Computing the greedy spanner in near-quadratic time

Prosenjit Bose* Paz Carmi* Mohammad Farshi* Anil Maheshwari*
Michiel Smid*

Abstract

It is well-known that the greedy algorithm produces high quality spanners and therefore is used in several applications. However, for points in d-dimensional Euclidean space the greedy algorithm has near-cubic running time. In this paper we present an algorithm that computes the greedy spanner (spanner computed by the greedy algorithm) for a set of n points from a metric space with bounded doubling dimension in $\mathcal{O}(n^2 \log n)$ time using $\mathcal{O}(n^2)$ space. Since the lower bound for computing such spanners is $\Omega(n^2)$, the time complexity of our algorithm is optimal to within a logarithmic factor.

1 Introduction

A *network* on a point set V is a connected graph $G(V, E)$. When designing a network several criteria are taken into account. In particular, in many applications it is important to ensure a fast connection between every pair of points. For this it would be ideal to have a direct connection between every pair of points—the network would then be a complete graph—but in most applications this is unacceptable due to very high costs associated in constructing such networks. This leads to the concept of spanners, as defined below.

Let $(V, \mathbf{d})$ be a metric space and $G(V, E)$ be a network on V such that the weight of each edge $e \in E$ is equal to the distance between its endpoints. We say that G is a *t-spanner* of V, for some constant $t > 1$, if for each pair of points $u, v \in V$, there exists a path in G between u and v of length at most $t \cdot \mathbf{d}(u, v)$. The *dilation* or *stretch factor* of G is the minimum t for which G is a t-spanner of V. Spanners were introduced by Peleg and Schäffer [11] in the context of distributed computing and by Chew [3] in the geometric context. Since then spanners have received a lot of attention, see [9, 10].

A classical algorithm for computing a geometric spanner for any set V of n points in $\mathbb{R}^d$, where $\mathbf{d}$ is the Euclidean metric, and for any fixed $t > 1$, is the *greedy algorithm* which was proposed independently by Bern in 1989 and Althöfer *et al.* [1]. The main steps of this algorithm are the following (see Algorithm A.1 for more details): first sort all the pairs of points in V with respect to their distances in increasing order and initialize the greedy graph $G(V, E)$ so that its edge set is empty. Next, the pairs are processed in sorted order. Processing a pair (u, v) entails a shortest path query in G between u and v. If there is no t-path between u and v (a path of length at most $t \cdot \mathbf{d}(u, v)$) in G then (u, v) is added to G, otherwise it is discarded. We will refer to the graph G generated by the greedy algorithm as the *greedy spanner*. The focus of this paper is to compute the greedy spanner efficiently.

*School of Computer Science, Carleton University, Ottawa, Ontario, K1S 5B6, Canada.
Email:{jit@scs.carleton.ca, paz@cg.scs.carleton.ca, m.farshi@gmail.com, anil@scs.carleton.ca, michiel@scs.carleton.ca}
Research supported in part by NSERC and MRI.

The time complexity of the greedy algorithm stated above is $\mathcal{O}(mn^2 + n^3 \log n)$ and it uses $\mathcal{O}(n^2)$ space, where n is the number of points and m is the number of edges in the spanner. It has been shown that for any set V of n points in $\mathbb{R}^d$ and for any fixed $t > 1$ in the Euclidean metric, the greedy spanner has $\mathcal{O}(n)$ edges, bounded degree, and its total weight is $\mathcal{O}(wt(MST(V)))$, where $wt(MST(V))$ is the weight of the minimum spanning tree of V [4, 9]. Unfortunately, the naïve implementation of the greedy algorithm runs in near-cubic time.

Due to the high time complexity of computing the greedy spanner, researchers have proposed algorithms for computing other types of sparse t-spanners, see [9]. But it turns out that in practice the greedy algorithm produces t-spanners of high quality in comparison to other spanners [5, 6]. For example, they have been used for protein visualization as a low-weight data structure, which is used as a contact map, that allows approximate reconstruction of the full distance matrix [12].

For points in the plane under the Euclidean metric, Farshi and Gudmundsson [5, 6] introduced a speed-up strategy that generates the greedy spanner much faster in practice. They conjectured that their algorithm runs in $\mathcal{O}(n^2 \log n)$ time. However, as we will show in this paper, this conjecture is incorrect.

For general metric spaces, there are cases where the complete graph is the only t-spanner of a point set. For example, assume V is a set of points from a metric space where the distance between any two distinct points is equal to 1. For any $1 < t < 2$, the complete graph is the only t-spanner of V. Therefore, in general metric spaces, we can not guarantee that the generated graph is sparse. The doubling dimension is defined as follows. Let λ be the smallest integer such that for each real number r, any ball of radius r can be covered by at most λ balls of radius $r/2$. The *doubling dimension* of V is defined to be $\log \lambda$. The doubling dimension is a generalization of the Euclidean dimension, as the doubling dimension of d-dimension Euclidean space is $\Theta(d)$. For such metric space we will show that the number of edges is linear.

