Computing the greedy spanner in near-quadratic time[*]

Prosenjit Bose[†] Paz Carmi[†] Mohammad Farshi[†] Anil Maheshwari[†]
Michiel Smid[†]

Abstract

The greedy algorithm produces high-quality spanners and, therefore, is used in several applications. However, even 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 for a set of n points in a metric space with bounded doubling dimension in $\mathcal{O}(n^2 \log n)$ time. Since computing the greedy spanner has an $\Omega(n^2)$ lower bound, the time complexity of our algorithm is optimal 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. For example, in many applications, it is important to ensure a short 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 the very high costs associated with constructing such a network. This leads to the concept of a spanner, as defined below.

Let $(V,\ \mathbf{d})$ be a finite metric space and let $G(V, E)$ be a network on V such that the weight of each edge (u, v) of E is equal to the distance $\mathbf{d}(u, v)$ between its endpoints u and v. For any two points u and v in V, we denote by $\mathbf{d}_G(u, v)$ the weight of a path in G between u and v of minimum weight. For a real number $t > 1$, we say that G is a *t-spanner* of V if for each pair of points $u, v \in V$, we have $\mathbf{d}_G(u, v) \leq t \cdot \mathbf{d}(u, v)$. Any path in G between u and v having weight at most $t \cdot \mathbf{d}(u, v)$ is called a *t-path*. 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 [20] in the context of distributed computing, and by Chew [5] in the geometric context. Since then, spanners have received a lot of attention; see the survey papers [10, 14, 22] and the books [18, 19].

A classical algorithm for computing a t-spanner for any finite metric space $(V,\ \mathbf{d})$ and for any real number $t > 1$ is the *greedy algorithm*, proposed by Althöfer *et al.* [1] and, as mentioned in [1], independently by Bern in 1989. The main steps of this algorithm are the following (see Algorithm 1.1 for more details): First, sort all pairs of distinct points in V

[*]An extended abstract of this paper appeared in the Proceedings of the 11th Scandinavian Workshop on Algorithm Theory (SWAT), Lecture Notes in Computer Science Vol. 5124, pp. 390–401, Springer-Verlag, 2008.

[†]School of Computer Science, Carleton University, Ottawa, Ontario, K1S 5B6, Canada. Research supported in part by NSERC and MRI.
Email: `jit@scs.carleton.ca`, `{paz,mfarshi}@cg.scs.carleton.ca`, `{anil,michiel}@scs.carleton.ca`

in non-decreasing order of their distances, and initialize a graph G with vertex set V whose edge set is empty. Then, process the pairs 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 in G, then the edge (u, v) is added to G, otherwise this edge is discarded. We will refer to the graph G computed by this algorithm as the *greedy spanner*. The focus of this paper is to compute the greedy spanner efficiently.

Algorithm 1.1: Original-Greedy(V, t)

```
   Input: metric space $(V, \mathbf{d})$ and real number $t > 1$.
   Output: the greedy $t$-spanner $G(V, E)$.
1  $E'$ := list of all pairs of distinct points in $V$, sorted in non-decreasing order of their distances;
2  $E := \emptyset$;
3  $G := (V, E)$;
4  foreach $(u, v) \in E'$ (in sorted order) do
5      if $\mathbf{d}_G(u, v) > t \cdot \mathbf{d}(u, v)$ then
6          $E := E \cup \{(u, v)\}$;
7      end
8  end
9  return $G = (V, E)$;
```

The shortest-path length $\mathbf{d}_G(u, v)$ in line 5 can be obtained from a single-source shortest-path (SSSP) computation with source u. Recall that such a computation yields, for each point $w \in V$, the value $\mathbf{d}_G(u, w)$. Using Dijkstra's algorithm [9], an SSSP computation takes $\mathcal{O}(n \log n + m)$ time, where n is the number of vertices and m is the number of edges in G; see also [6, Section 24.3].

Thus, since the greedy algorithm performs $\binom{n}{2}$ shortest path queries, the time complexity of the entire algorithm is $\mathcal{O}(mn^2 + n^3 \log n)$, where n is the number of points in V and m is the number of edges in the (final) spanner G.

The greedy algorithm has been subject to considerable research [3, 4, 7, 8, 15, 23]. It has been shown that for any set V of n points in the Euclidean space $\mathbb{R}^d$ and for any fixed $t > 1$, the greedy spanner has $\mathcal{O}(n)$ edges, maximum degree $\mathcal{O}(1)$, and total weight $\mathcal{O}(wt(MST(V)))$, where $wt(MST(V))$ is the weight of a minimum spanning tree of V; see [8, 18]. Thus, in $\mathbb{R}^d$, the naïve implementation of the greedy algorithm runs in near-cubic time.

Due to the high time complexity of the greedy algorithm, researchers have proposed algorithms for computing other types of sparse t-spanners, see [18]. For Euclidean space $\mathbb{R}^d$, there are several algorithms that construct t-spanners with $\mathcal{O}(n)$ edges in $\mathcal{O}(n \log n)$ time. All these algorithms use geometric properties of the input point set.

However, based on the experiments on spanner algorithms with randomly generated point sets in the plane, the greedy algorithm produces t-spanners of higher quality in comparison to other $\mathcal{O}(n \log n)$ time spanner algorithms, like (ordered) Θ-graph algorithm, sink spanner, skip-list spanner and WSPD-based spanners, in practice; see [12, 13]. The experiments show that the greedy algorithm produces graphs whose size, weight, maximum degree and number of crossings are much lower than the graphs produced by the other approaches. For example, for $t = 1.1$, the number of edges in the greedy t-spanner of a set of 8000 uniformly distributed random points in the plane is approximately $36K$, when the number of edges in the Θ-graph, which has the lowest number of edges between the rest of studied algorithms, on the same point set is $370K$; see Table 6.3 of [11]. The maximum degree of the greedy 1.1-spanner, generated on the same set, is 14 and its weight is 11 times the weight of a minimum spanning tree of the point set. To have a rough comparison, the best maximum degree after the greedy spanner belongs to the ordered Θ-graph algorithm which produces graph with maximum degree 130; see Table 6.4 of [11]; and between other algorithms, the Θ-graph has lowest weight which is 327 times the weight of a minimum spanning tree; see Table 6.5 of [11].

In the geometric case, there is an algorithm with $\mathcal{O}(n \log n)$ running time, which *approximates* the greedy spanner; see [8, 15]. The graph generated by this approximate greedy algorithm has the same theoretical properties as the greedy spanner. The experiments showed, however, that the graphs generated by this approximation algorithm are much worse in practice; see [13]. To illustrate the difference, for $t = 1.1$ and on a set of 8000 uniformly distributed points in the plane, the approximate greedy algorithm generates a graph with 852K edges and maximum degree 403; see Table 6.3 and 6.4 of [11]. This is much higher than the size and maximum degree of the greedy spanner on the same point set.

