Experimental Study of Geometric t-Spanners: A Running Time Comparison

Mohammad Farshi[1,⋆] and Joachim Gudmundsson[2]

[1] Department of Mathematics and Computing Science, TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
m.farshi@tue.nl
[2] NICTA[⋆⋆], Sydney, Australia
joachim.gudmundsson@nicta.com.au

Abstract. The construction of t-spanners of a given point set has received a lot of attention, especially from a theoretical perspective. We experimentally study the performance of the most common construction algorithms for points in the Euclidean plane. In a previous paper [10] we considered the properties of the produced graphs from five common algorithms. We consider several additional algorithms and focus on the running times. This is the first time an extensive comparison has been made between the running times of construction algorithms of t-spanners.

1 Introduction

Consider a set V of n points in the plane. A network on V can be modeled as an undirected graph G with vertex set V of size n and an edge set E of size m where every edge $e = (u, v)$ has a weight $wt(e)$. A geometric (Euclidean) network is a network where the weight of the edge $e = (u, v)$ is the Euclidean distance $|uv|$ between its endpoints u and v. Let $t > 1$ be a real number. We say that a geometric network $G(V, E)$ is a (*geometric*) *t-spanner* for V, if for each pair of points $u, v \in V$, there exists a path in G between u and v of weight at most $t \cdot |uv|$. We call this path a t-path between u and v. The minimum t such that G is a t-spanner for V is called the stretch factor, or dilation, of G. Finally, a subgraph G' of a given graph G is a t-spanner for G if for each pair of points $u, v \in V$, there exists a path in G' of weight at most t times the weight of the shortest path between u and v in G.

Complete graphs represent ideal communication networks, but they are expensive to build; sparse spanners are low-cost alternatives. The weight of the spanner is a measure of its sparseness; other sparseness measures include the number of edges, the maximum degree, and the number of crossings. Spanners for complete Euclidean graphs as well as for arbitrary weighted graphs find applications in robotics, network topology design, distributed systems, design of parallel machines, and many other areas and have been a subject of

⋆ Supported by Ministry of Science, Research and Technology of I. R. Iran.
⋆⋆ National ICT Australia Ltd. is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

considerable research. Recently low-weight spanners found interesting practical applications in areas such as metric space searching [16] and broadcasting in communication networks [14]. Well-known theoretical results also use the construction of t-spanners as a building block, for example, Rao and Smith [17] made a breakthrough by showing an optimal $\mathcal{O}(n \log n)$-time approximation scheme for the Euclidean *traveling salesperson problem*, using t-spanners (or banyans). The problem of constructing spanners has received considerable attention from a theoretical perspective, see the recent book by Narasimhan and Smid [15], but almost no attention from a practical or experimental perspective [10, 16, 18].

In this paper we consider the most well-known algorithms for the construction of t-spanners in the plane: variants of greedy spanners and Θ-graphs, spanners constructed from the well-separated pair decomposition (WSPD), skip-list spanners, sink spanners and some hybrid algorithms. Due to the space limitation we only compare the running times of these algorithms (in the full version the graph properties are also studied) for point sets of size up to 10K points, four different distributions; only two are discussed in this paper, and with values of t between 1.1 and 2. The properties of the standard greedy graph, the (ordered) Θ-graph, the WSPD-graph and the hybrid graphs were discussed in [10].

The paper is organized as follows. Next we briefly go through the desirable properties for t-spanners. In Section 2 we describe the implemented algorithms together with the theoretical bounds and implementation details. In Section 3 we discuss the results and finally discuss possible improvements and future research.

Throughout the paper t will be assumed to be a small constant. In the experiments we used values of t between 1.1 and 2. For larger values of t one can use the Delaunay triangulation which is known to have dilation ≈ 2.42 [13].

1.1 Spanner Properties

As input we are given a set V of n points in the plane and a real value $t > 1$. The aim is to compute a t-spanner for V with some good properties where the quality measurements that one consider are as follows:

Size: The number of edges in the graph. This is the most important measurement and all the implemented algorithms produce spanners with $\mathcal{O}(n)$ edges.

Degree: The maximum number of edges incident to a vertex.

Weight: The weight of a Euclidean network G is the sum of the edge weights. The best that can be achieved is a constant times the weight of the minimum spanning tree, denoted $wt(MST(V))$.

Spanner Diameter: Defined as the smallest integer d such that for any pair of vertices u and v in V, there is a path of length at most $t \cdot |uv|$ between u and v containing at most d edges.

2 Spanner Construction Algorithms

Here we give a short description of each of the implemented algorithms together with their theoretical bounds. Note that some of the properties are competing, e.g., a graph with constant degree cannot have constant spanner diameter, and a graph with small spanner diameter cannot have a linear number of edges [2].