1.1 Main results and organization of the paper

The main result of this paper is that for any metric space V of bounded doubling dimension, the greedy spanner of V can be computed in $\mathcal{O}(n^2 \log n)$ time, where $n = |V|$. The organization of the remainder of this paper is as follows. In Section 2, we review the improved greedy algorithm of [5, 6] and give a counterexample to the conjecture that this algorithm only performs $\mathcal{O}(n)$ shortest path queries. In Section 3, we present an algorithm that generates the greedy spanner in near-quadratic time for some special cases. These results are generalized to metric spaces of bounded doubling dimension in Section 4.

2 The improved greedy algorithm

As mentioned above the running time of a naïve implementation of the greedy algorithm is $\mathcal{O}(mn^2 + n^3 \log n)$. Farshi and Gudmundsson [5, 6] introduced an improved version of the algorithm and showed that it improves the running time for constructing the greedy spanner considerably in practice on point sets in the plane with the Euclidean metric. The improved greedy algorithm is as follows. The algorithm is the same as the original greedy algorithm (Algorithm A.1) except that it uses a matrix to save the length of the shortest path between each two points and updates the matrix only when it needs to. Thus the matrix is not always up to date. Instead of computing a shortest path for each pair, (line 6 of Algorithm A.1) it first checks the matrix to see if there is a t-path; if the answer is no, then it performs a shortest path query and updates the matrix which enables us to answer the distance queries correctly, see Algorithm B.1 for more details. They conjectured that the improved greedy algorithm performs only $\mathcal{O}(n)$ shortest path queries, and thus the running time is $\mathcal{O}(n^2 \log n)$.

2.1 A Counterexample

We give an example which shows that the improved greedy algorithm (Algorithm B.1) may perform $\Theta(n^2)$ shortest path queries. Consider a set $S = \{p_0, p_1, \ldots, p_{n-1}\}$ of n points on the real line such that $p_i = 2^i$. The algorithm sorts all pairs of points based on their distance. We also assume that for each pair (p_i, p_j) the index of the first point in the pair is less than the index of the second point, i.e. $i < j$. Now the claim is that the algorithm performs a shortest path query for each pair of points.

To show this we split the sorted list of pairs in blocks B_i, $0 < i < n$, such that $B_i = \{(p_{i-1}, p_i), (p_{i-2}, p_i), \ldots, (p_0, p_i)\}$. Obviously the algorithm starts with the pairs in B_1 and then continue to the pairs in B_2 and so on. For arbitrary i, the first pair in B_i that the algorithm checks is (p_{i-1}, p_i) and since the point p_i is disconnected from the current graph, the shortest path length between p_i and any other point is infinity in the weight matrix. Processing the pair (p_{i-1}, p_i) entails performing a shortest path query with source at p_{i-1} and updating the weight matrix and then adding an edge between p_{i-1} and p_i to the graph. Note that because the algorithm updates the row and the column corresponding to p_{i-1} in the weight matrix, the weight matrix still shows the infinity distance between p_i and p_j for any $j < i - 1$. The algorithm then process (p_{i-2}, p_i) and because of the infinity distance between p_i and p_{i-2} according to corresponding entry in the weight matrix, the algorithm performs a shortest path query with source at p_{i-2} and updates the row and column related to it. Note that after this update, the distance between p_i and p_j, for $0 \leq j < i - 2$, remains infinity in the weight matrix. Because all the shortest paths are with source p_j, for $j < i$, none on them updates the distance between p_i and p_k for $k < j$. This means that the algorithm performs a shortest path query for each pair of points.

Note that if we change the algorithm such that when a new edge is added to the greedy spanner, it performs a shortest path query for both endpoints of the new edge and updates the weight matrix then the new algorithm performs only $\mathcal{O}(n)$ shortest path queries on the above counterexample. However, in the full version of this paper, we give an example on which the new algorithm needs to perform $\Omega(n \log n)$ shortest path queries.

3 Near-quadratic time algorithm for special cases

Let V be a set of n points in a metric space with distance function $\mathbf{d}$. As mentioned before, for generating the greedy t-spanner, we start with a graph $G(V, E)$ with no edges and we go through all the pairs of points in V (in increasing order based on their distances). For each pair we check if there exists a t-path between them in G, if not we add the edge between them in G.

In the new algorithm we basically do the same thing as the improved greedy algorithm (see Algorithm B.1) in the sense that we use the weight matrix to answer the shortest path queries. The differences are the following:

- We process the pairs whose distance is less than L, for a real number L, exactly in the way as in the original (or improved) greedy algorithm.
- We split the rest of the pairs, into buckets such that the ith bucket contains all the pairs whose distance is between $2^{i-1}L$ and 2^iL.
- During the processing of each bucket, we keep the shortest path between the pairs in the bucket up to date. To do this we update the shortest path between all pairs in the bucket before processing the pairs in the bucket and during the process we update it when we add an edge to the graph. Note that we keep the shortest path length between pairs in a weight matrix.

