

# Computing the Greedy Spanner in Linear Space

Sander P. A. Alewijnse<sup>1</sup> · Quirijn W. Bouts<sup>1</sup>   
Alex P. ten Brink<sup>1</sup> · Kevin Buchin<sup>1</sup>

Received: 31 December 2013 / Accepted: 4 April 2015 / Published online: 20 April 2015  
© The Author(s) 2015. This article is published with open access at Springerlink.com

**Abstract** The greedy spanner is a high-quality spanner: its total weight, edge count and maximal degree are asymptotically optimal and in practice significantly better than for any other spanner with reasonable construction time. Unfortunately, all known algorithms that compute the greedy spanner on  $n$  points use  $\Omega(n^2)$  space, which is impractical on large instances. To the best of our knowledge, the largest instance for which the greedy spanner was computed so far has about 13,000 vertices. We present a linear-space algorithm that computes the same spanner for points in  $\mathbb{R}^d$  running in  $O(n^2 \log^2 n)$  time for any fixed stretch factor and dimension. We discuss and evaluate a number of optimizations to its running time, which allowed us to compute the greedy spanner on a graph with a million vertices. To our knowledge, this is also the first algorithm for the greedy spanner with a near-quadratic running time guarantee that has actually been implemented.

**Keywords** Geometric spanner · Dilation · Stretch factor · Greedy algorithm · Computational geometry

## 1 Introduction

A  $t$ -spanner on a set of points, usually in the Euclidean plane, is a graph on these points that is a ‘ $t$ -approximation’ of the complete graph, in the sense that shortest routes in the graph are at most  $t$  times longer than the direct geometric distance. The spanners

---

✉ Quirijn W. Bouts  
q.w.bouts@tue.nl

Kevin Buchin  
k.a.buchin@tue.nl

<sup>1</sup> Eindhoven University of Technology, Eindhoven, The Netherlands

considered in literature have only  $O(n)$  edges as opposed to the  $O(n^2)$  edges in the complete graph, or other desirable properties such as bounded diameter or bounded degree, which makes them a lot more pleasant to work with than the complete graph.

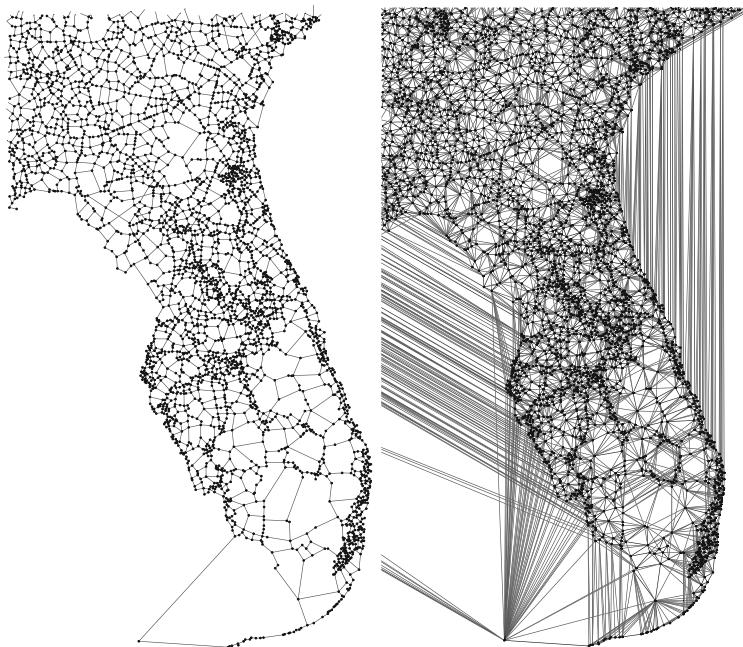
Spanners are used in wireless network design [8]: for example, high-degree routing points in such networks tend to have problems with interference, so using a spanner with bounded degree as network avoids these problems while maintaining connectivity. They are also used as a component in various other geometric algorithms, and are used in distributed algorithms. Spanners were introduced in network design [13] and geometry [6], and have since been subject to a considerable amount of research [10, 12].

There exists a large number of constructions of  $t$ -spanners that can be parameterized with arbitrary  $t > 1$ . They have different strengths and weaknesses: some are fast to construct but of low quality ( $\Theta$ -graph, which has no guarantees on its total weight), others are slow to construct but of high quality (greedy spanner, which has low total weight and maximum degree), some have an extremely low diameter (various dumbbell based constructions) and some are fast to construct in higher dimensions (well-separated pair decomposition spanners). See for example [12] for detailed expositions of these spanners and their properties.

The greedy spanner is one of the first spanner algorithms that was considered, and it has been subject to a considerable amount of research regarding its properties and more recently also regarding computing it efficiently. This line of research resulted in a  $O(n^2 \log n)$  algorithm [2] for metric spaces of bounded doubling dimension (and therefore also for Euclidean spaces). A different algorithm which has proven to work well in practice is the so called FG-Greedy algorithm introduced in [7]. Although a  $\Theta(n^3 \log n)$  bound was proven for this algorithm, experiments show that it tends to have near-quadratic behaviour in practice.<sup>1</sup> Among the many spanner algorithms known, the greedy spanner is of special interest because of its exceptional quality: its size, weight and degree are asymptotically optimal, and also in practice better than those of any other spanner construction algorithms with reasonable running times. For example, it produces spanners with about ten times as few edges, twenty times smaller total weight and six times smaller maximum degree than its closest well-known competitor, the  $\Theta$ -graph, on uniform pointsets. The contrast is clear in Fig. 1. Therefore, a method of computing it more efficiently is of considerable interest.

We present an algorithm whose space usage is  $\Theta(n)$  whereas existing algorithms use  $\Theta(n^2)$  space, while being only a logarithmic factor slower than the fastest known algorithm, thus answering a question left open in [2]. Our algorithm makes the greedy spanner practical to compute for much larger inputs than before: previously instances of up to 13,000 vertices have been calculated at which point these algorithms already used multiple gigabytes of memory. In contrast, we tested our algorithm on instances of up to 1,000,000 points using less than 8 GB of memory for the largest set. For previous algorithms calculations on these large pointsets would require multiple terabytes of memory. Furthermore, with the help of several optimizations we will present, the algorithm is also fast in practice, as our experiments show.

<sup>1</sup> In subsequent work [3] we have shown that a bound of  $O(n^2 \log^2 n)$  on the running time can be achieved when restricted to pointsets with polynomial spread, explaining this good performance in practice.



