \leq 2^{i+j+1} - 2^{i+2}$. Thus, $|P^j(p^{(i)})P^{j-1}(q^{(i+1)})| \leq |P^j(p^{(i)})p| + |pq| + |P^{j-1}(q^{(i+1)})q| \leq 2^{i+j+1} - 2^{i+1} + (c-1)2^{i+1} + 2^{i+j+1} - 2^{i+2} = 4{\cdot}2^{i+j} + (c-4)2^{i+1} \leq 4{\cdot}2^{i+j} + (c-4)2^{i+j} = c{\cdot}2^{i+j}$. Therefore $P^j(p^{(i)})$ and $P^{j-1}(q^{(i+1)})$ must be neighbors or the same node in level $i+j$. □

3 Algorithm Description

Let $S = \{p_1, p_2, \ldots, p_n\}$ be a set of imprecise points where for p_i, $\dot{p}_i$ is the center, r is the radius, and $\hat{p}_i$ is an instance of p_i, i.e., $|\dot{p}_i\hat{p}_i| \leq r$. Let $\dot{S} = \{\dot{p}_1, \dot{p}_2, \ldots, \dot{p}_n\}$ and $\hat{S} = \{\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n\}$.

3.1 Preprocessing

In the preprocessing phase, we are initially given $\dot{S}$ and r. We create a DefSpanner $\dot{G}$ with the point set $\dot{S}$ as in [9]. Here we choose $c = 2r + \frac{16}{\varepsilon} + 4$. The running time is $O(n(r + \frac{1}{\varepsilon})^d \log \alpha)$.

3.2 Spanner Construction

When the true positions of the points are revealed, we will update the spanner. We construct a DefSpanner $\hat{G}$ for $\hat{S}$ by inserting the points one by one into $\hat{G}$. That is, $\hat{G}$ is initially empty and we gradually create the hierarchy by expanding downwards.

An ordered list is maintained of the remaining points yet to be inserted into the spanner. When we begin, this list contains all of the nodes in the order they appear in the hierarchy in $\dot{G}$, from highest to lowest. So all nodes in level i are placed right before all nodes in level $i-1$. Parents in $\dot{G}$ will always appear before their children. After all nodes in level i have been inserted, we create level $i-1$ before adding the next node. Here each node at level i is repeated at level $i-1$ with itself as the parent. As we continue to insert nodes into $\hat{G}$, we may add levels to $\hat{G}$ that are beyond the range of levels in $\dot{G}$ – if the exact locations are closer than 1 for instance. During the insertion process for a node, we may find that it is necessary to move the node up or down the hierarchy. If a node is demoted to l, a lower level, l does not exist yet. Instead of creating more levels immediately, the point is removed from $\hat{G}$ and is reinserted into the list of remaining points right before all the other nodes in level l. Therefore, we wait until all levels above l are filled before inserting this node.

The algorithm terminates when all points are inserted. In some sense we are rebuilding the hierarchy for $\hat{G}$ by referring to the hierarchy of $\dot{G}$. We would like to reuse the structure of $\dot{G}$ as much as possible. We elaborate the procedure below.

We assume that $\hat{p}$ is at the beginning of the list, and we want to insert $\hat{p}$ into $\hat{G}$. We insert $\hat{p}$ into the bottom level of $\hat{G}$. Let's call this level i. There are three phases in our algorithm as described below.

Step 1: Check for demotions. First, we compare the distances between $\hat{p}$ and all of its neighbors in $\dot{G}$ that have been inserted in $\hat{G}$. If $|\hat{p}\hat{q}| < 2^i$, where $\hat{q}$ is a neighbor of $\hat{p}$, then $\hat{p}$ is too close to $\hat{q}$ and violates one of the properties of a DefSpanner. Therefore, we must move $\hat{p}$ to a lower level. We can calculate this level to be l, where $2^l \leq |\hat{p}\hat{q}| < 2^{l+1}$ and $\hat{q}$ is the closest node to $\hat{p}$. As discussed earlier, this level does not exist yet, so we must delay the insertion of $\hat{p}$. Meanwhile, we denote $c(\hat{p}) = \hat{q}$, the potential parent of $\hat{p}$.