Without loss of generality we assume that the diameter of the point set is one. Let $0 < L < 1$ be a real number to be specified later. We split the pairs of points into $k = \mathcal{O}(\log(\frac{1}{L}))$ buckets such that the ith bucket, i.e. E_i, contains all the pairs with distance in $[2^{(i-1)}L, 2^iL)$. Let E_0 contain the pairs with distance less than L.

The algorithm starts with the pairs in E_0. It process all the pairs in the set E_0 in the same manner as the original (or improved) greedy algorithm does. Therefore, if E_0 contains $\mathcal{O}(n^\beta)$ pairs then processing it takes $\mathcal{O}(n^{\beta+1} \log n)$ time. Let G denote the latest greedy spanner after processing E_0.

Now we show how to process the remaining buckets. Assume that we have processed buckets $E_1, E_2, \ldots, E_{i-1}$ and we need to process bucket E_i. Before processing the edges in this bucket, we compute single source shortest path with source at each point p and update the weight matrix. Then we make "*local*" updates when we add an edge to the graph. By "local" update, we mean we update the weight matrix for all points nearby one of the endpoints of the new edge. Since the weight matrix is up to date for all pairs in the current bucket we can answer the t-path queries in constant time using the weight matrix. For details see Algorithm 3.1.

Algorithm 3.1: NEW-GREEDY(V, t, L)

Input: V, $t > 1$ and $L > 0$.
Output: t-spanner $G = (V, E')$.

foreach $(u, v) \in V^2$ **do** $weight(u, v) := \infty$;
$E :=$ sorted list of all the pairs in V^2 by increasing distance; `/*ties are broken arbitrarily */`
$E_0 :=$ all the point pairs in E with distance in $[0, L)$;
$j := 1$;
while $E \setminus (\bigcup_{k=1}^{j-1} E_k) \neq \emptyset$ **do**
    $E_j :=$ all the point pairs in $E \setminus (\bigcup_{k=1}^{j-1} E_k)$ with distance in $[2^{j-1}L, 2^jL)$;
    $j := j + 1$;
end
$l := j - 1$;
$E' := \emptyset$;
$G := (V, E')$;
Process pairs in E_0 in the same way as the original (or improved) greedy algorithm;
for $i := 1, \ldots, l$ **do**
    $L_i := 2^{i-1}L$;
    foreach $u \in V$ **do**
        Compute single-source shortest paths with source at u and update the weight matrix;
    end
    foreach $(u, v) \in E_i$; `/* in sorted order */`
    do
        if $weight(u, v) \leq t \cdot \mathbf{d}(u, v)$ **then**
            Discard (u, v);
        else
            $E' := E' \cup \{(u, v)\}$;
            foreach $p \in V$ **do** **if** $\mathbf{d}(p, u) < (t - \frac{1}{2})L_i$ *or* $\mathbf{d}(p, v) < (t - \frac{1}{2})L_i$ **then**
                Compute single-source shortest paths with source at p and update the weight matrix;
            end
        end
    end
end
return $G(V, E')$;

First we show that the algorithm computes the greedy spanner and then show that it runs in near-quadratic time for some special cases.

Theorem 1 *Algorithm 3.1 generates the greedy t-spanner of the input point set.*

Proof. To prove the correctness of the algorithm, we need to prove that the t-path queries (line 20 of Algorithm 3.1) are answered correctly.

Let (p, q) be an arbitrary pair in E_i with $\mathbf{d}(p, q) \in [L_i, 2L_i)$ which is about to be processed in the algorithm. If there is no t-path between p and q in G then the algorithm answers the t-path query correctly since the entry in the weight matrix corresponding to the pair is at least equal to the shortest path length between p and q in G.

Assume that there is a t-path between p and q in G. We have two cases:

Case 1: The shortest path between p and q in G does not pass through any edges that was added during processing pairs in E_i. In this case we are done because before processing the pairs in E_i, we updated all-pair shortest paths and adding new edges to the graph does not change the shortest path length between p and q. So the entry in the weight matrix corresponding to p and q show the shortest path length between them in G correctly.

Case 2: The shortest path P in G between p and q passes through some edge of E_i. Among all edges of $E_i \cap P$, let (u, v) be the one that was added last by the algorithm. We may assume without loss of generality that, starting at p, the path P goes to u, then traverses (u, v), and then continues to q. We define

$$S_{(u,v)} = \{x \in V : \mathbf{d}(x, u) < (t - \frac{1}{2})L_i \text{ or } \mathbf{d}(x, v) < (t - \frac{1}{2})L_i\}.$$

We claim (and show below) that p or q belongs to $S_{(u,v)}$. This will imply that, in the iteration in which (u, v) is added to the graph, the algorithm computes the exact shortest-path length between p and all vertices of V, or between q and all vertices of V. Therefore, at the moment when (p, q) is processed, the value of $weight(p, q)$ is equal to the shortest-path length in G between p and q.