Since low size and low weight spanners are important, the greedy spanner is used in several applications, despite its high time complexity. For example, it has 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; see [21]. The authors needed a low weight spanner that consists of short edges because the interactions in a protein are local. These local interactions make it difficult to assign biological meaning to long edges. Therefore, the greedy spanner is a suitable choice. Russel and Guibas used heuristics based on the A^*-search algorithm, which, in practice, improves the computation.

For points in the plane under the Euclidean metric, Farshi and Gudmundsson [12, 13] introduced a speed-up strategy that generates the greedy spanner much faster than the naïve implementation of the greedy algorithm in practice. For values of t that are close to 1, their algorithm runs even faster than the near-linear time algorithm which approximates the greedy t-spanner. For example, for constructing a 1.1-spanner on a set of 8000 uniformly distributed points, their fast greedy algorithm runs 3 times faster than the $\mathcal{O}(n \log^2 n)$ algorithm which approximates the greedy spanner. 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 when 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 in which the distance between any two distinct points is equal to 1. Then for any t with $1 < t < 2$, the complete graph is the only t-spanner of V. Therefore, for general metric spaces, we cannot guarantee that the greedy spanner is sparse. As we will show in this paper, however, if the metric space has bounded doubling dimension, then the number of edges in the greedy spanner is linear in the number of points. The doubling dimension of a metric space 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-dimensional Euclidean space is $\Theta(d)$.

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 has a linear number of edges and 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 greedy algorithm of [12, 13] which we refer to as the FG-greedy algorithm, and give a counterexample to the conjecture that this algorithm performs only $\mathcal{O}(n)$ SSSP computations. In fact, we show that this algorithm performs $\Omega(n^2)$ SSSP computations in the worst case. In Section 2.2, we modify the FG-greedy algorithm and show that the new algorithm performs $\Omega(n \log n)$ SSSP computations in the worst case. In Section 3.1, we present an algorithm that computes the greedy spanner in near-quadratic time for some special cases. These results are generalized to metric spaces of bounded doubling dimension in Section 3.2.

Throughout this paper, we assume that the (upper bound on the) stretch factor of the greedy spanner is a real number t such that $1 < t < 2$. For $t \geq 2$, one can construct a t'-spanner with $1 < t' < 2$ which is a t-spanner too.

2 The FG-greedy algorithm

As mentioned before, the running time of a naïve implementation of the greedy algorithm is $\mathcal{O}(mn^2 + n^3 \log n)$, where n is the number of points and m is the number of edges in the greedy spanner. Farshi and Gudmundsson [12, 13] introduced a variant of the greedy algorithm and showed that, in practice, it improves the running time for constructing the greedy spanner considerably on point sets in the plane with the Euclidean metric. We will refer to this algorithm as the FG-greedy algorithm. The FG-greedy algorithm is the same as the original greedy algorithm (Algorithm 1.1), except that it uses a matrix to store the length of the shortest path between every two points. The algorithm updates the matrix only when it is required. Thus, the weights in the matrix are not always equal to the actual shortest path lengths in the current graph. Instead of computing the shortest path length for each pair (u, v) (see line 5 of Algorithm 1.1), it first checks the matrix to see if there is a t-path between u and v. If the answer is "no", then it performs an SSSP computation and updates the matrix. Thus, the algorithm answers the distance queries correctly. The algorithm is presented below as Algorithm 2.1. Farshi and Gudmundsson conjectured that the FG-greedy algorithm performs only $\mathcal{O}(n)$ SSSP computations, which would imply a total running time of $\mathcal{O}(n^2 \log n)$ for the case when the greedy spanner has $\mathcal{O}(n)$ edges.

2.1 A Counterexample

We give an example which shows that the FG-greedy algorithm (Algorithm 2.1) performs $\Theta(n^2)$ SSSP computations in the worst-case, i.e., line 8 may be executed $\Theta(n^2)$ times.

Consider the set $S = \{p_0, p_1, \ldots, p_{n-1}\}$ of n points on the real line, where $p_i = 2^i$ for $0 \leq i < n$. The algorithm sorts all pairs of points based on their distances. We assume that for each pair (p_i, p_j) in the sorted list, the index of the first point in the pair is less than the index of the second point, i.e., $i < j$. The claim is that the algorithm performs an SSSP computation for each pair of points.

To show this, we split the sorted list of pairs into blocks B_i, $1 \leq i \leq n-1$, 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, then continues with the pairs in B_2, and so on. For arbitrary i, the first pair in B_i that the algorithm considers is (p_{i-1}, p_i). Since, at that moment, the point p_i is disconnected in the current graph from all other points, all entries in the weight matrix that involve p_i are ∞. Processing the pair (p_{i-1}, p_i) thus entails performing an SSSP computation with source p_{i-1}, updating all entries in the weight matrix that involve p_{i-1}, and then adding the edge (p_{i-1}, p_i) to the graph. Note that because the algorithm updates the row and the column in the weight matrix corresponding to p_{i-1}, the value of $weight(p_j, p_i)$ is still ∞ for all j with $j \leq i - 2$.

Algorithm 2.1: FG-Greedy(V, t)

```
Input: metric space $(V, \mathbf{d})$ and real number $t > 1$.
Output: the greedy $t$-spanner $G(V, E)$.
1  foreach $u \in V$ do  $weight(u, u) := 0$;
2  foreach $(u, v) \in V^2$ with $u \neq v$ do  $weight(u, v) := \infty$;
3  $E'$ := list of all pairs of distinct points in $V$, sorted in non-decreasing order of their distances;
4  $E := \emptyset$;
5  $G := (V, E)$;
6  foreach $(u, v) \in E'$ (in sorted order) do
7      if $weight(u, v) > t \cdot \mathbf{d}(u, v)$ then
8          perform an SSSP computation in $G$ with source $u$;
9          foreach $w \in V$ do
10             $weight(u, w) := weight(w, u) := \min(weight(u, w), \mathbf{d}_G(u, w))$;
11         end
12         if $weight(u, v) > t \cdot \mathbf{d}(u, v)$ then
13             $E := E \cup \{(u, v)\}$;
14         end
15     end
16 end
17 return $G(V, E)$;
```

The algorithm then processes (p_{i-2}, p_i). Because the entry for p_{i-2} and p_i in the matrix is ∞, the algorithm performs an SSSP computation with source p_{i-2}, and updates the row and column corresponding to p_{i-2}. Afterwards, $weight(p_j, p_i)$ is still ∞ for all j with $j \leq i - 3$. Continuing this argument shows that the algorithm performs an SSSP computation for each pair of points.