**Fig. 1** The *left rendering* shows the greedy spanner on the USA, zoomed in on Florida, with  $t = 2$ . The dataset has 115,475 vertices, so it was infeasible to compute this graph before. The *right rendering* shows the  $\Theta$ -graph on the USA, zoomed in on Florida, with  $k = 6$  for which it was recently proven it achieves a dilation of two

The method used to achieve this consists of two parts: a framework that uses linear space and near-linear time, and a subroutine using linear space and (amortized) near-linear time, which is called a near-linear number of times by the framework. The subroutine solves the *bichromatic closest pair with dilation larger than  $t$*  problem. If there is an algorithm with a sub-linear running time for this subproblem (possibly tailored to our specific scenario), then our framework immediately gives an asymptotically faster algorithm than is currently known. This situation is reminiscent to that of the minimum spanning tree, for which it is known that it is essentially equivalent to the *bichromatic closest pair* problem.

The rest of the paper is organized as follows. In Sect. 2 we review a number of well-known definitions, algorithms and results. In Sect. 3 we give the properties of the WSPD and the greedy spanner on which our algorithm is based. In Sect. 4 we present our algorithm and analyze its running time and space usage. In Sect. 5 we discuss our optimizations of the algorithm. Finally, in Sect. 6 we present our experimental results and compare it to other algorithms.

## 2 Notation and Preliminaries

Let  $V$  be a set of points in  $\mathbb{R}^d$ , and let  $t \in \mathbb{R}$  be the intended dilation ( $t > 1$ ). Let  $G = (V, E)$  be a graph on  $V$ . For two points  $u, v \in V$ , we denote the Euclidean

distance between  $u$  and  $v$  by  $|uv|$ , and the distance in  $G$  by  $\delta_G(u, v)$ . If the graph  $G$  is clear from the context we will simply write  $\delta(u, v)$ . The *dilation* of a pair of points is  $t$  if  $\delta(u, v) \leq t \cdot |uv|$ . A graph  $G$  has dilation  $t$  if  $t$  is an upper bound for the dilations of all pairs of points. In this case we say that  $G$  is a  $t$ -spanner. To simplify the analysis, we assume without loss of generality that  $t < 2$ .

We will often say that two points  $u, v \in V$  have a *t-path* if their dilation is  $t$ . A pair of points is *without t-path* if its dilation is not  $t$ . When we say a pair of points  $(u, v)$  is the *closest* or *shortest* pair among some set of points, we mean that  $|uv|$  is minimal among this set. We will talk about a *Dijkstra computation* from a point  $v$  by which we mean a single execution of the single-source shortest path algorithm known as Dijkstra's algorithm from  $v$ .

Consider the following algorithm that was introduced by Keil [11]:

**Algorithm** GreedySpannerOriginal( $V, t$ )

1.  $E \leftarrow \emptyset$
2. **for** every pair of distinct points  $(u, v)$  in ascending order of  $|uv|$
3.     **do if**  $\delta_{(V,E)}(u, v) > t \cdot |uv|$
4.         **then add**  $(u, v)$  to  $E$
5. **return**  $E$

Obviously, the result of this algorithm is a  $t$ -spanner for  $V$ . The resulting graph is called the *greedy t-spanner* for  $V$  (or just the *greedy spanner*, with the parameter  $t$  implied), for which we shall present a more efficient algorithm than the above.

We will make use of the well-separated pair decomposition, or WSPD for short, as introduced by Callahan and Kosaraju in [4,5]. A WSPD is parameterized with a separation constant  $s \in \mathbb{R}$  with  $s > 0$ . This decomposition is a set of pairs of nonempty subsets of  $V$ . Let  $m$  be the number of pairs in a decomposition. We can number the pairs, and denote every pair as  $\{A_i, B_i\}$  with  $1 \leq i \leq m$ . Let  $u$  and  $v$  be distinct points, then we say that  $(u, v)$  is ‘in’ a well-separated pair  $\{A_i, B_i\}$  if  $u \in A_i$  and  $v \in B_i$  or  $v \in A_i$  and  $u \in B_i$ . A decomposition has the property that for every pair of distinct points  $u$  and  $v$ , there is exactly one  $i$  such that  $(u, v)$  is in  $\{A_i, B_i\}$ .

For two nonempty subsets  $X_k$  and  $X_l$  of  $V$ , we define  $\min(X_k, X_l)$  to be the shortest distance between the two circles (or balls in higher dimensions) around the bounding boxes of  $X_k$  and  $X_l$  and  $\max(X_k, X_l)$  to be the longest distance between these two circles. Let  $\text{diam}(X_k)$  be the diameter of the circle around the bounding box of  $X_k$ . Let  $\ell(X_k, X_l)$  be the distance between the centers of these two circles, also named the *length* of this pair. For a given separation constant  $s \in \mathbb{R}$  with  $s > 0$  as parameter for the WSPD, we require that all pairs in a WSPD are  $s$ -well-separated, that is,  $\min(A_i, B_i) \geq s \cdot \max(\text{diam}(A_i), \text{diam}(B_i))$  for all  $i$  with  $1 \leq i \leq m$ .

It is easy to see that  $\max(X_k, X_l) \leq \min(X_k, X_l) + \text{diam}(X_k) + \text{diam}(X_l) \leq (1 + 2/s) \min(X_k, X_l)$ . As  $t < 2$  and as we will pick  $s = \frac{2t}{t-1}$  later on, we have  $s > 4$ , and hence  $\max(X_k, X_l) \leq 3/2 \min(X_k, X_l)$ . Similarly,  $\ell(X_k, X_l) \leq \min(X_k, X_l) + \text{diam}(X_k)/2 + \text{diam}(X_l)/2 \leq (1 + 1/s) \min(X_k, X_l)$  and hence  $\ell(X_k, X_l) \leq 5/4 \min(X_k, X_l)$ .

For any  $V$  and any  $s > 0$ , there exists a WSPD of size  $m = O(s^d n)$  that can be computed in  $O(n \log n + s^d n)$  time and can be represented in  $O(s^d n)$  space [4]. Note

that the above four values ( $\min$ ,  $\max$ ,  $diam$  and  $\ell$ ) can easily be precomputed for all pairs with no additional asymptotic overhead during the WSPD construction.

### 3 Properties of the Greedy Spanner and the WSPD

In this section we will give the idea behind the algorithm and present the properties of the greedy spanner and the WSPD that make it work. We assume we have a set of points  $V$  of size  $n$ , an intended dilation  $t$  with  $1 < t < 2$  and a WSPD with separation factor  $s = \frac{2t}{t-1}$ , for which the pairs are numbered  $\{A_i, B_i\}$  with  $1 \leq i \leq m$ , where  $m = O(s^d n)$  is the number of pairs in the WSPD.

Similar to the original greedy spanner algorithm, other algorithms consider all possible edges in ascending order and focus on reducing the amount of work done in the shortest path computations through caching previous results [7] or undoing and redoing parts of dijkstra computations [2]. We use a completely different approach, instead of working on edges, we change the original greedy algorithm to work on well-separated pairs. We will end up adding the edges in the same order as the greedy spanner. To achieve this, we maintain a set of 'candidate' edges for every well-separated pair such that the shortest of these candidates is the next edge that needs be added. We then recompute a candidate for some of these well-separated pairs to maintain this property. We use two requirements to decide on which pairs we perform a recomputation, that together ensure that we do not do too many recomputations, but also that we do not fail to update pairs which needed updating.