It remains to prove the claim. Assume that neither p nor q is contained in $S_{(u,v)}$. Then $\mathbf{d}(p, u) \geq (t - \frac{1}{2})L_i$ and $\mathbf{d}(q, v) \geq (t - \frac{1}{2})L_i$. Thus, if we denote the shortest-path length between p and q by $\mathbf{d}_G(p, q)$, then

$$\mathbf{d}_G(p, q) \geq \mathbf{d}(p, u) + \mathbf{d}(u, v) + \mathbf{d}(v, q) \geq 2(t - \frac{1}{2})L_i + L_i = 2tL_i > t \cdot \mathbf{d}(p, q),$$

which contradicts the fact that there is a t-path in G between p and q. □

3.1 Running time

Now we show that the algorithm runs in near quadratic time in some special cases. Before that we need to recall the well-separated pair decomposition (WSPD) developed by Callahan and Kosaraju [2] in d-dimensional Euclidean space.

Definition 1 *Let $s > 0$ be a real number, referred to as the* separation constant*. We say that two point sets A and B in $\mathbb{R}^d$ are* well-separated *with respect to s, if there are two disjoint balls D_A and D_B of the same radius, r, such that*

(i) D_A contains A and D_B contains B,

(ii) the distance between D_A and D_B is at least $s \cdot r$.

Lemma 1 ([2]) *Let A and B be two finite sets of points that are well-separated with respect to s, let x and p be points of A, and let y and q be points of B. Then*

(i) $|xy| \leq (1 + 4/s) \cdot |pq|$, and

(ii) $|px| \leq (2/s) \cdot |pq|$.

Definition 2 *Let V be a set of n points in $\mathbb{R}^d$ and let $s > 0$ be a real number. A* well-separated pair decomposition *(WSPD) for V with respect to s is a collection $\mathcal{W} := \{(A_1, B_1), ..., (A_m, B_m)\}$ of pairs of non-empty subsets of V such that*

1. *A_i and B_i are well-separated with respect to s, for all $i = 1, \ldots, m$.*
2. *for any two distinct points p and q of V, there is exactly one pair (A_i, B_i) in the collection, such that (i) $p \in A_i$ and $q \in B_i$ or (ii) $q \in A_i$ and $p \in B_i$.*

The number of pairs, m, is called the *size* of the WSPD. Callahan and Kosaraju [2] have shown that any set $V \subseteq \mathbb{R}^d$ admits a WSPD of size $m = \mathcal{O}(s^d n)$. Har-Peled and Mendel [8] generalized the results to metric spaces with doubling dimension λ. They showed that any set of n points from a metric space with doubling dimension λ admits a WSPD with respect to $s > 1$, of size $\mathcal{O}(s^{\mathcal{O}(\lambda)} n)$. In the rest of the paper, we assume that V is a set of n points from a metric space with doubling dimension λ.

Observation 1 *If $\{(A_i, B_i)\}_i$ is a WSPD of a point set V with respect to $s = \frac{4(t+1)}{t-1}$ then for each i, there is at most one greedy edge between A_i and B_i in the t-spanner generated by the greedy algorithm.*

Proof. Assume that we have a pair (A, B) in the WSPD such that there exist two edges (a_1, b_1) and (a_2, b_2) in the greedy t-spanner where $a_1, a_2 \in A$ and $b_1, b_2 \in B$. Without loss of generality assume the greedy algorithm process the pair (a_1, b_1) before the pair (a_2, b_2).

Because A and B are s-well-separated pair, by Lemma 1, we have $\mathbf{d}(a_1, a_2) \leq \frac{2}{s}\mathbf{d}(a_2, b_2) < \mathbf{d}(a_2, b_2)$. By the same argument $\mathbf{d}(b_1, b_2) < \mathbf{d}(a_1, b_1)$. Therefore, there exists a t-path between a_1 and a_2 and a t-path between b_1 and b_2 when the greedy algorithm processes the pair (a_2, b_2). Let G' be the graph just before processing (a_2, b_2) and let P be a path in G' between a_2 and b_2 generated by concatenating a t-path between a_2 and a_1, the edge (a_1, b_1) and a t-path between b_1 and b_2. If $|P|$ denotes the length of the path P, then

$$\begin{aligned}
|P| &= \mathbf{d}_{G'}(a_2, a_1) + \mathbf{d}(a_1, b_1) + \mathbf{d}_{G'}(b_1, b_2) \\
&\leq t \cdot \mathbf{d}(a_2, a_1) + \mathbf{d}(a_1, b_1) + t \cdot \mathbf{d}(b_1, b_2) \\
&\leq t \cdot \frac{2}{s}\mathbf{d}(a_2, b_2) + (1 + \frac{4}{s})\mathbf{d}(a_2, b_2) + t \cdot \frac{2}{s}\mathbf{d}(a_2, b_2) \qquad \text{(by Lemma 1)} \\
&= (\frac{4t}{s} + 1 + \frac{4}{s})\mathbf{d}(a_2, b_2) \\
&= t \cdot \mathbf{d}(a_2, b_2).
\end{aligned}$$

This means that the greedy algorithm does not add (a_2, b_2) to the spanner since there already exist a t-path between them in G'. □

As a corollary, since there exists a linear size WSPD for any point set in a metric space with bounded doubling dimension, the size of a greedy t-spanner on such a point set is linear.