2.2 A variant of the FG-greedy algorithm

The FG-greedy algorithm does not update the weight matrix after adding an edge; see line 13 of Algorithm 2.1. The reader may ask what happens if after adding an edge, we perform two SSSP computations with sources at the endpoints of the new edge and update the weight matrix. Observe that this algorithm performs only $\mathcal{O}(n)$ SSSP computations on the counterexample in the previous section. In this section, we show that this variant makes $\Omega(n \log n)$ SSSP computations in the worse case, even in the one-dimensional Euclidean case.

Thus, we make the following modification to the FG-greedy algorithm: Each time the algorithm has just added an edge (u, v) to the greedy spanner, see line 13 in Algorithm 2.1, it performs one SSSP computation in the current graph with source u, one SSSP computation in the current graph with source v, and updates the rows and columns in the weight matrix that correspond to u and v.

Let n be a sufficiently large power of 2. We define (refer to Figure 1) $V_0 = \{0, 1\}$ and, for $i \geq 0$,

$$V_{i+1} = V_i \cup \left(V_i \oplus 3 \cdot 4^i\right),$$

where $V_i \oplus x = \{p + x : p \in V_i\}$. Thus, V_{i+1} is the union of V_i and a copy of V_i translated to the right by the amount of $3 \cdot 4^i$.

A straightforward induction proof shows that the set V_i consists of 2^{i+1} elements, $\min(V_i) = 0$, and $\max(V_i) = 4^i$.

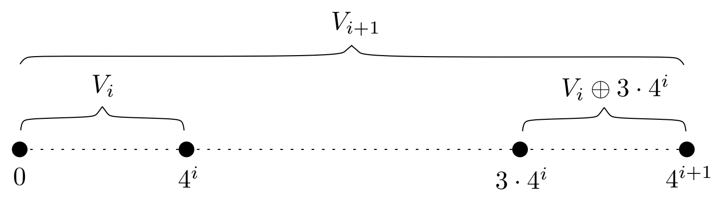

Figure 1: The set V_{i+1}.

Let $V = V_{\log n-1}$. Then V is a set of n points on the real line. We claim that the variant of the FG-greedy algorithm mentioned above performs $\Omega(n \log n)$ SSSP computations when it is run on the set V.

To prove this claim, observe that V is the union of $V_L := V_{\log n-2}$, which is contained in the interval $[0, n^2/16]$, and $V_R := V_{\log n-2} \oplus \frac{3}{16}n^2$, which is contained in the interval $[\frac{3}{16}n^2, n^2/4]$, and that $|V_L| = |V_R| = n/2$. We number the points of V_L in decreasing order as $l_1, l_2, \ldots, l_{n/2}$, and we number the points of V_R in increasing order as $r_1, r_2, \ldots, r_{n/2}$; see Figure 2.

$l_{n/2}$
l_1
V_L
r_1
$r_{n/2}$
V_R

Figure 2: The sets V_L and V_R.

The set of all pairs of distinct points in V can be split into three categories:

1. Pairs with both points in V_L.
2. Pairs with both points in V_R.
3. Pairs with one point in V_L and the other point in V_R.

Observe that the greedy algorithm processes all pairs in the first two categories before it processes any pair in the third category. We claim that the variant of the FG-greedy algorithm performs at least $n/2$ SSSP computations to process the pairs in the third category.

The first pair in the third category which the algorithm processes is (l_1, r_1). Since, at this moment, $\mathit{weight}(l_1, r_1) = \infty$, the algorithm performs an SSSP computation with source l_1, adds the edge (l_1, r_1) to the graph, performs two SSSP computations with sources l_1 and r_1, and updates the weight matrix. When processing (l_1, r_2), the algorithm does not perform an SSSP computation, because $\mathit{weight}(l_1, r_2)$ contains the correct shortest-path length between l_1 and r_2 in the current graph G. Similarly, when processing (l_2, r_1), the algorithm does not perform an SSSP computation. When processing (l_2, r_2), however, we have $\mathit{weight}(l_2, r_2) = \infty$ and, therefore, the algorithm performs one SSSP computation (observe that the edge (l_2, r_2) is not added to G). By repeating this argument, it follows that for each i with $1 \leq i \leq n/2$, the algorithm performs one SSSP computation when processing the pair (l_i, r_i).

If we denote by $N_{sp}(n)$ the number of SSSP computations performed by the algorithm on the point set V, then we have shown that

$$N_{sp}(n) = 2 \cdot N_{sp}(n/2) + n/2,$$

which implies that $N_{sp}(n) = \Omega(n \log n)$.

3 A near-quadratic greedy algorithm

In this section, we introduce an algorithm which computes the greedy spanner in near-quadratic time. In order to simplify the presentation, we start in Section 3.1 with presenting a simpler algorithm for which it is easier to prove that it computes the greedy spanner. This simpler algorithm runs in near-quadratic time in certain special cases. Then, in Section 3.2, we modify this algorithm in such a way that given a set of n points from a metric space with bounded doubling dimension, it computes the greedy spanner of the point set in $\mathcal{O}(n^2 \log n)$ time.

3.1 A preliminary algorithm

Let V be a set of n points in a metric space with distance function $\mathbf{d}$. Recall that the greedy t-spanner is obtained by starting with the graph $G(V, E = \emptyset)$, and then processing all pairs of distinct points in V in non-decreasing order of their distances. For each pair (u, v), we decide if there exists a t-path between u and v in G; if not, we add the edge (u, v) to E.

The new algorithm is similar to the FG-greedy algorithm (Algorithm 2.1) in the sense that, before doing an SSSP computation, it uses the weight matrix to decide if the currently processed edge has to be added to the graph. The new ingredients are the following:

- We choose a real number $L > 0$ and process the pairs (u, v) whose distances are less than L by performing an SSSP computation with source u.
- We divide the remaining pairs into buckets such that the i-th bucket contains all pairs whose distances are between $2^{i-1}L$ and 2^iL.
- We process the buckets one after another. When processing the pairs in the i-th bucket, we take care that, at any moment, $weight(u, v)$ is equal to the shortest-path distance between u and v in the current graph G, for all pairs (u, v) that are contained in the i-th bucket.

We assume without loss of generality that the diameter of the set V is equal to one. We fix a real number L with $0 < L < 1$, and partition the set of all pairs of distinct points in V into $l + 1 = \mathcal{O}(\log(1/L))$ buckets $E_0, E_1, \dots, E_l$, where E_0 contains all pairs with distance less than L and, for $1 \le i \le l$, the ith bucket E_i contains all pairs whose distances are in the interval $[2^{i-1}L, 2^iL)$.