We now give the properties on which our algorithm is based.

**Observation 1** (Bose et al. [2, Observation 1]) *For every  $i$  with  $1 \leq i \leq m$ , the greedy spanner includes at most one edge  $(u, v)$  with  $(u, v)$  in  $\{A_i, B_i\}$ .*

*Proof* In [2] this is proven by assuming that there are two edges in a well-separated pair and then considering the edge which was added last. According to the definition of the greedy spanner this edge could only be added if there was no  $t$ -path between its endpoints. They continue by showing that there is actually a valid  $t$ -path via the first edge and can hence derive a contradiction. It is important to note that our observation is not fully identical to the one from [2] as our definition of well-separatedness is slightly different than theirs. However, their proof uses only properties of their Lemma 2, which still holds true using our definitions as proven in Sect. 2. Their Lemma 2 is almost Lemma 9.1.2 in [12], whose definitions of well-separatedness is near identical to ours, except that they use radii rather than diameters and hence have different constants.  $\square$

An immediate corollary is:

**Observation 2** (Bose et al. [2, Corollary 1]) *The greedy spanner contains at most  $O\left(\frac{1}{(t-1)^d} n\right)$  edges.*

**Lemma 3** *Let  $E$  be some edge set for  $V$ . For every  $i$  with  $1 \leq i \leq m$ , we can compute the closest pair of points  $(u, v) \in A_i \times B_i$  among the pairs of points with dilation larger than  $t$  in  $G = (V, E)$  in  $O(\min(|A_i|, |B_i|)(|V| \log |V| + |E|))$  time and  $O(|V|)$  space.*

*Proof* Assume without loss of generality that  $|A_i| \leq |B_i|$ . We perform a Dijkstra computation for every point  $a \in A_i$ , maintaining the closest point in  $|B_i|$  such that its dilation with respect to  $a$  is larger than  $t$  over all these computations. To check whether a point that is considered by the Dijkstra computation is in  $|B_i|$ , we precompute a boolean array of size  $|V|$ , in which points in  $|B_i|$  are marked as true and the rest as false. This costs  $O(|V|)$  space,  $O(|V|)$  time and achieves a constant lookup time. A Dijkstra computation takes  $O(|V| \log |V| + |E|)$  time and  $O(|V|)$  space, but this space can be reused between computations.  $\square$

**Fact 4** (*Callahan [4, Chapter 4.5]*)  $\sum_{i=1}^m \min(|A_i|, |B_i|) = O(s^d n \log n)$

*Proof* This is not stated explicitly in [4], but it is a direct consequence of the construction of what they call the one-to-many realization, a special variant of the WSPD where all pairs are of the form  $\{\{a\}, B\}$ . It is proven that this one-to-many realization consists of  $O(s^d n \log n)$  pairs, and the construction splits every pair in the canonical realization into  $\min(|A_i|, |B_i|)$  pairs, hence the lemma follows.  $\square$

**Observation 5** Let  $E$  be some edge set for  $V$ . Let  $(a, b) \in E$ . Let  $c \in V$  and  $d \in V$  be points such that  $|ac|, |ad|, |bc|, |bd| > t|cd|$ . Then any  $t$ -path between  $c$  and  $d$  will not use the edge  $(a, b)$ .

*Proof* This directly follows from the fact that  $c$  and  $d$  are so far away from  $a$  and  $b$  that just getting to  $a$  or  $b$  is already longer than allowed for a  $t$ -path.  $\square$

Let  $Box(S)$  denote the bounding box of the points in  $S$ . To prove our next lemma we first quote a Lemma from [12]. They use an elaborate packing argument to show the following (slightly paraphrased to use our notation)

**Fact 6** (*Narasimhan and Smid [12, Lemma 11.3.4]*) Let  $\gamma$  and  $\ell$  be positive real numbers, and let  $D(U, V)$  be a dumbbell whose length is in the interval  $[\ell, 2\ell]$ . The number of dumbbells  $D(a, b)$  such that

1. The length of  $D(A, B)$  is in the interval  $[\ell, 2\ell]$  and
2. At least one of  $Box(A)$  and  $Box(B)$  is within distance  $\gamma\ell$  of either  $Box(U)$  or  $Box(V)$ .

is less than or equal to  $c_s y := O(s^d(1 + \gamma s)^d)$ .

The dumbbells mentioned in Fact 6 are basically the same as our well-separated pairs, but with the separation defined based on the radius instead of the diameter. Through this similarity we can prove the following lemma about our well-separated pairs.

**Lemma 7** Let  $\gamma$  and  $\ell$  be positive real numbers, and let  $\{A_i, B_i\}$  be a well-separated pair in the WSPD with length  $\ell(A_i, B_i) = \ell$ . The number of well-separated pairs  $\{A'_i, B'_i\}$  such that

1. The length of the pair is in the interval  $[\ell/2, 2\ell]$  and
2. At least one of  $\text{Box}(A'_i)$  and  $\text{Box}(B'_i)$  is within distance  $\gamma\ell$  of either  $\text{Box}(A_i)$  or  $\text{Box}(B_i)$

is less than or equal to  $c_{sy} = O(s^d(1 + \gamma s)^d)$ .

*Proof* This follows easily from the very similar Fact 6 which we paraphrased from Lemma 11.3.4 of [12]. We first note the differences between their definitions and ours. Their statement involves *dumbbells*, but these are really the same as our well-separated pairs with a slightly different definition of well-separatedness: their dumbbells use the radius and we use the diameter. We can easily amend this by observing that a WSPD with separation factor  $s$  using our definitions is identical to their dumbbell WSPD with separation factor  $2s$ . This means our constant  $c_{sy}$  is larger than their constant because our  $s$  is doubled, but this is asymptotically irrelevant.

Now we shall see how we can prove our statement using Fact 6. In Fact 6 the interval for values of  $D(A, B)$  is  $[\ell, 2\ell]$ , but our interval for lengths of pairs is  $[\ell/2, 2\ell]$ . Let  $\ell' = \ell(A'_i, B'_i)$ , then we can obtain our lemma by invoking Fact 6 twice, first by setting  $\ell = \ell'$  (resulting in an upper bound on the number such pairs  $n_1$ ) and then by setting  $\ell = \ell'/2$  (resulting in an upper bound on the number of such pairs  $n_2$ ). This counts the number of pairs with lengths in the interval  $[\ell/2, 2\ell]$ . These are exactly the pairs we are interested in, and hence the constant we obtain is  $n_1 + n_2$ , thus proving the lemma.  $\square$

This concludes the theoretical foundations of the algorithm. We will now present the algorithm and analyze its running time.