Lemma 2 *While processing the pairs in E_i, for each point p, line 25 in Algorithm 3.1 is executed $\mathcal{O}(\frac{1}{(t-1)^{\mathcal{O}(\lambda)}})$ times.*

Proof. For simplicity, we first prove the lemma in the 2-dimensional Euclidean case. Assume the distance between the pairs in E_i is in $[L, 2L)$. Algorithm 3.1 performs a single-source shortest path computation with source at p, after adding an edge (u, v) to the graph, if $\mathbf{d}(p, u) < (t - \frac{1}{2})L$ or $\mathbf{d}(p, v) < (t - \frac{1}{2})L$. So if we draw a ball C with center at p and radius $(2t + 1)L$, then all the edges which affect p lies inside the ball C. So the number of times we need to execute line 25 for p is at

most the number of edges in the greedy spanner with length between L and $2L$ which lie inside C. Now we show that the number of such edges is at most $\mathcal{O}(\frac{1}{(t-1)^2})$.

To show this, assume B is a square with side length $2(2t+1)L$ which includes C. We cover the square B with cells of side length $\ell = \frac{L}{\sqrt{2}(s+4)}$ where $s = \frac{4(t+1)}{t-1}$. The number of such cells inside B is $\left(2\sqrt{2}(2t+1)(s+4)\right)^2 = \mathcal{O}(\frac{1}{(t-1)^2})$. Let (u,v) be a greedy edge with distance in the interval $[L, 2L)$. First we show that the grid cells which contains u and v are s-well-separated. Assume C_1 and C_2 are balls with radius $\sqrt{2}\ell$ which contain the grid cell of u and the grid cell of v, respectively— see Figure 1. Since $\mathbf{d}(u,v) \geq L$ and the radius of the circles are $\sqrt{2}\ell$, the distance between C_1 and C_2 is at least $L - 4\sqrt{2}\ell$. Therefore

$$d(C_1, C_2) \geq L - 4\sqrt{2}\ell = L - 4\sqrt{2}\frac{L}{\sqrt{2}(s+4)} = L(\frac{s}{s+4}) = s(\frac{L}{s+4}) = s \times \sqrt{2}\ell$$

which means the cells are s-well-separated.

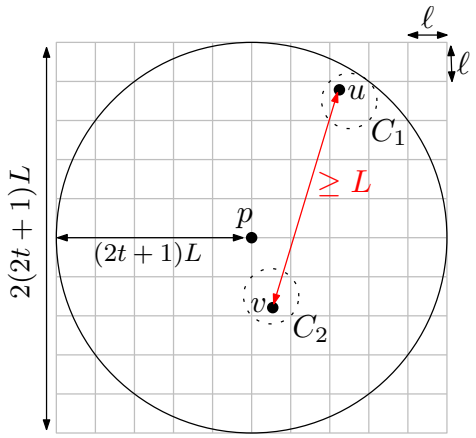

Figure 1: Illustration for the proof of Lemma 2.

Therefore, by Observation 1, we have at most one greedy edge between grid cells which are well-separated. This means that the number of the greedy edges with distance in $[L, 2L)$ inside the circle C is bounded by the number of cell pairs which is $\mathcal{O}(\frac{1}{(t-1)^4})$.

For the general case, the same argument works. In this case we use the property of doubling dimension which guarantees that each ball of radius $r > 0$ can be covered by 2^λ balls of radius $r/2$. This means that the number of balls with radius $\sqrt{2}\ell$ inside the ball C centered at p is $\mathcal{O}(\frac{1}{(t-1)^{\mathcal{O}(\lambda)}})$. □

Now we are ready to compute the time complexity of Algorithm 3.1. Clearly lines 1–11 of the algorithm take $\mathcal{O}(n^2 \log n)$ time. For line 12, if the size of E_0 is β then it takes $\mathcal{O}(\beta(m + n \log n))$ since for each pair it performs a shortest path query.

For each interval, computing all-pairs shortest path, lines 15–17, takes $\mathcal{O}(mn + n^2 \log n)$ time, and by Lemma 2, the update procedure takes at most $\mathcal{O}(\frac{1}{(t-1)^{\mathcal{O}(\lambda)}}(mn + n^2 \log n))$. Since the number of intervals is $\mathcal{O}(\log(1/L))$, processing all intervals takes $\mathcal{O}\left(\frac{\log(1/L)}{(t-1)^{\mathcal{O}(\lambda)}}(mn + n^2 \log n)\right)$ time. For a metric space with doubling dimension λ, the size of the generated graph is $\mathcal{O}(\frac{n}{(t-1)^{\mathcal{O}(\lambda)}})$, the total running time of Algorithm 3.1 is $\mathcal{O}\left(\frac{\beta n + \log(1/L) n^2 \log n}{(t-1)^{\mathcal{O}(\lambda)}}\right)$. Therefore for a point set V with the property that there exists a real number L such that $\frac{1}{L} = \mathcal{O}(n^c)$ (c is a constant) and $\beta = \mathcal{O}(n \log^2 n)$, the greedy spanner can be computed in $\mathcal{O}(n^2 \log^2 n)$ time.