When we try to reinsert a demoted node such as $\hat{p}$, we still need to check that it does not need to be demoted again. Since demoted nodes are reinserted into $\hat{G}$ before other nodes in the same level, a node can only be demoted by another previously demoted node. Therefore, we only need to test the previously demoted cousins of $\hat{p}$ (through $c(\hat{p})$) for possible demotion. We can strategically place pointers to keep track of these previously demoted nodes.

Step 2: Find a parent. Now we determine the parent of $\hat{p}$. If $c(\hat{p})$ has been defined for $\hat{p}$, then we assign $P(\hat{p}) = c(\hat{p})$. As mentioned in step 1, in this case $\hat{p}$ was demoted from a higher level and is now reinserted into level i so $|\hat{p}c(\hat{p})| \leq 2^{i+1}$. If $c(\hat{p})$ is not defined, we need to search for the parent of $\hat{p}$. First we define the point $u_{\hat{p}}$, the starting point for parent search, and prove a property of it. For this purpose we refer to the spanner $\dot{G}$. Let $u_{\hat{p}} = \hat{q}$ where $\dot{q} = P(\dot{p})$ in $\dot{G}$. Notice that we must have at least attempted to insert $\hat{q}$ in the new spanner. If $\hat{q}$ is not in level $i+1$ ($\hat{q}$ may have been demoted), then we let $u_{\hat{p}} = P(\hat{q})$. If $\hat{q}$ has not been inserted yet (an insertion was attempted but $\hat{q}$ was demoted and has not been reinserted yet), then let $u_{\hat{p}} = c(\hat{q})$.

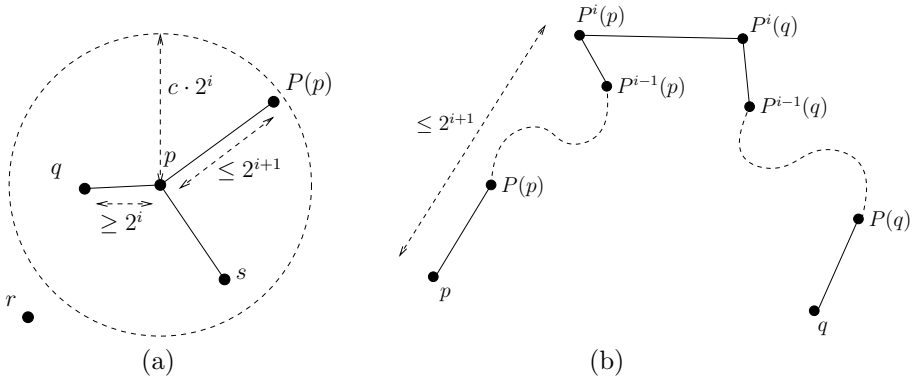

Figure 1: (a) The neighbors of p in level i include q, s, and $P(p)$. $P(p)$ must also be in level $i+1$. Since we have denoted p's parent to be another node, we know that p is not in level $i+1$. This diagram does not specify if q, r, or s are also in level $i+1$. (b) A path between p and q that fulfills the $(1+\epsilon)$-spanner property. There is no mutual link between $P^{i-1}(p)$ and $P^{i-1}(q)$.

Lemma 2 *Let $\hat{p}$ be a node inserted in to $\hat{G}$ at level i. $P(\hat{p}) \in N_j(P^{j-i-1}(u_{\hat{p}}))$ for some $j \geq i+1$ or does not exist. That is, the parent of $\hat{p}$ is a neighbor of $u_{\hat{p}}$, or a neighbor of an ancestor of $u_{\hat{p}}$, or does not exist (in which case $\hat{p}$ belongs at the top of the hierarchy).*