## 4 Algorithm

We will now describe the algorithm *GreedySpanner* in detail. It first computes the WSPD for  $V$  with  $s = \frac{2t}{t-1}$  and sorts the resulting pairs according to their smallest distance  $\min(A_i, B_i)$ . It then alternates between calling the *FillQueue* procedure that attempts to add well-separated pairs to a priority queue  $Q$ , and removing an element from  $Q$  and adding a corresponding edge to  $E$ . If  $Q$  is empty after a call to *FillQueue*, the algorithm terminates and returns  $E$ .

We assume we have a procedure *ClosestPair*( $i$ ) that for the  $i$ th well-separated pair computes the closest pair of points without  $t$ -path in the graph computed so far, as presented in Lemma 3, and returns this pair, or returns **nil** if no such pair exists. For the priority queue  $Q$ , we let  $\min(Q)$  denote the value of the key of the minimum of  $Q$ . Recall that  $m = O(s^d n)$  denotes the number of well-separated pairs in the WSPD that we compute in the algorithm.

We maintain an index  $i$  into the sorted list of well-separated pairs. It points to the smallest untreated well-separated pair—we treat the pairs in ascending order of  $\min(A_i, B_i)$  in the *FillQueue* procedure. When we treat a pair  $\{A_i, B_i\}$ , we call *ClosestPair*( $i$ ) on it, and if it returns a pair  $(u, v)$ , we add it to  $Q$  with key  $|uv|$ . We link

entries in the queue, its corresponding pair  $\{A_i, B_i\}$  and  $(u, v)$  together so they can quickly be requested. We stop treating pairs and return from the procedure if we have either treated all pairs, or if  $\min(A_i, B_i)$  is larger than the key of the minimal entry in  $Q$  (if it exists).

After extracting a pair of points  $(u, v)$  from  $Q$ , we add it to  $E$ . Then, we update the information in  $Q$ : for every pair  $\{A_j, B_j\}$  having an entry in  $Q$  for which either bounding box is at most  $t|uv|$  away from  $\{A_i, B_i\}$ , we recompute  $\text{ClosestPair}(j)$  and updates its entry in  $Q$  as follows. If the recomputation returns **nil**, we remove its entry from  $Q$ . If it returns a pair  $(u', v')$ , we link the entry of  $j$  in  $Q$  with this new pair and we increase the key of its entry to  $|u'v'|$ .

**Algorithm**  $\text{FillQueue}(Q, i)$

1. **while**  $i \leq m$ , and either  $\min(A_i B_i) \leq \min(Q)$  or  $Q$  is empty
2.   **do**  $p \leftarrow \text{ClosestPair}(i)$
3.     **if**  $p$  is not **nil**, but a pair  $(u, v)$
4.       **then** add  $(u, v)$  to  $Q$  with key  $|uv|$ , and associate this entry with  $\{A_i, B_i\}$
5.      $i \leftarrow i + 1$

**Algorithm**  $\text{GreedySpanner}(V, t)$

1. Compute the WSPD  $W$  for  $V$  with  $s = \frac{2t}{t-1}$ , and let  $\{A_i, B_i\}$  be the resulting pairs,  $1 \leq i \leq m$
2. Sort the pairs  $\{A_i, B_i\}$  according to  $\min(A_i, B_i)$
3.  $E \leftarrow \emptyset$
4.  $Q \leftarrow$  an empty priority queue
5.  $i \leftarrow 1$
6.  $\text{FillQueue}(Q, i)$
7. **while**  $Q$  is not empty
8.   **do** extract the minimum from  $Q$ , let this be  $(u, v)$
9.     add  $(u, v)$  to  $E$
10.   **for** all pairs  $\{A_j, B_j\}$  with an entry in  $Q$  for which either bounding box is at most  $t \cdot \min(A_j, B_j)$  away from either  $u$  or  $v$
11.     **do**  $p \leftarrow \text{ClosestPair}(j)$
12.       **if**  $p$  is **nil**, remove the entry in  $Q$  associated with  $\{A_j, B_j\}$  from  $Q$
13.       **if**  $p$  is a pair  $(u', v')$ , update the entry in  $Q$  associated with  $\{A_j, B_j\}$  to contain  $(u', v')$  and increase its key to  $|u'v'|$
14.      $\text{FillQueue}(Q, i)$
15. **return**  $E$

We now prove correctness and a bound on the running time of the algorithm.

**Theorem 8** *Algorithm GreedySpanner computes the greedy spanner for dilation  $t$ .*

*Proof* We will prove that if the algorithm adds  $(u, v)$  to  $E$  on line 9, then  $(u, v)$  is the closest pair of points without a  $t$ -path between them in the graph computed so far. The greedy spanner consists of exactly these edges and hence this is sufficient to prove the theorem.

It is obvious that if we call  $\text{ClosestPair}(i)$  on every well-separated pair and take the closest pair of the non-**nil** results, then that would be the closest pair of points

without a  $t$ -path between them. Our algorithm essentially does this, except it does not recalculate  $\text{ClosestPair}(i)$  for every pair after every added edge, but only for specific pairs. We will prove that the calls which are not made due to this optimization do not change the values in  $Q$ . Our first optimization is that if a call  $\text{ClosestPair}(i)$  returns **nil** it will always return **nil**, so we need not call  $\text{ClosestPair}(i)$  again, which is therefore a valid optimization.

The restriction ‘for which either bounding box is at most  $t \cdot \min(A_j, B_j)$  away from either  $u$  or  $v$ ’ from the for-loop on line 10 is the second optimization. Its validity is a direct consequence of Observation 5: all pairs of points in well-separated pairs more than  $t \min(A_j, B_j)$  away from either  $u$  or  $v$  are too far away to use the newly-added edge to gain a  $t$ -path. Therefore re-running  $\text{ClosestPair}(i)$  and performing lines 12 and 13 will not change any entries in  $Q$  as claimed.

As  $\min(A_i B_i)$  is a lower bound on the minimal distance between any two points  $(a, b)$  in  $\{A_i, B_i\}$ , it immediately follows that calling  $\text{FillQueue}(Q, i)$  on a pair  $\{A_i, B_i\}$  with  $\min(A_i B_i) > \min(Q)$  cannot possibly yield a pair that can cause  $\min(Q)$  to become smaller. As the pairs are treated in order of  $\min(A_i B_i)$ , this means the optimization that is the condition on line 1 in  $\text{FillQueue}(Q, i)$  is a valid optimization. This proves the theorem.  $\square$

We will now analyze the running time and space usage of the algorithm. We will use the observations in Sect. 3 to bound the amount of work done by the algorithm.

**Lemma 9** *For any well-separated pair  $\{A_i, B_i\}$  ( $1 \leq i \leq m$ ), the number of times  $\text{ClosestPair}(i)$  is called is at most  $1 + c_{st}$ .*

*Proof*  $\text{ClosestPair}(i)$  is called once for every  $i$  in the  $\text{FillQueue}$  procedure.  $\text{ClosestPair}(i)$  may also be called after an edge is added to the graph. We will show that if a well-separated pair  $\{A_j, B_j\}$  causes  $\text{ClosestPair}(i)$  to be called, then  $\ell(A_j, B_j) \in [\ell(A_i, B_i)/2, 2\ell(A_i, B_i)]$ . Then, by the condition of line 10, the collection of pairs that call  $\text{ClosestPair}(i)$  satisfy the requirements of Lemma 7 by setting  $\gamma = t$ , so we can conclude this happens only  $c_{st}$  times. The lemma follows.