2.1 The Original Greedy Algorithm and an Improvement

The greedy algorithm was discovered independently by Bern in 1989 and Althöfer et al. [1]. The graph constructed using the greedy algorithm will be called a greedy graph. The original algorithm starts with the complete graph G while maintaining a partial spanner graph G' of G. All the edges of G are sorted with respect to their length in increasing order. Next the edges are processed in sorted order. Processing an edge (p, q) entails a shortest path query in G' between p and q. If there is no t-path between p and q in G' then (p, q) is added to G' otherwise it is discarded. The time complexity of the original greedy algorithm is $\mathcal{O}(n^3 \log n)$ and it uses $\mathcal{O}(n^2)$ space.

In [10] we proposed a modifications of the greedy algorithm, denoted improved greedy, that we conjectured should have a running time of $\mathcal{O}(n^2 \log n)$. The idea is that every time a shortest path query is performed from p to q Dijkstra's algorithm computes the shortest distance from p to all other points in V. Instead of neglecting all this information we store it in a matrix. When the next shortest path query, say between u and v, is performed we first look in the matrix if there is a t-path between u and v; if there is then discard (u, v) otherwise perform the query on G' as above. We experimentally compare the running time of the original greedy algorithm with the modified version.

Implementation. The implementations of the algorithms are straight-forward. The shortest path queries are done by using the `Dijkstra` function in LEDA.

2.2 The Approximate Greedy Algorithm

In [10] only the original greedy implementation was considered. It was shown that the quality of the networks produced by the greedy algorithm was superior to the other approaches in terms of number of edges, weight and degree. However, a naïve implementation of it has a running time of $\mathcal{O}(n^3 \log n)$, thus any approach that can speed-up the algorithm would be of great interest. The running time is mainly due to the fact that $\Theta(n^2)$ shortest path queries needed to be answered in a graph with $\mathcal{O}(n)$ edges, each of which could take $\mathcal{O}(n \log n)$ time.

Das and Narasimhan [8] showed how to use clustering to speed up shortest path queries. The approximate greedy algorithm starts with a $\sqrt{t/t'}$-spanner G' with $\mathcal{O}(n)$ edges and constant degree generated by an $\mathcal{O}(n \log n)$-time algorithm. Note that this network does not have to have small weight. Then it computes a $\sqrt{tt'}$-spanner of G' using an approximate variant of the greedy algorithm. To obtain $G(V, E)$ from G' the approximate algorithm starts with $E = \emptyset$ and adds all the short edges (i.e. those of length at most D/n, where D is the distance between the farthest pair of points) to E. For the remaining edges, the algorithm sorts them by increasing weight and processes them in $\log n$ phases. Processing an edge $e = (u, v)$ entails a shortest path query which is answered by performing an approximate shortest path query on a "cluster graph" H, which is simultaneously maintained. The cluster graph H has the following properties:

1. distances in H "closely" approximate distances in the current graph G'.
2. every vertex in H has bounded degree, and
3. "specialized" shortest path queries in H can be answered in constant time.

For more details see [8] or [15]. The time complexity of this algorithm is $\mathcal{O}(n \log^2 n)$. Note that the graph generated by this algorithm is an approximate version of the graph generated by the original greedy algorithm since the algorithm prunes a graph with linear number of edges and answers shortest path queries using an approximate shortest path query procedure.

Gudmundsson et al. [11] later improved the running time to $\mathcal{O}(n \log n)$ but the modified version is quite involved and therefore we decided to only implement the above version. The following theorem states the theoretical bounds.

Theorem 1. *The approximate greedy graph is a t-spanner of V with $\mathcal{O}(n/(t-1)^3)$ edges, $\mathcal{O}(\frac{1}{(t-1)^3})$ maximum degree and weight $\mathcal{O}(wt(MST(V))/(t-1)^4)$, and can be computed in time $\mathcal{O}(\frac{n}{(t-1)^7}\log n)$.*

Implementation. The initial $\sqrt{t/t'}$-spanner G' was constructed using the sink-spanner algorithm (Section 2.7). This guarantees that the number of edges is $\mathcal{O}(n)$ and that the graph has constant degree. We implemented a variant of Dijkstra's algorithm which answers shortest path queries in constant time in the cluster graph. The query time can be achieved since the maximum degree of the cluster graph is constant and there is a constant upper bound B on the number of edges along a shortest path in the cluster graph, thus we may discard any path containing more than B edges in the priority queue. The bound B can be obtained by choosing the size of the clusters in the cluster graph appropriately.

2.3 The Θ-Graph

The Θ-graph was discovered independently by Clarkson [7] and Keil [12]. Keil only considered the graph in two dimensions while Clarkson extended his construction to also include three dimensions.