The algorithm starts by processing the pairs in E_0. Each of these pairs (u, v) is processed by performing an SSSP computation with source u in the current graph G.

Assume that the algorithm has already processed all pairs in the buckets $E_0, E_1, \dots, E_{i-1}$. The pairs in bucket E_i are processed as follows: In a preprocessing step, we perform, for each point u in V, an SSSP computation with source u in the current graph G, and update the weight matrix. Thus, afterwards, we have $weight(u, v) = \mathbf{d}_G(u, v)$ for all pairs of points in V. Now the actual processing of bucket E_i starts. For each pair (u, v) in this bucket, we check if $weight(u, v) > t \cdot \mathbf{d}(u, v)$. If the answer is "yes", we add the edge (u, v) to the graph G and make "local" updates in the weight matrix in order to guarantee that all entries that correspond to pairs in E_i are equal to the shortest-path distance in the new graph G. By "local" updates we mean that we update the weight matrix for points that are not too far away from u or v. By these updates we make sure that for any pair of points (p, q) in the current bucket which both p and q are far away from u and v, either adding the new edge (u, v) does not change the shortest path length between p and q or, otherwise, the path between them which goes through u and v is not a t-path. More specifically, as we prove below, it is sufficient to run an SSSP computation with source p for each point $p \in V$ for which $\mathbf{d}(p, u) < (t - \frac{1}{2})2^{i-1}L$ or $\mathbf{d}(p, v) < (t - \frac{1}{2})2^{i-1}L$. A formal description of the algorithm is given in Algorithm 3.1.

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

```
   Input: metric space $(V, \mathbf{d})$ and real numbers $t > 1$ and $L > 0$.
   Output: the greedy $t$-spanner $G(V, E)$.
1  foreach $u \in V$ do  $weight(u, u) := 0$;
2  foreach $(u, v) \in V^2$ with $u \neq v$ do  $weight(u, v) := \infty$;
3  $E'$ := list of all pairs of distinct points in $V$, sorted in non-decreasing order of their distances;
4  $E_0$ := sorted list of all pairs in $E'$ whose distances are in $[0, L)$;
5  $i := 1$;
6  while $E' \setminus (\bigcup_{k=0}^{i-1} E_k) \neq \emptyset$ do
7      $E_i$ := sorted list of all pairs in $E' \setminus (\bigcup_{k=0}^{i-1} E_k)$ whose distances are in $[2^{i-1}L, 2^i L)$;
8      $i := i + 1$;
9  end
10 $l := i - 1$;
11 $E := \emptyset$;
12 $G := (V, E)$;
13 process the pairs in $E_0$ in the same way as in the original greedy algorithm;
14 for $i := 1, \dots, l$ do
15     $L_i := 2^{i-1}L$;
16     foreach $u \in V$ do
17         perform an SSSP computation in $G$ with source $u$ and update all entries in the weight
           matrix that correspond to $u$;
18     end
19     foreach $(u, v) \in E_i$ (in sorted order) do
20         if $weight(u, v) > t \cdot \mathbf{d}(u, v)$ then
21             $E := E \cup \{(u, v)\}$;
22             foreach $p \in V$ do
23                 if $\mathbf{d}(p, u) < (t - \frac{1}{2})L_i$ or $\mathbf{d}(p, v) < (t - \frac{1}{2})L_i$ then
24                     perform an SSSP computation in $G$ with source $p$ and update all entries in
                       the weight matrix that correspond to $p$;
25                 end
26             end
27         end
28     end
29 end
30 return $G(V, E)$;
```

Before we consider the running time of this algorithm, we prove that it computes the greedy spanner.

Lemma 1 *Algorithm 3.1 computes the greedy t-spanner of the input set V.*

Proof. It follows from line 20 in Algorithm 3.1 that it is sufficient to prove the following for each i with $1 \leq i \leq l$ and for each pair (p, q) in E_i: At the moment when the algorithm processes (p, q), we have $weight(p, q) > t \cdot \mathbf{d}(p, q)$ if and only if $\mathbf{d}_G(p, q) > t \cdot \mathbf{d}(p, q)$.

Let (p,q) be an arbitrary pair in E_i. Thus, $\mathbf{d}(p,q) \in [L_i, 2L_i)$. Assume that (p,q) is just about to be processed by the algorithm. Let G be the graph at this moment. Observe that, again at this moment, $weight(p,q) \geq \mathbf{d}_G(p,q)$. Therefore, if $\mathbf{d}_G(p,q) > t \cdot \mathbf{d}(p,q)$, then we have $weight(p,q) > t \cdot \mathbf{d}(p,q)$. We assume from now on that $\mathbf{d}_G(p,q) \leq t \cdot \mathbf{d}(p,q)$. Thus, we have to show that $weight(p,q) \leq t \cdot \mathbf{d}(p,q)$. We distinguish two cases.

Case 1: The shortest path between p and q in G does not contain any edge that has been added to G during the processing of pairs in E_i (prior to the processing of (p,q)).

In this case, it follows from line 17 in Algorithm 3.1 that $weight(p,q) = \mathbf{d}_G(p,q)$, which implies that $weight(p,q) \leq t \cdot \mathbf{d}(p,q)$.

Case 2: The shortest path π between p and q in G contains at least one edge of E_i.

Among all edges of $E_i \cap \pi$, let (u,v) be the one that was added last by the algorithm. We may assume without loss of generality that, when starting at p, the path π 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 and, therefore, $weight(p,q) \leq t \cdot \mathbf{d}(p,q)$.

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, we have

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

which contradicts our assumption that $\mathbf{d}_G(p,q) \leq t \cdot \mathbf{d}(p,q)$. □

3.1.1 The running time of Algorithm 3.1

Before we can analyze the running time of Algorithm 3.1, we recall the well-separated pair decomposition (WSPD) [2]. Consider the metric space $(V, \mathbf{d})$. For subsets A and B of V, we define

$$\mathbf{diam}(A) = \max\{\mathbf{d}(a,b) : a, b \in A\}$$

and

$$\mathbf{d}(A,B) = \min\{\mathbf{d}(a,b) : a \in A, b \in B\}.$$

Definition 1 *Let $s > 0$ be a real number, referred to as the* separation constant. *We say that two subsets A and B of V are s-*well-separated, *if*

$$\mathbf{d}(A,B) \geq s \cdot \max\{\mathbf{diam}(A), \mathbf{diam}(B)\}.$$

The following lemma follows from the definition above.

Lemma 2 *Let A and B be two subsets of V that are s-well-separated, let x and p be points of A, and let y and q be points of B. Then*

1. $\mathbf{d}(p, x) \leq (1/s) \cdot \mathbf{d}(p, q)$ *and*

2. $\mathbf{d}(x, y) \leq (1 + 2/s) \cdot \mathbf{d}(p, q)$.