Proof. Define $\hat{q}$ such that $\dot{q} = P(\dot{p})$ in $\dot{G}$. We know that $u_{\hat{p}}$ may be one of three nodes, $\hat{q}$, $P(\hat{q})$, or $c(\hat{q})$. We know that $|\hat{p}\hat{q}| \leq 2^{i+1} + 2r$, $|\hat{p}P(\hat{q})| \leq |\hat{p}\hat{q}| + |\hat{q}P(\hat{q})| \leq 2^{i+1} + 2r + 2^{i+1}$, and $|\hat{p}c(\hat{q})| \leq |\hat{p}\hat{q}| + |\hat{q}c(\hat{q})| \leq 2^{i+1} + 2r + 2^{i+1}$. Since all of these distances are less than or equal to $(c-2)2^{i+1}$, we can invoke Lemma 1. Therefore, for all cases of $u_{\hat{p}}$, $P(\hat{p})$ and $P^{j-i-1}(u_{\hat{p}})$ must be neighbors for some $j \geq i+1$ or $\hat{p}$ is at the top and $P(\hat{p})$ does not exist. □

By Lemma 2, in order to find the parent of $\hat{p}$ we must search neighbors of $u_{\hat{p}}$ and neighbors of ancestors of $u_{\hat{p}}$. More specifically, $P(\hat{p})$ is the lowest node in this set such that $|\hat{p}P(\hat{p})| < 2^{j+1}$ where $\hat{p}$ is in some level $j \geq i$ and $P(\hat{p})$ is in level $j+1$. We search for $P(\hat{p})$ from bottom up, first searching all neighbors of $u_{\hat{p}}$ in one level before moving upward to the next level. The first node that fulfills the parent requirement is selected to be $\hat{p}$'s parent. As we move up the hierarchy in search of $P(\hat{p})$, we promote $\hat{p}$ up the hierarchy as well. For each level j that $\hat{p}$ is promoted to, we let $P(\hat{p}^{(j-1)}) = \hat{p}^{(j)}$. If $\hat{p}$ reaches the top of the hierarchy, then we know that $\hat{p}$ has no parent and $\hat{p}$ becomes the parent of the node that was previously at the top.

Step 3: Find all neighbors. In this step we find the neighbors of $\hat{p}$ for every level it resides in. If $\hat{p}$ lies in the highest level of the graph, then the only neighbor $\hat{p}$ has is itself. Otherwise, all neighbors of $\hat{p}$ in a level must also be a cousin (children of neighbors of $P(\hat{p})$) of $\hat{p}$ in the same level. This is trivial to prove. Therefore, we only need to search among the cousins of $\hat{p}$ to determine if they are neighbors. If $|\hat{p}\hat{q}| < c \cdot 2^j$ where $\hat{p}$ and $\hat{q}$ are cousins in level j, then we denote $\hat{p}$ and $\hat{q}$ to be neighbors and create a link between the two nodes. Note that if we have promoted $\hat{p}$, we need to repeat our search in all levels $\hat{p}$ resides in. This search is executed top down in the hierarchy. After all neighbors of $\hat{p}$ are found in all levels, we conclude the insertion process for $\hat{p}$.

When a new level is created below the bottom level j, we have to maintain our structure. To do this, all nodes in level j are copied to the new level $j-1$ and neighbors in level j are searched to find neighbors in level $j-1$.

After all nodes are inserted, it is possible that we need to demote a node below G_0. In this case, we can simply add more levels to the bottom of the hierarchy. It is also possible that we promote so many nodes such that lower levels are redundant. In this case, we can delete the redundant levels.

The following theorem concludes with the correctness of the algorithm.

Theorem 3 *This algorithm creates a correct* DefSpanner *$\hat{G}$ of $\hat{S}$.*

Proof. First, we argue that our algorithm terminates. A node may not be demoted by the same node more than once. Therefore the number of demotions is bounded, all nodes are inserted into $\hat{G}$ and the algorithm terminates.

In a DefSpanner, in every level, all points must be sufficiently far apart and every point has a designated parent (except for the topmost node) and links to all neighbors. We argue that, our algorithm, after each point is inserted into $\hat{G}$, constructs a graph that fulfills all of these properties.