Initially we set θ such that $t = \frac{1}{\cos\theta - \sin\theta}$. For each point $u \in V$ consider k non-overlapping cones, $C_i, 1 \leq i \leq k$, with angle $\theta = \frac{2\pi}{k}$ and with apex u. For each cone C_i we add an edge between u and the point within C_i whose orthogonal projection onto the bisector of C_i is closest to u. Note that instead of the bisector of C_i, we can use any line in the cone passing through the apex of the cone. We use one of the boundary lines of the cone instead of the bisector.

Theorem 2. *The Θ-graph is a t-spanner of V for $t = \frac{1}{\cos\theta - \sin\theta}$ with $\mathcal{O}(kn)$ edges and can be computed in $\mathcal{O}(kn \log n)$ time.*

Implementation. To implement the Θ-graph algorithm, we need a dynamic data structure, see [15] for more details, that can perform a point query in a cone in $\mathcal{O}(\log n)$ time. This data structure is implemented using red-black trees. Since there is no dependence between the cones, one can work on one cone direction at a time, which means that in practice only $\mathcal{O}(n)$ work space is needed.

A problem that we do not consider in the Θ-graph implementation is rounding errors, which may cause some edges not to be added. For example, if a point lies on the boundary of an, otherwise empty, cone then a small rounding error may "move" the point outside the cone. One way to get rid of this error is to use exact arithmetics. A different possibility is to allow the cones to slightly overlap.

2.4 The Ordered Θ-Graph

A simple variant of the Θ-graph that has been shown to have good theoretical performance is the *ordered Θ-graph* by Bose et al. [5]. An ordered Θ-graph of V is obtained by inserting the points of V in some order. When a point p is inserted, we draw the cones around p and connect p to the previously inserted point with closest orthogonal projection in each cone, like the Θ-graph algorithm.

The order is decided as follows. Initially choose an arbitrary vertex $v_n \in V$ and set its order to n, i.e. this is the last point that will be added to the graph. Process v_n by placing k cones with apex at v_n and then adding the edges as in the Θ-graph algorithm. In a generic step, assume we have processed $i-1$ vertices. In the ith step, choose a point with maximum degree from $V - \{v_n, \ldots, v_{n-(i-1)}\}$ and set its order to $n-i$ and then process v_{n-i} assuming that we have the point set $V - \{v_n, \ldots, v_{n-i+1}\}$. This decides an order on the point set.

Theorem 3. *The ordered Θ-graphs is a t-spanner of V for $t = \frac{1}{\cos\theta - \sin\theta}$ with $\mathcal{O}(kn)$ edges and $\mathcal{O}(k \log n)$ degree, and can be computed in $\mathcal{O}(kn \log n)$ time.*

Implementation. For the implementation we use a data structure which is somewhat more complicated than the data structure used for the Θ-graph, since we require the structure to allow for deletions. Due to [5], we use k range trees, one for each cone with apex at the origin. In each range tree we store all points represented in the coordinate system of the two boundaries of the cone. To find the suitable point in a cone with apex at u, it is sufficient to perform a range query with coordinates of u as keys and choose the suitable point between the points reported by the query. We add one extra pointer to each node of the range tree which shows the point with minimum y (or x) coordinate in the subtree. Using this pointer, we can find the suitable point without going through all reported points of the range query. Each range query requires $\mathcal{O}(\log^2 n)$ time, so the total time complexity of the implemented algorithm is $\mathcal{O}(n \log^2 n)$ which is slightly more than the theoretical time bound but much simpler to implement.

In each step of the ordered Θ-graph algorithm the node with maximum degree has to be selected. To find this point, we used a priority queue of all the points. Initially all the nodes have priority n. When an edge (p, q) is added to the partial spanner graph, the priority of p and q is decreased by 1. The point with minimum priority in the queue is the point with maximum degree in the graph.

There is a major difference between the Θ-graph algorithm and the ordered Θ-graph algorithm when it comes to the space complexity. One can construct the Θ-graph by working on one cone direction at a time, while the ordered Θ-graph algorithm requires us to keep all the cones (range trees) in memory. This is due to the fact that the order is not known in advance. During the processing of one node, we need to check all the cones and add edges if necessary, thus $\Theta(kn)$ space is needed. For small values of t this might cause a major problem. To be more precise, the Θ-graph algorithm used roughly 2% of the memory when constructing a 1.05-spanner on a set with 10,000 points, while the ordered Θ-graph algorithm used almost 85%.

2.5 The Random Ordered Θ-Graph

The ordered Θ-graph algorithm inserts points into the graph in a specific order. However, if the points are processed in random order then the spanner diameter will be bounded by $\mathcal{O}(\log n)$ with high probability [5]. Unfortunately, the degree bound does not hold in this case. There are two reasons why we implemented the random Θ-graph. (1) Random ordered Θ-graphs and skip-list spanners (Section 2.8) are the only two spanners guaranteed to have bounded spanner diameter. Thus a comparison in practice between the two graphs is interesting. (2) Since the vertices are processed in random order we may fix a random order at the beginning which implies that the algorithm only requires $\mathcal{O}(n)$ space, compared to $\mathcal{O}(kn)$ space needed to construct ordered Θ-graphs.