Definition 2 *Consider the metric space $(V, \mathbf{d})$ and let $s > 1$ be a real number. A* well-separated pair decomposition *(WSPD) for V with respect to s is a set*

$$\{(A_1, B_1), \ldots, (A_m, B_m)\}$$

of pairs of non-empty subsets of V such that

1. *A_i and B_i are s-well-separated for all $i = 1, \ldots, m$, and*

2. *for any two distinct points p and q of V, there is exactly one pair (A_i, B_i) in the set, 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 m of pairs is called the *size* of the WSPD.

The WSPD was developed by Callahan and Kosaraju [2] for d-dimensional Euclidean space. They showed that for any set V of n points in $\mathbb{R}^d$, a WSPD of size $m = \mathcal{O}(s^d n)$ exists. Talwar [24] transferred the definition to an arbitrary metric space and proved that any set of n points from a metric space with doubling dimension d admits a WSPD of size $s^{\mathcal{O}(d)} n \log \alpha$, where α is the aspect ratio of the point set. Har-Peled and Mendel [17] improved the size in the latter result to $s^{\mathcal{O}(d)} n$.

Observation 1 *Let A and B be two subsets of V that are s-well-separated for $s = \frac{2t}{t-1}$. The greedy t-spanner contains at most one edge between A and B.*

Proof. Assume that the greedy t-spanner contains two edges (a_1, b_1) and (a_2, b_2), where $a_1, a_2 \in A$ and $b_1, b_2 \in B$. We may assume without loss of generality that the greedy algorithm processes the pair (a_1, b_1) before the pair (a_2, b_2). Thus, we have $\mathbf{d}(a_1, b_1) \leq \mathbf{d}(a_2, b_2)$.

Since A and B are s-well-separated, it follows from Lemma 2 that

$$\mathbf{d}(a_1, a_2) \leq \frac{1}{s} \cdot \mathbf{d}(a_2, b_2) < \mathbf{d}(a_2, b_2),$$

and

$$\mathbf{d}(b_1, b_2) \leq \frac{1}{s} \cdot \mathbf{d}(a_2, b_2) < \mathbf{d}(a_2, b_2).$$

Let G be the graph just before the pair (a_2, b_2) is processed by the greedy algorithm. This graph contains (i) a t-path between a_1 and a_2, (ii) the edge (a_1, b_1), and (iii) a t-path between b_1 and b_2. This, together with Lemma 2, implies that

$$\begin{aligned} \mathbf{d}_G(a_2, b_2) &\leq \mathbf{d}_G(a_2, a_1) + \mathbf{d}(a_1, b_1) + \mathbf{d}_G(b_1, b_2) \\ &\leq t \cdot \mathbf{d}(a_2, a_1) + \mathbf{d}(a_1, b_1) + t \cdot \mathbf{d}(b_1, b_2) \\ &\leq \frac{t}{s} \cdot \mathbf{d}(a_2, b_2) + \mathbf{d}(a_2, b_2) + \frac{t}{s} \cdot \mathbf{d}(a_2, b_2) \\ &= t \cdot \mathbf{d}(a_2, b_2). \end{aligned}$$

Thus, the greedy algorithm does not add (a_2, b_2) as an edge to the spanner, which is a contradiction. $\square$

By combining Observation 1 and the result of Har-Peled and Mendel [17], we obtain the following result:

Corollary 1 *For every metric space V with doubling dimension d, and for every real number $1 < t < 2$, the greedy t-spanner contains $\frac{1}{(t-1)^{\mathcal{O}(d)}} n$ edges, where $n = |V|$.*

In the rest of the paper, we assume that V is a set of n points from a metric space with doubling dimension d.

Lemma 3 *Consider the variable l that is computed in line 10 of Algorithm 3.1. Let i be an integer with $1 \le i \le l$, and let p be a point of V. During the processing of the pairs in E_i, the number of times that line 24 in Algorithm 3.1 is executed for p is at most*

$$\left(\frac{(4t+6)(4t-2)}{t-1}\right)^{2d} = \frac{1}{(t-1)^{\mathcal{O}(d)}}.$$

Proof. Recall from the algorithm that for each pair (u, v) in E_i, $\mathbf{d}(u, v)$ is in the interval $[L_i, 2L_i)$. Let B be the ball with center p and radius $(t + \frac{3}{2})L_i$. The algorithm performs an SSSP computation with source p, each time an edge (u, v) is added to the graph for which $\mathbf{d}(p, u) < (t - \frac{1}{2})L_i$ or $\mathbf{d}(p, v) < (t - \frac{1}{2})L_i$. Since $\mathbf{d}(u, v) < 2L_i$, it follows that both u and v are contained in B. Thus, the number of times that line 24 is executed for the point p (during the processing of E_i) is bounded from above by the number of edges in the greedy t-spanner whose lengths are in the interval $[L_i, 2L_i)$ and both of whose endpoints are contained in B.

Let $R = (t + \frac{3}{2})L_i$ and

$$k = \left\lceil \log\left(\frac{(4t+6)(2t-1)}{t-1}\right)\right\rceil.$$

Observe that $2^k \ge \frac{(4t+6)(2t-1)}{t-1}$. By repeatedly applying the definition of doubling dimension, we can cover the ball B by 2^{kd} balls $B_1, B_2, \ldots, B_{2^{kd}}$ of radius $R/2^k$.

Let (u, v) be an edge in the greedy t-spanner such that $\mathbf{d}(u, v) \in [L_i, 2L_i)$, $u \in B$, and $v \in B$. We may assume without loss of generality that $u \in B_1$ and $v \in B_2$. We have

$$\mathbf{diam}(B_1) \le R/2^{k-1} \le \frac{R(t-1)}{(2t+3)(2t-1)} = \frac{t-1}{4t-2} L_i$$

and

$$\mathbf{diam}(B_2) \le \frac{t-1}{4t-2} L_i.$$

Also,

$$\mathbf{d}(B_1, B_2) \ge \mathbf{d}(u, v) - 4R/2^k \ge L_i - \frac{R(t-1)}{(t+\frac{3}{2})(2t-1)} = \frac{t}{2t-1} L_i.$$

By combining these inequalities, it follows that

$$\mathbf{d}(B_1, B_2) \ge \frac{2t}{t-1} \cdot \max\{\mathbf{diam}(B_1), \mathbf{diam}(B_2)\},$$

i.e., the balls B_1 and B_2 are s-well-separated for $s = \frac{2t}{t-1}$. Thus, by Observation 1, (u, v) is the only edge in the greedy t-spanner such that $\mathbf{d}(u, v) \in [L_i, 2L_i)$, $u \in B_1$, and $v \in B_2$.