The distance between every pair of nodes in any level i must be greater than or equal to 2^i. We prove this by contradiction. Let us assume that two points $\hat{p}$ and $\hat{q}$ are too close in some level i, that is $|\hat{p}\hat{q}| < 2^i$. Without loss of generality, let us assume that $\hat{p}$ is inserted after $\hat{q}$ which also means that $\hat{p}$ has been inserted into a level that $\hat{q}$ resides in. If $|\hat{p}\hat{q}| < 2^i$ then $|\hat{p}\hat{q}| < 2^j$ must be

true for any level $j \geq i$ and specifically the level that $\hat{p}$ is inserted into. Here we have a contradiction because in our algorithm, $\hat{p}$ would have been demoted down to a level i where $|\hat{p}\hat{q}| \geq 2^i$.

Now we must show that every node in every level has a designated parent. When a point $\hat{p}$ is inserted into $\hat{G}$, either $c(\hat{p})$ is assigned to be the parent of $\hat{p}$ or neighbors of ancestors of $u_{\hat{p}}$ are searched for $P(\hat{p})$. By Lemma 2, if the latter is the case, we know that $P(\hat{p})$ must be found or $\hat{p}$ is promoted to the top of the hierarchy. The parent for each node in the lower levels is the same node in the level above. This is assigned when a node is promoted up the hierarchy or when a level is added to the bottom of the hierarchy.

This leaves showing that the neighbors for each node in all levels are found. For any pair of nodes that are neighbors, the connection is made when the second node is inserted into the graph. In the third step, after the parent of the second node is found, the algorithm then finds the first node among the second node's cousins and creates the link between the two. □

4 Analysis

The cost of preprocessing is the DefSpanner construction cost or $O(n(r+1/\varepsilon)^d \log_2 \alpha)$ [9]. The running time of our algorithm is bounded by the cost of inserting each node, and the cost of adding new levels to the bottom of the hierarchy.

There are three distinct phases during the insertion of a node: checking for demotion, finding the parent, and finding the new neighbors. We will bound the cost of the three phases separately.

Lemma 4 *The number of comparisons conducted while determining whether a node should be demoted is* $O(n(r+\frac{1}{\varepsilon})^d)$.

Proof. The first step of inserting a node $\hat{p}$ is to check the distance between $\hat{p}$ and all of its previous neighbors in $\dot{G}$ to determine if any two nodes are too close and if so, $\hat{p}$ is to be demoted. According to our algorithm, if a node $\hat{p}$ is demoted, then the next time it is inserted into $\hat{G}$, it is generally inserted before other nodes in that level are inserted. This way, if there is another node that is too close to $\hat{p}$, then the other node is demoted instead of $\hat{p}$. Therefore, for the majority of demoted nodes, they are only demoted once.

The only case where a node is demoted more than once is when two nodes are demoted to the same level and when they are reinserted, they are found to be too close and one of the two nodes is demoted again. In this case, if we let the other node be $\hat{q}$ and the level the two nodes are reinserted to be i, then $|\dot{p}\dot{q}| \leq |\hat{p}\dot{p}| + |\hat{p}\hat{q}| + |\hat{q}\dot{q}| \leq 2r + 2^i \leq c \cdot 2^i \leq c \cdot 2^j$ for some $j > i$ where $\dot{p}, \dot{q} \in \dot{G}_j$ and either $\dot{p} \notin \dot{G}_{j+1}$ or $\dot{q} \notin \dot{G}_{j+1}$. Then $\dot{p}$ and $\dot{q}$ must be neighbors in level j. Since one of the nodes has been demoted before the other was inserted, $\hat{p}$ and $\hat{q}$ have not been compared yet in the construction of $\hat{G}$. Therefore, the total number of comparisons is bounded by n times the number of neighbors a node may have in one level.