We will now show that  $\ell(A_j, B_j) \in [\ell(A_i, B_i)/2, 2\ell(A_i, B_i)]$ . Recall the following from Sect. 2:

$$\begin{aligned} \max(A_i, B_i) &\leq \frac{3}{2} \min(A_i, B_i) \\ \max(A_j, B_j) &\leq \frac{3}{2} \min(A_j, B_j) \\ \min(A_i, B_i) &\leq \ell(A_i, B_i) \leq \frac{5}{4} \min(A_i, B_i) \\ \min(A_j, B_j) &\leq \ell(A_j, B_j) \leq \frac{5}{4} \min(A_j, B_j) \end{aligned}$$

The algorithm ensures the following:

$$\begin{aligned} \min(A_j, B_j) &\leq \min(Q) \leq \max(A_i, B_i) \\ \min(A_i, B_i) &\leq \min(Q) \leq \max(A_j, B_j) \end{aligned}$$

Combining these we have:

$$\begin{aligned}\ell(A_i, B_i) &\leq \frac{5}{4} \min(A_i, B_i) \leq \frac{5}{4} \frac{3}{2} \min(A_j, B_j) < 2\ell(A_j, B_j) \\ \ell(A_i, B_i) &\leq 2\ell(A_j, B_j) \\ \ell(A_j, B_j) &\leq \frac{5}{4} \min(A_j, B_j) \leq \frac{5}{4} \frac{3}{2} \min(A_i, B_i) < 2\ell(A_i, B_i) \\ \ell(A_j, B_j)/2 &\leq \ell(A_i, B_i)\end{aligned}$$

It follows that  $\ell(A_j, B_j) \in [\ell(A_i, B_i)/2, 2\ell(A_i, B_i)]$ .  $\square$

**Theorem 10** *Algorithm GreedySpanner computes the greedy spanner for dilation  $t$  in  $O\left(n^2 \log^2 n \frac{1}{(t-1)^{3d}} + n^2 \log n \frac{1}{(t-1)^{4d}}\right)$  time and  $O\left(\frac{1}{(t-1)^d} n\right)$  space.*

*Proof* We can easily precompute which well-separated pairs are close to a particular well-separated pair as needed in line 10 in  $O(m^2)$  time, without affecting the running time. By Lemma 7 there are only at most  $c_{st}$  such well-separated pairs per well-separated pair, so this uses  $O\left(\frac{1}{(t-1)^d} n\right)$  space.

Combining Observation 2 with Lemma 3 we conclude that we can compute *ClosestPair*( $i$ ) in

$$O\left(\min(|A_i|, |B_i|) \left(n \log n + \frac{1}{(t-1)^d} n\right)\right)$$

time and  $O(n)$  space. By Lemma 9 the time taken by all *ClosestPair*( $i$ ) calls is therefore

$$O\left(\sum_{i=1}^m (1 + c_{st}) \min(|A_i|, |B_i|) \left(n \log n + \frac{1}{(t-1)^d} n\right)\right)$$

and its space usage is  $O(n)$  by reusing the space for the calls. Using Fact 4, this is at most

$$O\left(\frac{1}{(t-1)^d} \left(1 + \frac{t}{t-1}\right)^d \frac{1}{(t-1)^d} n \log n \left(n \log n + \frac{1}{(t-1)^d} n\right)\right)$$

which simplifies to

$$O\left(n^2 \log^2 n \frac{1}{(t-1)^{3d}} + n^2 \log n \frac{1}{(t-1)^{4d}}\right).$$

The time taken by all other steps of the algorithm is insignificant compared to the time used by *ClosestPair*( $i$ ) calls. These other steps are: computing the WSPD and all actions with regard to the queue. All these other actions use  $O\left(\frac{1}{(t-1)^d} n\right)$  space. Combining this with Theorem 8, the theorem follows.  $\square$

## 5 Making the Algorithm Practical

Experiments suggested that implementing the above algorithm as-is does not yield a practical algorithm. With the four optimizations described in the following sections, the algorithm attains running times that are a small constant slower than the algorithm introduced in [7] called FG-greedy, which is considered the fastest practical algorithm known in literature.

### 5.1 Finding Close-By Pairs

The algorithm at some point needs to know which pairs are ‘close’ to the pair for which we are currently adding an edge. In our proof above, we suggested that these pairs be precomputed in  $O(m^2)$  time. Unfortunately, this precomputation step turns out to take much longer than the rest of the algorithm. If  $n = 100$ , then (on a uniform pointset)  $m \approx 2000$  and  $m^2 \approx 4000000$  while the number of edges  $e$  in the greedy spanner is about 135. Our solution is to simply find them using a linear search every time we need to know this information. This only takes  $O(e \cdot m)$  time, which is significantly faster.

### 5.2 Reducing the Number of Dijkstra Computations

After decreasing the time taken by preprocessing, the next part that takes the most time is the Dijkstra computations, whose running time dwarfs the rest of the operations. We would therefore like to optimize this part of the algorithm. For every well-separated pair, we save the length of the shortest path found by any Dijkstra computation performed on it, that is, its source  $s$ , target  $t$  and distance  $\delta(s, t)$ . Then, if we are about to perform a Dijkstra computation on a vertex  $u$ , we first check if the saved path is already good enough to ‘cover’ all nodes in  $B_i$ . Let  $c$  be the center of the circle around the bounding box of  $B_i$  and  $r$  its radius. We check if  $t \cdot |us| + \delta(s, t) + t \cdot (|uc| + r) \leq t \cdot (|uc| - r)$  and mark it as ‘irrelevant for the rest of the algorithm’. This optimization roughly improves its running time by a factor three.

### 5.3 Sharpening the Bound of Observation 5

The bound given in Observation 5 can be improved. Let  $\{A_i, B_i\}$  be the well-separated pair for which we just added an edge and let  $\{A_j, B_j\}$  be the well-separated pair under consideration in our linear search. First, some notation: let  $X_k, X_l$  be sets belonging to some well-separated pair (not necessarily the same pair), then  $\min(X_k, X_l)$  denotes the (shortest) distance between the two circles around the bounding boxes of  $X_k$  and  $X_l$  and  $\max(X_k, X_l)$  the longest distance between these two circles. Let  $\ell = \ell(A_i, B_i)$ . We can then replace the condition of Lemma 5 by the sharper condition  $\min(A_i, A_j) + \ell + \min(B_j, B_i) \leq t \cdot \max(A_j, B_j) \vee \min(A_i, B_j) + \ell + \min(A_j, B_i) \leq t \cdot \max(B_j, A_j)$ . The converse of the condition implies that the edge just added cannot be part of a  $t$ -path

between a node in  $\{A_j, B_j\}$ , so the correctness of the algorithm is maintained. This leads to quite a speed increase.