It follows that the number of edges in the greedy t-spanner whose lengths are in $[L_i, 2L_i)$ and both of whose endpoints are contained in B is at most $\left(2^{kd}\right)^2$, which is $\frac{1}{(t-1)^{\mathcal{O}(d)}}$. $\square$

Now we are ready to estimate the time complexity of Algorithm 3.1. Clearly lines 1–12 take $\mathcal{O}(n^2 \log n)$ time. Let β be the number of pairs in E_0 and let m be the number of edges in the greedy t-spanner. Then line 13 takes $\mathcal{O}(\beta(m + n \log n))$ time, because for each pair in E_0, the algorithm performs an SSSP computation.

For each of the $\mathcal{O}(\log(1/L))$ sets E_i, lines 16–17 take $\mathcal{O}(mn + n^2 \log n)$ time. In lines 19–28 we process all the pairs in E_i and when we add an edge to the graph, we check all the points and update the row and column corresponding to them in the weight matrix, if necessary. By Lemma 3, for a fixed point p we update the weight matrix at most $\frac{1}{(t-1)^{\mathcal{O}(d)}}$ times which means we spend $\frac{1}{(t-1)^{\mathcal{O}(d)}}(mn + n^2 \log n)$ time for updating the weight matrix during processing the pairs in E_i. Therefore lines 19–28 take $\frac{1}{(t-1)^{\mathcal{O}(d)}}(mn + n^2 \log n)$ time.

Since, by Corollary 1, $m = \frac{1}{(t-1)^{\mathcal{O}(d)}} n$, the overall running time of the algorithm is

$$\beta \left(\frac{n}{(t-1)^{\mathcal{O}(d)}} + \mathcal{O}(n \log n) \right) + \frac{\log(1/L)}{(t-1)^{\mathcal{O}(d)}} n^2 \log n.$$

Recall that we assumed that the diameter of V is equal to one, and that β is the number of pair-wise distances in V that are less than L. If there exists a real number L such that $1/L$ is polynomial in n and β is near-linear in n, then the running time of Algorithm 3.1 is near-quadratic.

3.2 The final algorithm

In this section, we show how the approach of the previous section can be modified such that for any metric space of bounded doubling dimension, the greedy spanner can be computed in $\mathcal{O}(n^2 \log n)$ time.

Before we present the details, we recall Dijkstra's SSSP algorithm. Let G be an edge-weighted graph and let u be a vertex of G. Dijkstra's algorithm computes the shortest path-distance in G between u and each vertex of G. For each vertex v, the algorithm maintains a tentative distance $\textit{tent\_dist}(v)$, whose value is the length of the shortest path between u and v found so far. Initially, $\textit{tent\_dist}(u) = 0$ and $\textit{tent\_dist}(v) = \infty$ for all $v \neq u$. The vertices v of G for which $\mathbf{d}_G(u, v)$ has not been determined yet are maintained in a priority queue PQ, where the key of each such v is the value $\textit{tent\_dist}(v)$. This priority queue can be implemented either as a heap or as a Fibonacci heap.

In one iteration, the algorithm takes the vertex v in PQ whose key is minimum. It is well-known that, at this moment, the value of $\textit{tent\_dist}(v)$ is equal to $\mathbf{d}_G(u, v)$ and, thus, v can be deleted from PQ. The algorithm considers all edges (v, w) with $w \in PQ$, sets

$$\textit{tent\_dist}(w) = \min(\textit{tent\_dist}(w), \textit{tent\_dist}(v) + \mathbf{d}(v, w)),$$

and, in case $\textit{tent\_dist}(w)$ has a new value now, updates PQ to reflect the decrease in value of the key of w. The algorithm terminates as soon as the priority queue is empty.

Dijkstra's algorithm with source u computes the sequence of all shortest-path distances $\mathbf{d}_G(u,v)$ in non-decreasing order of their values. This implies that, given a real number $L > 0$, we obtain all values $\mathbf{d}_G(u,v)$ which are at most L, by running Dijkstra's algorithm with source u and terminating as soon as the minimum key in PQ is larger than L. We will refer to the modification algorithm as the *bounded Dijkstra's algorithm* with source u and distance L.

Our final greedy spanner algorithm uses the following ingredients:

- We partition the $\binom{n}{2}$ pairs of distinct points in V into buckets, such that within each bucket, distances differ by at most a factor of two. (As shown in [16], $\mathcal{O}(n)$ buckets are sufficient for any metric space.)

- We process the buckets one after another. Consider the current bucket containing all pairs whose distances are in the interval $[L, 2L)$. For each point u of V, we maintain a stack storing all operations performed by the bounded Dijkstra's algorithm with source u and distance $2tL$. Thus, for each vertex v such that $\mathbf{d}_G(u,v) \leq 2tL$, we know the value of $\mathbf{d}_G(u,v)$, which is stored as $weight(u,v)$ in the distance matrix. When we add an edge (u,v) to the greedy spanner, we take all points p for which $\mathbf{d}(p,u) < (t-\frac{1}{2})L$ or $\mathbf{d}(p,v) < (t-\frac{1}{2})L$. Instead of running the bounded Dijkstra's algorithm with source p and distance $2tL$ from scratch (as we did in Algorithm 3.1), we do the following: We use the stack stored with p to *undo* the execution of the bounded Dijkstra's algorithm (in the graph prior to the insertion of the edge (u,v)) until the minimum key in the priority queue is at most $\min((t-\frac{1}{2})L, L)$. Then, we restart Dijkstra's algorithm from this state, using the graph that contains the new edge (u,v), and terminate as soon as the minimum key in the priority queue is larger than $2tL$; during the execution, we store the sequence of all operations in the stack associated with p.

Consider again the bucket containing all pairs whose distances are in the interval $[L, 2L)$. Why is it sufficient to run the bounded Dijkstra's algorithm with length $2tL$? Assume $\mathbf{d}(p,q)$ is in $[L, 2L)$ and consider the moment when the algorithm processes the pair (p,q). Obviously, if $\mathbf{d}_G(p,q) \geq 2tL$, then $\mathbf{d}_G(p,q) > t \cdot \mathbf{d}(p,q)$. As a result, it is sufficient in this case to have a value $weight(p,q)$ which is equal to the shortest-path distance between p and q in an old version of the graph (see also the proof of Lemma 1). This value $weight(p,q)$ will allow us to make the correct decision of not adding (p,q) to the greedy spanner.

A detailed description of the algorithm is given in Algorithms 3.2–3.4.