Note that previously, we mentioned that the number of neighbors a node has on one level is bounded by $(1+2c)^d - 1$. Given our definition of c, this is $O((r+\frac{1}{\varepsilon})^d)$. Thus, the number of comparisons is bounded by $O(n(r+\frac{1}{\varepsilon})^d)$. □

Lemma 5 *Two nodes can be neighbors in at most* $O(\log(r+\frac{1}{\varepsilon}))$ *levels.*

Proof. Let $\hat{p}$ and $\hat{q}$ be two nodes that are only neighbors in levels j to i, $j \leq i$. Since $\hat{p}$ and $\hat{q}$ are both in level i and are neighbors in level j, $2^i \leq |\hat{p}\hat{q}| \leq c \cdot 2^j$. From this, we can bound the number of levels $\hat{p}$ and $\hat{q}$ are neighbors in, $i - j + 1$, by $\log c + 1$. Therefore, the maximum number of levels that two nodes can be neighbors in is $\log c + 1 = O(\log(r+\frac{1}{\varepsilon}))$.

□

Lemma 6 *Let $\hat{p}$ be a node inserted into $\hat{G}$ at level i and let j be the highest level it resides in. In levels $i+1$ to $j-1$, $\hat{p}$ and all neighbors of ancestors of $u_{\hat{p}}$ are cousins.*

Proof. For a level k, $i+1 \leq k \leq j-1$, $\hat{p}$ is in level $k+1$ and $P^{k-i-1}(u_{\hat{p}})$ is the ancestor of $u_{\hat{p}}$ in level k. Let $\hat{q}$ be a node in level k that is also neighbor of $P^{k-i-1}(u_{\hat{p}})$. We need to show that the parents of $\hat{p}$ and $\hat{q}$ are neighbors in level $k+1$.

From Lemma 2 we know that $|\hat{p}u_{\hat{p}}| \leq (c-2)2^{i+1}$. We can determine the distance between $\hat{p}$ and $\hat{q}$ to be $|\hat{p}\hat{q}| \leq |\hat{p}u_{\hat{p}}| + |u_{\hat{p}}P^{k-i-1}(u_{\hat{p}})| + |\hat{q}P^{k-i-1}(u_{\hat{p}})| \leq (c-2)2^{i+1} + 2 \cdot 2^k - 2^{i+2} + c \cdot 2^k$. Using this information, we can determine the distance between the parents of $\hat{p}$ (which is itself) and $\hat{q}$ to be $|\hat{p}P(\hat{q})| \leq |\hat{p}\hat{q}| + |\hat{q}P(\hat{q})| \leq (c-2)2^{i+1} + 2 \cdot 2^k - 2^{i+2} + c \cdot 2^k + 2^{k+1} = (c-4)2^{i+1} + (c+4)2^k \leq (c-4)2^k + (c+4)2^k = c \cdot 2^{k+1}$. Therefore, the parents of $\hat{p}$ and $\hat{q}$ are neighbors in level $k+1$ and $\hat{p}$ and $\hat{q}$ are cousins in level k.

□

Lemma 7 *The total cost of finding the new neighbors of all nodes during their insertion is* $O(n(r+\frac{1}{\varepsilon})^d \log(r+\frac{1}{\varepsilon}))$.

Proof. For a node $\hat{p}$ inserted into level i and promoted to level j, the search for $\hat{p}$'s neighbors begins after the parent has been found. First, the cousins of $\hat{p}$ in level j are searched for neighbors. Since, a, the maximum number of children a node can have is bounded by 5^d, this takes $O((r+\frac{1}{\varepsilon})^d)$ time. For all nodes, it takes $O(n(r+\frac{1}{\varepsilon})^d)$ time. Then the cousins in levels below j are checked for neighbors. We can bound the number

of nodes we check by charging each cousin to the neighbor link between $\hat{p}$ and the cousins' parent. We charge a operations to each link. By Lemma 5, we know that each edge in $\hat{G}$ represents a neighbor link between two nodes in $O(\log(r+\frac{1}{\varepsilon}))$ levels. When we take into account the time it takes to find neighbors for all nodes, we can bound all operations (that do not include finding neighbors of a node in the highest level it resides in) by the number of edges in $\hat{G}$ multiplied by $O(a\log(r+\frac{1}{\varepsilon}))$. The number of edges in $\hat{G}$ is bounded by $O(n(r+\frac{1}{\varepsilon})^d)$. The total time it takes to find the new neighbors of all nodes is $O(n(r+\frac{1}{\varepsilon})^d) + O(5^d \cdot n(r+\frac{1}{\varepsilon})^d \log(r+\frac{1}{\varepsilon})) = O(n(r+\frac{1}{\varepsilon})^d \log(r+\frac{1}{\varepsilon}))$. □