#### 5.4 Miscellaneous Optimizations

There are two further small optimizations we have added to our implementation.

Firstly, rather than using the implicit linear space representation of the WSPD, we use the explicit representation where every node in the split tree stores the points associated with that node. For pointsets where the ratio of the longest and the shortest distance is bounded by some polynomial in  $n$ , this uses  $O(n \log n)$  space rather than  $O(n)$  space. This is true for all practical cases, which is why we used it in our implementation. For arbitrary pointsets, this representation uses  $O(n^2)$  space. In practice, this extra space usage is hardly noticeable and it speeds up access to the points significantly.

Secondly, rather than performing Dijkstra's algorithm, we use the  $A^*$  algorithm. This algorithm uses geometric estimates to the target to guide the computation to its goal, thus reducing the search space of the algorithm [9].

We have tried a number of additional optimizations, but none of them resulted in a speed increase. We describe them here.

We have tried to replace  $A^*$  by *ALT*, a shortest path algorithm that uses landmarks—see [9] for details on *ALT*—which gives better lower bounds than the geometric estimates used in  $A^*$ . However, this did not speed up the computations at all, while costing some amount of overhead.

We have also tried to further cut down on the number of Dijkstra computations. We again used that we store the lengths of the shortest paths found so far per well-separated pair. Every time after calling *ClosestPair*( $i$ ) we checked if the newly found path is ‘good enough’ for other well-separated pairs, that is, if the path combined with  $t$ -paths from the endpoints of the well-separated pairs would give  $t$ -paths for all pairs of points in the other well-separated pair. This decreased the number of Dijkstra computations performed considerably, but the overhead from doing this for all pairs was greater than its gain.

We tried to speed up the finding of close-by pairs by employing range trees. We also tried using the same range trees to perform the optimization of the previous paragraph only to well-separated pairs ‘close by’ our current well-separated pair. Both optimizations turned out to give a speed increase and in particular the second retained most of its effectiveness even though we only tried it on close-by pairs, but the overhead of range trees was vastly greater than the gain—in particular the space usage of range trees made the algorithm use about as much space as the original greedy algorithms.

### 6 Experimental Results

We have run our algorithm on pointsets whose size ranged from 100 to 1,000,000 vertices on several distributions. If the set contained at most 10,000 points, we have also run the FG-greedy algorithm to compare the two algorithms. We have recorded

both space usage and running time (wall clock time). We have also performed a number of tests with decreasing values of  $t$  on datasets of size 10,000 and 50,000. Finally, as this is the first time we can compute the greedy spanner on large graphs,<sup>2</sup> we have compared it to the  $\Theta$ -graph and the WSPD-based spanner algorithm on large instances. Note that the construction of this WSPD-based spanner is unrelated to our use of the WSPD. Their approach simply chooses the separation constant to be sufficiently high to guarantee the desired dilation when adding a single edge per well-separated pair. Although this approach also results in a linear number of edges their constants are much bigger (see Sect. 6.4).

Similar to [7] we have used three kinds of distributions from which we draw our points: a uniform distribution, a gamma distribution with shape parameter 0.75, and a distribution consisting of  $\sqrt{n}$  uniformly distributed pointsets of  $\sqrt{n}$  uniformly distributed points. The results from the gamma distribution were nearly identical to those of the uniform pointset, so we did not include them. All our pointsets are two-dimensional.

## 6.1 Experiment Environments

The algorithms have been implemented in C++. We have implemented all data structures not already in the `std`. The random generator used was the Mersenne Twister PRNG—we have used a C++ port by J. Bedaux of the C code by the designers of the algorithm, M. Matsumoto and T. Nishimura.

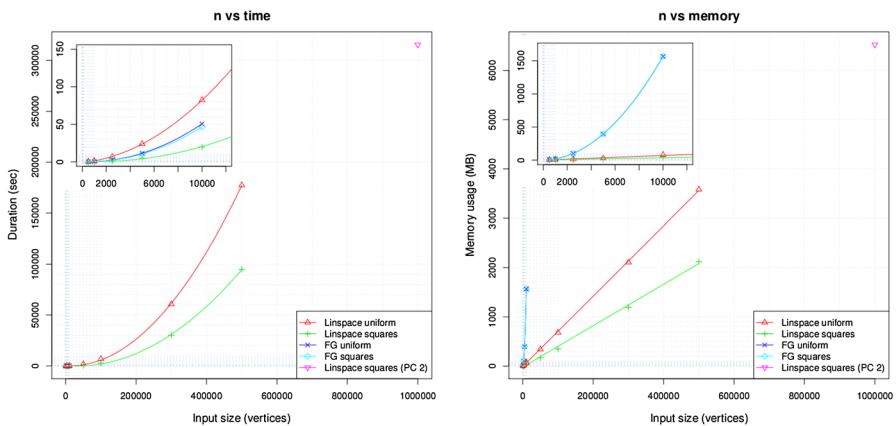
We have used two servers for the experiments. Most experiments have been run on the first server, which uses an Intel Core i5-3470 (3.20GHz) and 4 GB (1600MHz) RAM. It runs the Debian 6.0.7 OS and we compiled for 32 bits using G++ 4.7.2 with the -O3 option. For some tests we needed more memory, so we have used a second server. This server uses an Intel Core i7-3770k (3.50GHz) and 32 GB RAM. It runs Windows 8 Enterprise and we have compiled for 64 bits using the Microsoft C++ compiler (17.00.51106.1) with optimizations turned on.

## 6.2 Dependence on Instance Size

Our first set of tests compared FG-greedy and our algorithm for different values of  $n$ . The results are plotted in Fig. 2. As FG-greedy could only be ran on relatively small instances, its data points are difficult to see in the graph, so we added a zoomed-in plot for the bottom-left part of the plot.

We have used standard fitting methods to our data points: the running time of all algorithms involved fits a quadratic curve well, the memory usage of our algorithm is linear and the memory usage of FG-greedy is quadratic. This nicely fits our theoretical analysis. In fact, the constant factors seem to be much smaller than the bound we gave in our proof. We do note a lack of ‘bumps’ that often occur when instance sizes start exceeding caches: this is probably due to the cache-unfriendly behavior of our

<sup>2</sup> Other linear space algorithms for computing the greedy spanner have been proposed and compared to our algorithm in subsequent work [1, 3].



**Fig. 2** The *left plot* shows the running time of our algorithm on uniform and clustered data for variously sized instances. The *right plot* shows the memory usage of our algorithm on the same data. The lines are fitted quadratic (*right*) and linear (*left*) curves. The outlier at the *right side* was from an experiment performed on a different server. Results for FG-greedy are also shown but in the main plots but are near-impossible to see, so a zoomed-in view of the leftmost corner of both plots is included in the *top-left* of each plot. The memory usage explosion of FG-greedy is visible in the zoomed-out part of the *right plot*

algorithm and the still significant constant factor in our memory usage that fills up caches quite quickly.