Lemma 4 *Algorithm 3.2 computes the greedy t-spanner of the input set V.*

Proof. Let i be an integer with $1 \leq i \leq l$ and consider the iteration of the algorithm when the edges of E_i are processed. Lines 18–23 guarantee that the weight matrix stores all shortest-path distances in the current graph G that are at most $2tL_i$. Since all distances in E_i are less than $2L_i$, there is no need to compute shortest-path distances that are larger than $2tL_i$.

Let (u,v) be a pair in E_i and assume that the algorithm adds the edge (u,v) to the graph. Let G be the graph prior to the addition of this edge, and let G' denote the graph just after this edge has been added. The algorithm considers all points p in V for which $\mathbf{d}(p,u) < (t-\frac{1}{2})L_i$ or $\mathbf{d}(p,v) < (t-\frac{1}{2})L_i$. We have seen in the proof of Lemma 1 that it is sufficient to consider only these points. Recall that Algorithm 3.1 performs an SSSP computation in G' with source p. We have to show that lines 29-40 have the same effect (up to shortest-path distances that are at most $2tL_i$). If neither of the conditions in lines 29 and 35

Algorithm 3.2: New-Greedy(V, t)

```
Input: metric space $(V, \mathbf{d})$ and real number $t > 1$.
Output: the greedy $t$-spanner $G(V, E)$.
foreach $u \in V$ do  $weight(u, u) := 0$;
foreach $(u, v) \in V^2$ with $u \neq v$ do  $weight(u, v) := \infty$;
$E'$ := list of all pairs of distinct points in $V$, sorted in non-decreasing order of their distances;
$i := 1$;
while $E' \setminus (\bigcup_{k=1}^{i-1} E_k) \neq \emptyset$ do
    $L_i$ := distance of the shortest pair in $E' \setminus (\bigcup_{k=1}^{i-1} E_k)$;
    $E_i$ := sorted list of all pairs in $E' \setminus (\bigcup_{k=1}^{i-1} E_k)$ whose distances are in $[L_i, 2L_i)$;
    $i := i + 1$;
end
$l := i - 1$;
$E := \emptyset$;
$G := (V, E)$;
foreach $u \in V$ do
    $PQ_u$ := priority queue storing all $v \in V$ with key $weight(u, v)$;
    $\tau_u$ := empty stack;
end
for $i := 1, \ldots, l$ do
    foreach $u \in V$ do
        if $i > 1$ then
            Dijkstra-Undo$(\tau_u, PQ_u, (t - \frac{1}{2})L_{i-1})$;
        end
        Dijkstra-Bounded$(G, u, 2tL_i, PQ_u, \tau_u)$;
    end
    foreach $(u, v) \in E_i$ (in sorted order) do
        if $weight(u, v) > t \cdot \mathbf{d}(u, v)$  and $weight(v, u) > t \cdot \mathbf{d}(u, v)$ then
            $E := E \cup \{(u, v)\}$;
            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
                    if $weight(p, u) + \mathbf{d}(u, v) < weight(p, v)$ then
                        Dijkstra-Undo$(\tau_p, PQ_p, \min((t - \frac{1}{2})L_i, L_i))$;
                        in $PQ_p$, decrease the key of $v$ to $weight(p, u) + \mathbf{d}(u, v)$;
                        $weight(p, v) := weight(p, u) + \mathbf{d}(u, v)$;
                        Dijkstra-Bounded$(G, p, 2tL_i, PQ_p, \tau_p)$
                    end
                    if $weight(p, v) + \mathbf{d}(u, v) < weight(p, u)$ then
                        Dijkstra-Undo$(\tau_p, PQ_p, \min((t - \frac{1}{2})L_i, L_i))$;
                        in $PQ_p$, decrease the key of $u$ to $weight(p, v) + \mathbf{d}(u, v)$;
                        $weight(p, u) := weight(p, v) + \mathbf{d}(u, v)$;
                        Dijkstra-Bounded$(G, p, 2tL_i, PQ_p, \tau_p)$
                    end
                end
            end
        end
    end
end
return $G(V, E)$;
```

Algorithm 3.3: DIJKSTRA-BOUNDED(G, s, L, PQ, τ)

```
Input: graph $G$, vertex $s$, real number $L > 0$, priority queue $PQ$, stack $\tau$.
Output: using $PQ$, continue Dijkstra's algorithm with source $s$ until all shortest-path
        distances in $G$ which are at most $L$ have been computed; the algorithm stores all
        operations in $\tau$ (the pseudocode does not explicitly mention this).
1 while the minimum key in $PQ$ is at most $L$ do
2     delete the element $u$ with minimum key from $PQ$;
3     foreach node $v$ adjacent to $u$ in $G$ do
4         if $weight(s,u) + \mathbf{d}(u,v) < weight(s,v)$ then
5             in $PQ$, decrease the key of $v$ to $weight(s,u) + \mathbf{d}(u,v)$;
6             $weight(s,v) := weight(s,u) + \mathbf{d}(u,v)$
7         end
8     end
9 end
```

Algorithm 3.4: DIJKSTRA-UNDO(τ, PQ, L)

```
Input: stack $\tau$, priority queue $PQ$, real number $L > 0$.
1 while the minimum key in $PQ$ is larger than $L$ do
2     pop the top element $c$ from $\tau$;
3     undo the changes based on the information in $c$;
4 end
```

hold, then the addition of (u, v) does not change the behavior of Dijkstra's algorithm with source p up to shortest-path distances that are at most $2tL_i$. Assume that the condition in line 29 holds. Then the first time that Dijkstra's algorithm with source p behaves differently on G and G' is the moment when v is the element with the minimum key in the corresponding priority queue. Therefore, it is sufficient to undo Dijkstra's algorithm on G up to the distance $\min((t - \frac{1}{2})L_i, L_i)$, decrease the key in p's priority queue to $weight(p, u) + \mathbf{d}(u, v)$, and continue Dijkstra's algorithm with G' up to the distance $2tL_i$. This is exactly what Algorithm 3.2 does. $\square$

3.2.1 The running time of Algorithm 3.2

In this section, we show that Algorithm 3.2 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 for p is proportional to the time for running Dijkstra's SSSP algorithm with source p on the entire greedy spanner (which, using Corollary 1, is $\mathcal{O}(n \log n)$). Recall that we assume that the value of t is close to one. In particular, we have $t < 2$.

Recall that Dijkstra's algorithm on a graph G with source p computes shortest-path distances $\mathbf{d}_G(p, q)$ (for $q \in V$) in non-decreasing order of their values. For real numbers $L' > L > 0$, the portion of Dijkstra's algorithm *in the interval* $[L, L')$ is defined to be the part of the computation in which all shortest-path distances $\mathbf{d}_G(p, q)$ are computed that satisfy $L \leq \mathbf{d}_G(p, q) < L'$.