Lemma 8 *The total cost of inserting all nodes into $\hat{G}$ is $O(n(r+\frac{1}{\varepsilon})^d \log(r+\frac{1}{\varepsilon}))$.*

Proof. We have already proved time bounds for two phases of insertion: checking for demotions ($O(n(r+\frac{1}{\varepsilon})^d)$ by Lemma 4) and finding new neighbors ($O(n(r+\frac{1}{\varepsilon})^d \log(r+\frac{1}{\varepsilon}))$ by Lemma 7). All that we need to do is to bound the time spent searching for the parents of the nodes during insertion. By Lemma 6, we note that for a node $\hat{p}$ where $\hat{p} \in \hat{G}_j$, $\hat{p} \notin \hat{G}_{j+1}$, all nodes that we test as a possible parent in all levels below j are also checked as possible neighbors. All nodes that we test as a possible parent in level $j+1$ for a node $\hat{p}$ can be bound by the number of neighbors $u_{\hat{p}}$ can have or $O((r+\frac{1}{\varepsilon})^d)$. Therefore, for all nodes, the time it takes to find the parents is $O(n(r+\frac{1}{\varepsilon})^d) + O(n(r+\frac{1}{\varepsilon})^d \log(r+\frac{1}{\varepsilon}))$. When we add up all costs of insertion, the term that dominates is $O(n(r+\frac{1}{\varepsilon})^d \log(r+\frac{1}{\varepsilon}))$. □

Lastly, we need to consider the time it takes to add new levels to the bottom of the hierarchy while the spanner is built.

Lemma 9 *The cost of adding new levels to the bottom of the hierarchy during spanner construction is $O(n(r+\frac{1}{\varepsilon})^d \log(r+\frac{1}{\varepsilon}))$.*

Proof. Each time a new level is added to the bottom of the hierarchy, all nodes on the bottom level are compared to all of their neighbors. By Lemma 5, the maximum number of times that two nodes may be compared is the maximum number of levels they are neighbors in or $O(\log(r+\frac{1}{\varepsilon}))$. Since the number of edges in a DefSpanner represents the number of neighbor pairs, the total cost is $O(n(r+\frac{1}{\varepsilon})^d \log(r+\frac{1}{\varepsilon}))$. □

Now we can finalize our analysis.

Theorem 10 *The entire algorithm after preprocessing takes $O(n(r+\frac{1}{\varepsilon})^d \log(r+\frac{1}{\varepsilon}))$.*

Proof. Summing the costs of inserting all nodes (Lemma 8) and creating new levels (Lemma 9), the entire algorithm takes $O(n(r+\frac{1}{\varepsilon})^d \log(r+\frac{1}{\varepsilon}) + n(r+\frac{1}{\varepsilon})^d \log(r+\frac{1}{\varepsilon})) = O(n(r+\frac{1}{\varepsilon})^d \log(r+\frac{1}{\varepsilon}))$. □

5 Conclusion

We have presented an $O(n(r+\frac{1}{\varepsilon})^d \log(r+\frac{1}{\varepsilon}))$ time algorithm to construct a $(1+\varepsilon)$-spanner with $O(n\log(\alpha)(r+\frac{1}{\varepsilon})^d)$ preprocessing, when the accurate positions of the points are revealed and each point is at most distance r from its old position. There are many applications of a DefSpanner [9]: well-separated pair decomposition, all near neighbors query, $(1+\varepsilon)$-nearest neighbor, closest pair and collision detection, and k-center. Note that our algorithm for constructing the DefSpanner will immediately imply that given the accurate positions we can immediately get a structure for the above problems in linear running time.

Acknowledgements The authors would like to acknowledge the support from NSF through CNS-1116881, CNS- 1217823, and DMS-1221339.