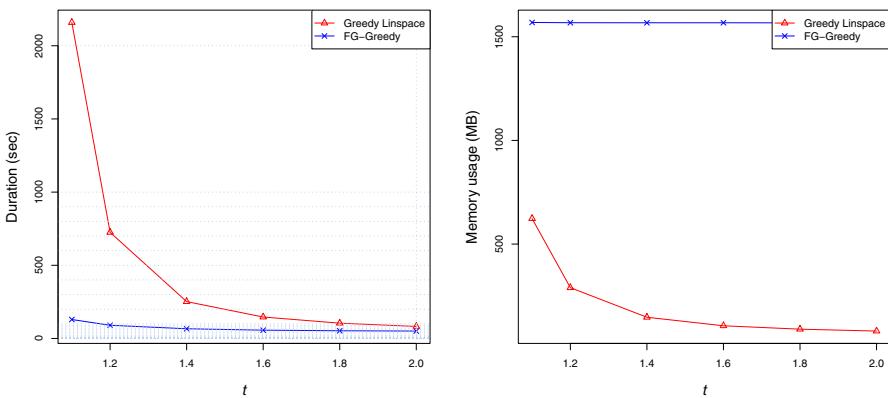
Comparing our algorithm to FG-greedy, it is clear that the memory usage of our algorithm is vastly superior. The plot puts into perspective just how much larger the instances are that we are able to deal with using our new algorithm compared to the old algorithms. Furthermore, our algorithm is about twice as fast as FG-greedy on the clustered datasets, and only about twice as slow on uniform datasets. On clustered datasets the number of computed well-separated pairs is much smaller than on uniform datasets so this difference does not surprise us. These plots suggest that our aim—roughly equal running times at vastly reduced space usage—is reached with this algorithm.

### 6.3 Dependence on $t$

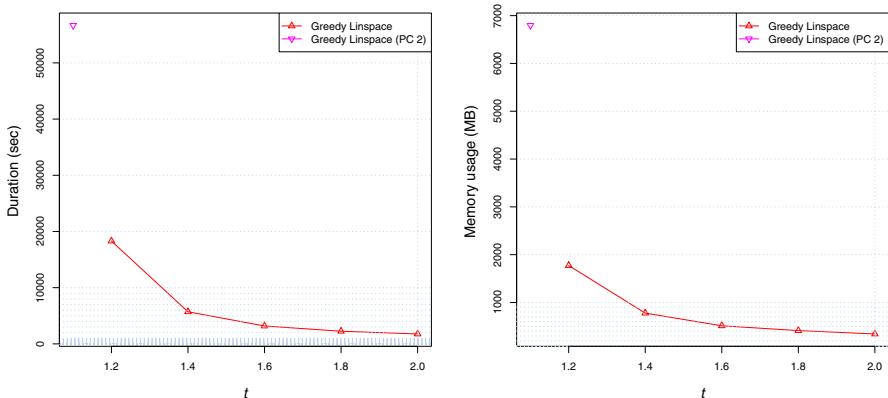
We have tested our algorithms on datasets of 10,000 and 50,000 points, setting  $t$  to 1.1, 1.2, 1.4, 1.6, 1.8 and 2.0 to test the effect of this parameter. The effects of the parameter ended up being rather different between the uniform and clustered datasets.

On uniform pointsets, see Figs. 3 and 4, our algorithm is about as fast as FG-greedy when  $t = 2$ , but its performance degrades quite rapidly as  $t$  decreases compared to FG-greedy. A hint to this behavior is given by the memory usage of our algorithm: it starts vastly better but as  $t$  decreases it becomes only twice as good as FG-greedy. This suggests that the number of well-separated pairs grows rapidly as  $t$  decreases, which explains the running time decrease.

On clustered pointsets, see Figs. 5 and 6, the algorithms compare very differently. FG-greedy starts out twice as slow as our algorithm when  $t = 2$  and when  $t = 1.1$ , our



**Fig. 3** The left plot shows the running time of our algorithm on a dataset of 10,000 uniformly distributed points for various values of  $t$ . The right plot shows the memory usage of our algorithm for the same settings

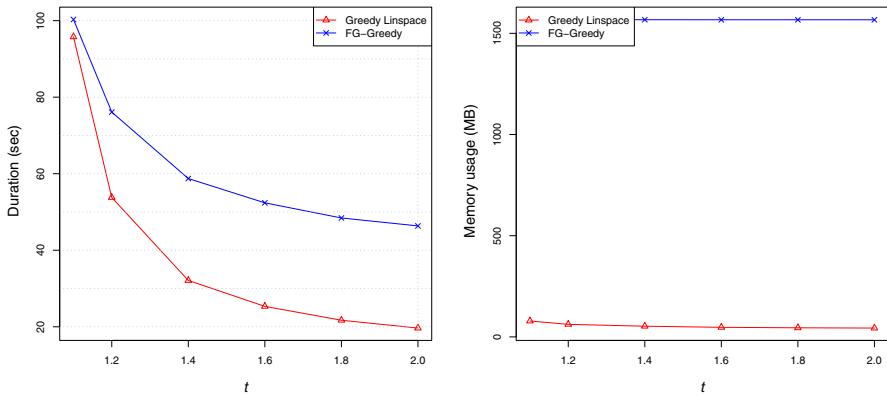


**Fig. 4** The left plot shows the running time of our algorithm on a dataset of 50,000 uniformly distributed points for various values of  $t$ . The right plot shows the memory usage of our algorithm for the same settings. The outlier at the left side was from an experiment performed on a different server. Note that we could not run the FG algorithm on this dataset because it uses too much memory

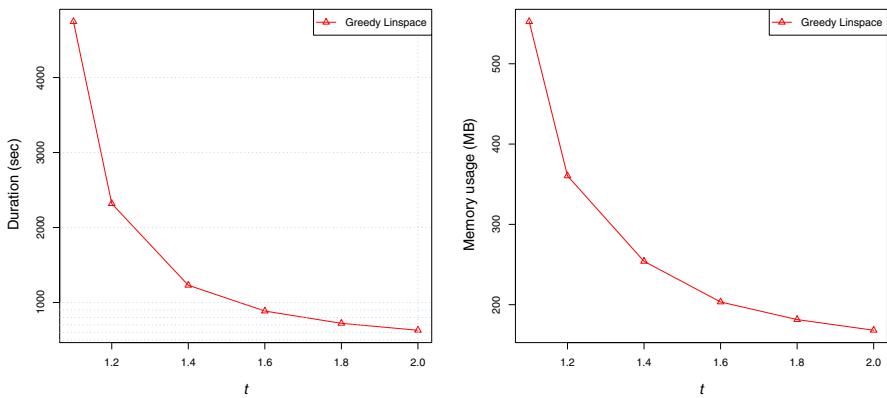
algorithm is only slightly faster than FG-greedy. The memory usage of our algorithm is much less dramatic than in the uniform point case: it hardly grows with  $t$  and therefore stays much smaller than FG-greedy. The memory usage of FG-greedy only depends on the number of points and not on  $t$  or the distribution of the points, so its memory usage is the same.