We fix a point p in V. Consider the iteration in which the algorithm processes the pairs in E_i. Let (u, v) be a pair in E_i that is added as an edge to the greedy spanner, and assume that the condition in line 28 holds. Also, assume that one of the conditions in lines 29 and 35 holds, w.l.o.g. we may assume the one in line 29. The algorithm calls Dijkstra-Undo, which runs Dijkstra's algorithm backwards as long as the minimum key in the priority queue PQ_p of p is at least $\min((t-\frac{1}{2})L_i, L_i)$, which is at least $\frac{1}{2}L_i$. Then, the algorithm calls Dijkstra-Bounded, which continues Dijkstra's algorithm as long as the minimum key in PQ_p is at most $2tL_i$, which is less than $4L_i$. Thus, when the edge (u,v) is added, the time spent for p is at most twice the time spent by Dijkstra's algorithm with source p in the interval $\frac{1}{2}L_i, 4L_i)$ (once backwards and once forwards). By Lemma 3, the number of times that this happens for p, during the processing of E_i, is $\frac{1}{(t-1)^{\mathcal{O}(d)}}$.

It follows from the algorithm that $L_i \geq 2L_{i-1}$. This implies that, over the entire algorithm and for the point p, Dijkstra's algorithm with source p in the interval $[L_i, 2L_i)$ is run $\frac{1}{(t-1)^{\mathcal{O}(d)}}$ times. During the course of the algorithm, edges are added to the graph. Therefore, the total time spent for p is $\frac{1}{(t-1)^{\mathcal{O}(d)}}$ times the time for one complete SSSP computation with source p in the final greedy spanner. Since, by Corollary 1, this spanner has $\frac{1}{(t-1)^{\mathcal{O}(d)}}n$ edges, it follows that the total time spent for point p is $\frac{1}{(t-1)^{\mathcal{O}(d)}}n\log n$.

To complete the proof of the running time of Algorithm 3.2, we need the following lemma, which gives an upper bound on the number of buckets E_i:

Lemma 5 *The value of l computed in line 10 of Algorithm 3.2 is $\mathcal{O}(n)$.*

Proof. The proof follows from the fact that, for a metric space of bounded doubling dimension, a well-separated pair decomposition with $\mathcal{O}(n)$ pairs exists; see [17]. For each pair (A, B) in the well-separated pair decomposition, assume (p, q) is the closest pair such that $p \in A$ and $q \in B$. Assume $(p, q) \in E_i$ which means $\mathbf{d}(p, q)$ is in interval $[L_i, 2L_i)$. By Lemma 2, for any pair (x, y) with $x \in A$ and $y \in B$, by choosing $s = 2$, we have $\mathbf{d}(x, y) \leq 2 \cdot \mathbf{d}(p, q) < 4L_i$ which means the pair (x, y) is in E_i or E_{i+1}. Therefore the number of buckets is at most twice the number of pairs in the well-separated pair decomposition which is linear.

In fact, the lemma holds for any metric space; see [16]. □

This lemma implies that the time spent by the algorithm, besides the shortest-path computations, is $\mathcal{O}(n^2)$. We have proved the main result of this paper:

Theorem 1 *Let $(V, \mathbf{d})$ be a metric space of size n having doubling dimension d and let $1 < t < 2$ be a real number. The greedy t-spanner of V can be computed in $\frac{1}{(t-1)^{\mathcal{O}(d)}}n^2\log n$ time.*

4 Conclusion

We have presented an algorithm which, when given a set V of n points from a metric space of bounded doubling dimension, computes the greedy spanner of V in $\mathcal{O}(n^2 \log n)$ time. Observe that in the greedy spanner, every point is connected to its nearest neighbor in V. Therefore, given the greedy spanner, we can solve the all-nearest-neighbors problem on V in $\mathcal{O}(n)$ time. Har-Peled and Mendel [17] have shown that the latter problem has an $\Omega(n^2)$ lower bound for metric spaces of bounded doubling dimension. This implies that computing the greedy spanner also has an $\Omega(n^2)$ lower bound. We leave open the problem of closing the logarithmic gap between the running time of our algorithm and this lower bound.

The algorithms proposed in this paper have quadratic space complexities. An interesting open problem, especially from a practical point of view, is to improve the space complexity of the greedy algorithm, without significantly increasing the running time. Another open problem is to decide whether the greedy spanner can be computed in $o(n^2)$ time for point sets in Euclidean space $\mathbb{R}^d$. Finally, consider an arbitrary metric space of size n. Is it possible to compute the greedy spanner in $o(mn^2)$ time, where m denotes the number of edges in the spanner?

5 Acknowledgements

The authors would like to thank Otfried Cheong and Aram Han for pointing out some mistakes in Algorithm 3.2.

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] B. Chandra. Constructing sparse spanners for most graphs in higher dimensions. *Information Processing Letters*, 51(6):289–294, 1994.

[4] B. Chandra, G. Das, G. Narasimhan, and J. Soares. New sparseness results on graph spanners. *International Journal of Computational Geometry and Applications*, 5:124–144, 1995.

[5] 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.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press and McGraw-Hill, 2nd edition, 2001.

[7] G. Das, P. J. Heffernan, and G. Narasimhan. Optimally sparse spanners in 3-dimensional Euclidean space. In *SCG'93: Proceedings of the 9th Annual ACM Symposium on Computational Geometry*, pages 53–62, 1993.

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

[9] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

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

[11] M. Farshi. *A Theoretical and Experimental Study of Geometric Networks.* Ph.D. thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 2008.

[12] 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.

[13] 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.

[14] J. Gudmundsson and C. Knauer. Dilation and detour in geometric networks. In T. Gonzalez, editor, *Handbook on approximation algorithms and metaheuristics.* Chapman & Hall/CRC, Amsterdam, 2007.

[15] J. Gudmundsson, C. Levcopoulos, and G. Narasimhan. Improved greedy algorithms for constructing sparse geometric spanners. *SIAM Journal on Computing*, 31(5):1479–1500, 2002.

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

[17] 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.

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

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

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

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

[22] M. Smid. Closest point problems in computational geometry. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 877–935. Elsevier Science Publishers, Amsterdam, 2000.

[23] J. Soares. Approximating Euclidean distances by small degree graphs. *Discrete and Computational Geometry*, 11:213–233, 1994.

[24] K. Talwar. Bypassing the embedding: algorithms for low dimensional metrics. In *STOC'04: Proceedings of the 36th Annual ACM Symposium on Theory of Computing*, pages 281–290, New York, NY, USA, 2004. ACM.