References

[1] M. A. Abam, P. Carmi, M. Farshi, and M. Smid. On the power of the semi-separated pair decomposition. In *Proceedings of the 11th International Symposium on Algorithms and Data Structures*, WADS '09, pages 1–12, Berlin, Heidelberg, 2009. Springer-Verlag.

[2] M. A. Abam and M. de Berg. Kinetic spanners in $\mathbb{R}^d$. *Discrete Comput. Geom.*, 45(4):723–736, June 2011.

[3] K. Buchin, M. Löffler, P. Morin, and W. Mulzer. Preprocessing imprecise points for delaunay triangulation: Simplified and extended. *Algorithmica*, 61(3):674–693, 2011.

[4] K. Buchin and W. Mulzer. Delaunay triangulations in $o(sort(n))$ time and more. In *Proceedings of the 2009 50th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '09, pages 139–148, Washington, DC, USA, 2009. IEEE Computer Society.

[5] P. B. Callahan and S. R. Kosaraju. Faster algorithms for some geometric graph problems in higher dimensions. In *Proceedings of the 4th annual ACM-SIAM Symposium on Discrete algorithms*, SODA '93, pages 291–300, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[6] O. Devillers. Delaunay triangulation of imprecise points, preprocess and actually get a fast query time. *JoCG '11*, pages 30–45, 2011.

[7] D. Eppstein. Spanning trees and spanners. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 425–461. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.

[8] E. Ezra and W. Mulzer. Convex hull of imprecise points in $o(n\log n)$ time after preprocessing. In *Proceedings of the 27th Annual ACM Symposium on Computational Geometry*, SCG '11, pages 11–20, New York, NY, USA, 2011. ACM.

[9] J. Gao, L. J. Guibas, and A. Nguyen. Deformable spanners and applications. In *In Proceedings of the 20th*

ACM Symposium on Computational Geometry (SCG 04, pages 179–199, 2004.

[10] L.-A. Gottlieb and L. Roditty. An optimal dynamic spanner for doubling metric spaces. In *Proceedings of the 16th Annual European Symposium on Algorithms*, ESA '08, pages 478–489, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] L. Guibas, D. Salesin, and J. Stolfi. Constructing strongly convex approximate hulls with inaccurate primitives. In *Proceedings of the International Symposium on Algorithms*, SIGAL '90, pages 261–270, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[12] M. Held and J. S. B. Mitchell. Triangulating input-constrained planar point sets. *Inf. Process. Lett.*, 109(1):54–56, Dec. 2008.

[13] M. Löffler and J. M. Phillips. Shape fitting on point sets with probability distributions. *CoRR '08*, abs/0812.2967, 2008.

[14] M. Löffler and J. Snoeyink. Delaunay triangulation of imprecise points in linear time after preprocessing. *Comput. Geom. Theory Appl.*, 43(3):234–242, Apr. 2010.

[15] T. Nagai, S. Yasutome, and N. Tokura. Convex hull problem with imprecise input. In *Revised Papers from the Japanese Conference on Discrete and Computational Geometry*, JCDCG '98, pages 207–219, London, UK, UK, 2000. Springer-Verlag.

[16] G. Narasimhan and M. Smid. *Geometric Spanner Networks*. Cambridge University Press, 2007.

[17] L. Roditty. Fully dynamic geometric spanners. In *Proceedings of the 23rd Annual Symposium on Computational Geometry*, SCG '07, pages 373–380, New York, NY, USA, 2007. ACM.

[18] D. Salesin, J. Stolfi, and L. Guibas. Epsilon geometry: Building robust algorithms from imprecise computations. In *Proceedings of the 5th Annual Symposium on Computational Geometry*, SCG '89, pages 208–217, New York, NY, USA, 1989. ACM.

[19] M. van Kreveld, M. Löffler, and J. S. B. Mitchell. Preprocessing imprecise points and splitting triangulations. *SIAM J. Comput.*, 39(7):2990–3000, June 2010.

[20] M. L. Yiu, C. S. Jensen, X. Huang, and H. Lu. Spacetwist: Managing the trade-offs among location privacy, query performance, and query accuracy in mobile services. In *ICDE*, pages 366–375, 2008.