#### 6.4 Comparison with Other Spanners

We have compared the greedy spanner, the WSPD-spanner and the  $\Theta$ -graph on three big data sets for a dilation of 2. The first set contains geographical data from the USA, it contains 115,475 points and is partly shown in Fig. 1. The other two sets contain synthetic data. The first set consists of 500,000 uniformly distributed points and second set contains 1,000,000 clustered points. Both synthetic sets are generated



**Fig. 5** The *left plot* shows the running time of our algorithm on a dataset of 10,000 clustered points for various values of  $t$ . The *right plot* shows the memory usage of our algorithm for the same settings



**Fig. 6** The *left plot* shows the running time of our algorithm on a dataset of 50,000 clustered points for various values of  $t$ . The *right plot* shows the memory usage of our algorithm for the same settings. Note that we could not run the FG algorithm on this dataset because it uses too much memory

using the method discussed at the start of Sect. 6. We compared the number of edges, the maximum degree and the total weight of the spanners computed by these methods. We were unable to compute the WSPD-spanner on the synthetic sets since our PC ran out of memory. The results are shown in Table 1.

We have also compared the performance of these three spanners when  $t$  is small. Specifically we have computed the greedy spanner on a pointset of 50,000 uniformly distributed points with  $t = 1.1$ . On this instance the greedy spanner has 225,705 edges, a maximum degree of 18 and a weight of 15,862,195. On the same instance, the  $\Theta$ -graph with  $k = 73$  (which, to our knowledge, is the smallest  $k$  for which a guarantee of  $t = 1.1$  has been proven) has 2,396,361 edges, a maximum degree of 146 and a weight of 495,332,746. We were unable to run the WSPD-based spanner algorithm on this pointset with  $t = 1.1$  due to its memory usage.

**Table 1** We computed 2-spanners using different algorithms on big data sets, the WSPD-spanner ran out of memory on the last two sets

Spanner	Dataset	$ V $	$ E $	Max degree	Weight
Greedy	USA	115,475	171,456	5	11,086,417
$\Theta$ -graph	USA	115,475	465,230	62	53,341,205
WSPD	USA	115,475	16,636,489	1,271	20,330,193,426
Greedy	Uniform	500,000	720,850	6	9,104,690
$\Theta$ -graph	Uniform	500,000	2,063,164	22	39,153,380
WSPD	Uniform	500,000	—	—	—
Greedy	Clustered	1,000,000	1,409,946	6	4,236,016
$\Theta$ -graph	Clustered	1,000,000	4,157,016	135	59,643,264
WSPD	Clustered	1,000,000	—	—	—

These results show that the greedy spanner really is an excellent spanner, even on large instances and for low  $t$ , as predicted by its theoretical properties.

## 7 Conclusion

We have presented an algorithm that computes the greedy spanner in Euclidean space in  $O(n^2 \log^2 n)$  time and  $O(n)$  space for any fixed stretch factor and dimension. Our algorithm avoids computing all distances by considering well-separated pairs instead. It consists of a framework that computes the greedy spanner given a subroutine for a bichromatic closest pair problem. We give such a subroutine which leads to the desired result.

We have presented several optimizations to the algorithm. Our experimental results show that these optimizations make our algorithm have a running time close to the fastest known algorithms for the greedy spanner, while massively decreasing space usage. It allowed us to compute the greedy spanner on very large instances of a million points, compared to the earlier instances of at most 13,000 points. Given that our algorithm is the first algorithm with a near-quadratic running time guarantee that has actually been implemented, that it has linear space usage and that its running time is comparable to the best known algorithms, we think our algorithm is the method of choice to compute greedy spanners.

We leave open the problem of providing a faster subroutine for solving the *bichromatic closest pair with dilation larger than  $t$*  problem in our framework, which may allow the greedy spanner to be computed in sub-quadratic time for certain distance measures. Particularly the case of the Euclidean plane seems interesting, as the closely related ‘ordinary’ *bichromatic closest pair* problem can be solved quickly in this setting.

**Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution,

and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

## References

1. Alewijnse, S.P., Bouts, Q.W., ten Brink, A.P.: Distribution-sensitive construction of the greedy spanner. In: 22nd Annual European Symposium on Algorithms (ESA), volume 8737 of Lecture Notes in Computer Science, pp. 61–73. Springer (2014)
2. Bose, P., Carmi, P., Farshi, M., Maheshwari, A., Smid, M.: Computing the greedy spanner in near-quadratic time. *Algorithmica* **58**(3), 711–729 (2010)
3. Bouts, Q.W., ten Brink, A.P., Buchin, K.: A framework for computing the greedy spanner. In: Proceedings of the Thirtieth Annual Symposium on Computational Geometry, SOCG '14, pp. 11–19. ACM (2014)
4. Callahan, P.B.: Dealing with higher dimensions: the well-separated pair decomposition and its applications. PhD thesis, Johns Hopkins University, Baltimore, Maryland (1995)
5. Callahan, P.B., Kosaraju, S.R.: A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM* **42**(1), 67–90 (1995)
6. Chew, L.P.: There are planar graphs almost as good as the complete graph. *J. Compute. Syst. Sci.* **39**(2), 205–219 (1989)
7. Farshi, M., Gudmundsson, J.: Experimental study of geometric t-spanners. *ACM J. Exp. Algorithm.* **14**, 3 (2009)
8. Gao, J., Guibas, L.J., Hershberger, J., Zhang, L., Zhu, A.: Geometric spanners for routing in mobile networks. *IEEE J. Sel. Areas Commun.* **23**(1), 174–185 (2005)
9. Goldberg, A.V., Harrelson, C.: Computing the shortest path: a search meets graph theory. In: 16th ACM-SIAM Symposium on Discrete Algorithms, pp. 156–165. SIAM (2005)
10. Gudmundsson, J., Knauer, C.: Dilation and detours in geometric networks. In: Gonzales, T. (ed.) *Handbook on Approximation Algorithms and Metaheuristics*, pp. 52–1–52–16. Chapman & Hall/CRC, Boca Raton (2006)
11. Keil, J.M.: Approximating the complete euclidean graph. In: 1st Scandinavian Workshop on Algorithm Theory (SWAT), volume 318 of LNCS, pp. 208–213. Springer (1988)
12. Narasimhan, G., Smid, M.: *Geometric Spanner Networks*. Cambridge University Press, New York (2007)
13. Peleg, D., Schäffer, A.A.: Graph spanners. *J. Graph Theory* **13**(1), 99–116 (1989)