3.1.1 Points sets with polynomial aspect ratio

If the input point set has aspect ratio less than n^c, for some constant c, then by scaling the point set such that the longest distance is 1 and setting $L = \frac{1}{n^c}$, we have no pair of points in the scaled point set with distance less than L. Therefore the running time of Algorithm 3.1 in this case is $\mathcal{O}(\frac{1}{(t-1)^{\mathcal{O}(\lambda)}} n^2 \log^2 n)$.

3.1.2 Uniformly distributed point set

Assume we have a set of n points uniformly distributed inside the unit square and let $L = \frac{1}{\sqrt{n}}$. Since the points are uniformly distributed, for each point p, the expected number of points inside the ball with center at p and radius L is $L^2 n$. So the expected number of pairs with distance less than L is $L^2 n^2/2 = \mathcal{O}(n)$. Therefore, the expected running time of Algorithm 3.1 in this case is $\mathcal{O}(\frac{1}{(t-1)^{\mathcal{O}(\lambda)}} n^2 \log^2 n)$.

4 Near-quadratic time algorithm for the general case

To generalize the results of the previous section bounded doubling dimension, we have to overcome obstacles. First one is that we need to speedup processing the pairs in the first set (i.e. the set E_0). The second problem is that if we decrease the number L to bound the number of pairs in the first interval, it increases the number of buckets which causes higher time complexity.

We overcome these difficulties by modifying the previous algorithm in the following way.

- we partition the $\binom{n}{2}$ pairs into linear number of buckets,
- we maintain a data structure for each point during the algorithm. When we need to update a point, instead of doing a single-shortest path computation from scratch, we use the data structure to update just the necessary part and use it to update the weight matrix.

First we claim that for updating the shortest path lengths with source at a point p, performing a length-bounded Dijkstra's algorithm is sufficient. More precisely, if we are working on point pairs in a bucket with distances in $[L, 2L)$ and we need to update the shortest path lengths with source at p, it is sufficient to update the distance between p and all the points q such that $\mathbf{d}_G(p, q) < 2tL$, where G is the current graph. The reason is that if $\mathbf{d}_G(p, q) \geq 2tL$ then either the pair (p, q) is outside the current bucket or there is no t-path between p and q in G. In both the cases, we do not need to update the weight matrix for that pair.

So, in the new algorithm, we maintain a data structure for each point, which is the same as the data structure used in the Dijkstra's single-source shortest paths algorithm. When we perform a shortest path query with bound U, we execute Dijkstra's algorithm but stop when the key of the element on the top of the priority queue (heap) is bigger than U. We also maintain a stack storing all changes that are made to the heap in this process, so that we can undo the procedure, if required.

Note that our graph is dynamic and we add edge(s) to the graph during the process. We use the "undo facility" to fix the part of the execution of Dijkstra's algorithm which will be affected by the insertion of an edge. As we will see, by adding an edge in the greedy algorithm, some part of the procedure is the same because we add edges to the graph in increasing order of their lengths. For the details of the algorithm, see Algorithm 4.1.

To complete the correctness of the algorithm we show that in the algorithm the shortest path queries are answered correctly. To do this, we show that the limited Dijkstra's algorithm on a

Algorithm 4.1: QUAD. GREEDY(V, t)

Input: V and $t > 1$.
Output: t-spanner $G = (V, E')$.

foreach $(u, v) \in V^2$ **do** $weight(u, v) := \infty$;
$E :=$ sorted list of all the pairs in V^2 by increasing distance; `/*ties are broken arbitrarily */`
$L_1 :=$ the distance between the closest pair in E;
$E_1 :=$ all the pairs in E with distance in $[L_1, 2L_1)$;
$j := 2$;
while $E \setminus (\bigcup_{k=1}^{j-1} E_k) \neq \emptyset$ **do**
    $L_j :=$ the distance between the closest pair in $E \setminus (\bigcup_{k=1}^{j-1} E_k)$;
    $E_j :=$ all the pairs in $E \setminus (\bigcup_{k=1}^{j-1} E_k)$ with distance in $[L_j, 2L_j)$;
    $j := j + 1$;
end
$l := j - 1$;
$E' := \emptyset$;
$G := (V, E')$;
foreach $u \in V$ **do** Initialize PQ_u required for executing Dijkstra's algorithm with source at u;
for $i := 1, \ldots, l$ **do**
    foreach $u \in V$;
    do
        $\tau_u := \emptyset$;
        DIJKSTRA-BOUNDED$(G, u, 2tL_i, PQ_u, \tau_u)$;
    end
    foreach $(u, v) \in E_i$; `/* in sorted order */`
    do
        if $weight(u, v) \leq t \cdot \mathbf{d}(u, v)$ **then**
            Discard (u, v);
        else
            $E' := E' \cup \{(u, v)\}$;
            foreach $p \in V$ **do** **if** $\mathbf{d}(p, u) < (t - \frac{1}{2})L_i$ *or* $\mathbf{d}(p, v) < (t - \frac{1}{2})L_i$ **then**
                UPDATE$(G, p, u, v, 2tL_i, PQ_p, \tau_p)$;
            end
        end
    end
end
return $G(V, E')$;

subgraph of the greedy spanner works exactly same as the Dijkstra's algorithm on the greedy spanner.

Lemma 3 *Let G_e be the subgraph of the greedy spanner G that contains all the edges added to the graph up to the processing of the pair $e = (p, q)$ in the greedy algorithm. Let u be an arbitrary vertex of G. As long as the key of the element on the top of the heap in Dijkstra's algorithm with source at u is less than $\mathbf{d}(p, q)$ the algorithm works the same on G and G_e.*

Proof. The proof is straight-forward. Let x be a vertex of the graph G and assume the shortest path between u and x passes through at least one edge in $E \setminus E_e$. Since the length of each edge in $E \setminus E_e$ is at least $\mathbf{d}(p, q)$, the key of the point x in the heap is at least $\mathbf{d}(p, q)$. This completes the proof. □

Algorithm 4.2: UPDATE$(G, s, u, v, L, PQ, \tau)$

1 **if** $weight(s,v) < weight(s,u) + \mathbf{d}(u,v)$ *and* $weight(s,u) < weight(s,v) + \mathbf{d}(u,v)$; `/* This means adding` (u,v) `does not change any shortest path with source at` s `*/`
2 **then**
3 | **return**;
4 **else**
5 | **if** $weight(s,v) < weight(s,u) + \mathbf{d}(u,v)$ **then**
6 | | Swap u and v;
7 | **end**
8 **end**
9 DIJKSTRA-UNDO$(\tau, PQ, weight(s,u) + \mathbf{d}(u,v))$;
10 Decrease the key of v to $weight(s,u) + \mathbf{d}(u,v)$ in PQ;
11 DIJKSTRA-BOUNDED(G, s, L, PQ, τ)

Algorithm 4.3: DIJKSTRA-BOUNDED(G, s, L, PQ, τ)

Input: Graph G, a vertex s, a real number L and a priority queue PQ.
Output: Shortest path length between s and all other vertices in G which has length at most L.
1 **while** *The key of the element on the top of* PQ *is at most* L **do**
2 | $u :=$ pop the element with minimum key from PQ;
3 | $weight(s,u) := weight(u,s) :=$ the key of u;
4 | **foreach** *node* v *adjacent to* u *in* G **do**
5 | | **if** $weight(s,u) + wt(u,v) < weight(s,v)$ **then**
6 | | | Decrease the key of v in PQ to $weight(s,u) + wt(u,v)$ and add all the changes in PQ to the stack τ ; `/* To be used in the undo procedure. */`
7 | | **end**
8 | **end**
9 **end**

4.1 Running time

Now, we show that Algorithm 4.1 runs in $\mathcal{O}(n^2 \log n)$ time. To this end, we show that for each point $p \in V$ the overall time spent is proportional to running Dijkstra's single-source shortest paths algorithm with source p.

Algorithm 4.1 basically performs Dijkstra's algorithm with source at each point of the graph. The only differences are

- it performs bounded Dijkstra's algorithm and
- it fixes the process after adding edge(s) to the graph by undoing some parts and redoing it.

The following lemma follows from [7]:

Lemma 4 *The value of l computed in line 11 of Algorithm 4.1 is $\mathcal{O}(n)$.*

Algorithm 4.4: DIJKSTRA-UNDO(τ, PQ, L)

Input: a stack τ, a priority queue PQ and a real number L.
1 **while** *the key of the element on the top of* τ *is at most* L **do**
2 | Pop the top element from τ;
3 | Undo the changes on PQ based on the information in the element;
4 **end**

Now we show that for any point p, the time we spend to update p in the whole process is proportional to the time for running Dijkstra's algorithm with source at p. Assume we are processing the point pairs in E_i and let G be the current graph. As one can see in Algorithm 4.1, when we process the pairs in E_i, for updating followed by adding an edge, we undo the execution of Dijkstra's algorithm until the key of the points on the top of the heap is less than the length of the new edge and redo the execution until the key of the point on the top of the heap is more than $2tL_i$. But the length of the new edge is at least L_i which means that the undo process never goes further when the key on the top of heap is less than L_i. We say that a vertex q is in the undo area of E_i if $L_i \leq \mathbf{d}_G(p,q) \leq 2tL_i$. The claim is that each point q appears in the undo area for a constant number of sets. To prove this let q be in the undo area of E_i. This means

$$L_i \leq \mathbf{d}_G(p,q) \leq 2tL_i. \tag{1}$$

We show that q can not be in the undo area of E_j if $j > i + \log t$. Let G' be the graph when we process the pairs in E_j. Since we add edge(s) to the graph, we know that for each pair (p,q) of points $\mathbf{d}_{G'}(p,q) \leq \mathbf{d}_G(p,q)$. Since for each i, $L_{i+1} > 2L_i$ we have

$$L_{i+k} > 2^k L_i, \quad \text{for each } k > 0.$$

Using Equation 1, we have

$$\mathbf{d}_{G'}(p,q) \leq \mathbf{d}_G(p,q) \leq 2tL_i < \frac{2t}{2^{\log t}} L_j \leq L_j,$$

which means q is not in the undo area of E_j.

On the other hand, by Lemma 2, we update the shortest paths with source p at most $\mathcal{O}(\frac{1}{(t-1)^{\mathcal{O}(\lambda)}})$ times in each set E_i. This means in total we need to recompute the shortest path between p and any point q at most $\mathcal{O}(\frac{\log t}{(t-1)^{\mathcal{O}(\lambda)}})$ times. Therefore the whole process for a fixed point p takes $\mathcal{O}(\frac{\log t}{(t-1)^{\mathcal{O}(\lambda)}} n \log n)$ time. So we have the following theorem.

Theorem 2 *For each point set V of n points from a metric space with doubling dimension λ, we can compute the greedy t-spanner of V in $\mathcal{O}(\frac{\log t}{(t-1)^{\mathcal{O}(\lambda)}} n^2 \log n)$ time using $\mathcal{O}(n^2)$ space.*

5 Conclusion

In this paper we presented an algorithm which, given a set of n points from a metric space with bounded doubling dimension, computes the greedy spanner of the point set in $\mathcal{O}(n^2 \log n)$ time. In the greedy spanner any point is connected to its nearest neighbor. Therefore, we can compute all nearest neighbors of the input point set using the greedy spanner in $\mathcal{O}(n)$ time. Har-Peled and Mendel [8] showed that all nearest neighbor problem has $\Omega(n^2)$ lower bound for metric spaces with bounded doubling dimension. This implies that computing greedy t-spanner also has $\Omega(n^2)$ lower bound. There is a logarithmic gap between the running time of the greedy algorithm presented in this paper and the lower bound.

References

[1] I. Althöfer, G. Das, D. P. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete and Computational Geometry*, 9(1):81–100, 1993.

[2] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM*, 42:67–90, 1995.

[3] L. P. Chew. There is a planar graph almost as good as the complete graph. In *SCG'86: Proceedings of the 2nd Annual ACM Symposium on Computational Geometry*, pages 169–177, 1986.

[4] G. Das and G. Narasimhan. A fast algorithm for constructing sparse Euclidean spanners. *International Journal of Computational Geometry and Applications*, 7:297–315, 1997.

[5] M. Farshi and J. Gudmundsson. Experimental study of geometric t-spanners. In *ESA'05: Proceedings of the 13th Annual European Symposium on Algorithms*, volume 3669 of *Lecture Notes in Computer Science*, pages 556–567. Springer-Verlag, 2005.

[6] M. Farshi and J. Gudmundsson. Experimental study of geometric t-spanners: A running time comparison. In *WEA'07: Proceedings of the 6th Workshop on Experimental Algorithms*, volume 4525 of *Lecture Notes in Computer Science*, pages 270–284. Springer-Verlag, 2007.

[7] S. Har-Peled. A simple proof?, 2006. http://valis.cs.uiuc.edu/blog/?p=441.

[8] S. Har-Peled and M. Mendel. Fast construction of nets in low-dimensional metrics and their applications. *SIAM Journal on Computing*, 35(5):1148–1184, 2006.

[9] G. Narasimhan and M. Smid. *Geometric spanner networks*. Cambridge University Press, 2007.

[10] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM, Philadelphia, PA, 2000.

[11] D. Peleg and A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13:99–116, 1989.

[12] D. Russel and L. J. Guibas. Exploring protein folding trajectories using geometric spanners. *Pacific Symposium on Biocomputing*, pages 40–51, 2005.

A Original greedy algorithm

Algorithm A.1: ORIGINAL GREEDY(V, t)

Input: point set V and $t > 1$.
Output: t-spanner $G = (V, E')$.

$E :=$ sorted list of pairs of points in V^2 by increasing distance; `/*ties are broken arbitrarily */`
$E' := \emptyset$;
$G := (V, E')$;
foreach $(u, v) \in E$; `/* in sorted order */`
do
  if SHORTESTPATH$(G, u, v) > t \cdot \mathbf{d}(u, v)$ **then** $E' := E' \cup \{(u, v)\}$;
end
return $G = (V, E')$;

B Improved greedy algorithm

Algorithm B.1: IMP. GREEDY(V, t)

Input: point set V and $t > 1$.
Output: t-spanner $G = (V, E')$.

foreach $(u, v) \in V^2$ **do** $weight(u, v) := \infty$;
$E :=$ sorted list of pairs of points in V^2 by increasing distance; `/*ties are broken arbitrarily */`
$E' := \emptyset$;
$G := (V, E')$;
foreach $(u, v) \in E$; `/* in sorted order */`
do
  if $weight(u, v) \leq t \cdot \mathbf{d}(u, v)$ **then**
    Discard (u, v);
  else
    Compute single-source shortest path with source u;
    foreach $w \in V$ **do** update $weight(u, w)$ and $weight(w, u)$;
    if $weight(u, v) \leq t \cdot \mathbf{d}(u, v)$ **then** Discard (u, v);
    else $E' := E' \cup \{(u, v)\}$;
  end
end
return $G(V, E')$;