Implementation. The implementation is the same as for the ordered Θ-graph. We only make a random permutation on the input point set and then process the points in the order they appear in after permutation.

2.6 The WSPD-Graph

The well-separated pair decomposition (WSPD) was developed by Callahan and Kosaraju [6].

Definition 1. *Let $s > 0$ be a real number and let A and B be two finite sets of points in $\mathbb{R}^d$. We say that A and B are* well-separated *with respect to s, if there are two disjoint d-dimensional balls C_A and C_B, having the same radius, such that (i) C_A contains A, (ii) C_B contains B, and (iii) the distance between C_A and C_B is at least s times the radius of C_A.*

Definition 2. *Let V be a set of n points in $\mathbb{R}^d$, and let $s > 0$ be a real number. A WSPD for V with respect to s is a sequence of pairs of non-empty subsets of V, $\{A_i, B_i\}_{i=1}^m$, such that (i) A_i and B_i are well-separated w.r.t. s, for all $i = 1, \ldots, m$. (ii) for any two distinct points p and q of V, there is exactly one pair $\{A_i, B_i\}$ in the sequence, such that $p \in A_i$ and $q \in B_i$, or $q \in A_i$ and $p \in B_i$. The integer m is called the* size *of the WSPD.*

Callahan and Kosaraju showed that a set of well-separated pairs of size $m = \mathcal{O}(s^d n)$ can be computed in $\mathcal{O}(s^d n + n \log n)$ time. Constructing a t-spanner using the WSPD is surprisingly easy. It is sufficient to compute a WSPD of V w.r.t. $s = \frac{4(t+1)}{t-1}$ and then for every well-separated pair (A, B) in the WSPD an edge is added between an arbitrary point in A and an arbitrary point in B.

Theorem 4. *The WSPD-graph is a t-spanner for $V \subset \mathbb{R}^2$ with $\mathcal{O}((\frac{t}{t-1})^2 n)$ edges, $\mathcal{O}(\log n \cdot wt(MST(V)))$ weight and can be constructed in time $\mathcal{O}((\frac{t}{t-1})^2 n + n \log n)$.*

We also implemented two versions of the algorithm to improve the degree [2] and the spanner diameter [3]. However, since the experiments showed no improvements for any of these properties we decided not to include them in this paper.

Implementation. We used a split tree for the construction of the WSPD. The points stored at a node is partitioned into two sets by partitioning the non-empty bounding box along its longest side into two boxes of equal size. The tree construction only requires a few percent of the total running time in all our tests.

To decide in constant time if two sets are well-separated we save the smallest enclosing circle of the points in each node. However, their smallest enclosing circles may have different radius so one way to handle this is to say that two sets are well-separated w.r.t. s if the distance between the smallest enclosing circles is at least s times the maximum radius of the two smallest enclosing circles.

2.7 The Sink-Spanner

The sink-spanner construction was defined by Arya et al. in [2] which construct t-spanners with constant degree. The main idea is as follows. We start with a directed $\sqrt{t}$-spanner with bounded out-degree, denoted $\overrightarrow{G}$. We will use the Θ-graph which easily can be seen to have out-degree k, but linear in-degree. For each vertex q in $\overrightarrow{G}$, replace every "star" (the subgraph consisting of all edges in $\overrightarrow{G}$ pointing to q) in $\overrightarrow{G}$ by a $\sqrt{t}$-q-sink spanner. A $\sqrt{t}$-q-sink spanner is a directed graph where each point has a directed $\sqrt{t}$-path to q. It can be obtained by processing each node q in $\overrightarrow{G}$ as follows. Consider all points which have an edge pointing to q. Let A_q be the set of all such a nodes. We replace all the edges pointing to q by a $\sqrt{t}$-path using the partial sink spanner procedure. In the partial sink spanner procedure we look at k cones with apex at q and we partition the points in A_q based on the cones. Let S_i be the points in the ith cone. For each cone i, add an edge between q and the closest point in S_i, say q_i, and then recurse on the partial sink spanner procedure on q_i and $S_i \setminus \{q_i\}$. In the case that one cone contains more than half of the points, split the points in the cone to two almost equal parts and do the same thing as above. This guarantees that the subproblems half in size, thus we get:

Theorem 5. *The sink-spanner is a t-spanner for $V \subset \mathbb{R}^2$ with $\mathcal{O}(kn)$ edges and $\mathcal{O}(\frac{1}{(t-1)^2})$ maximum degree, and can be constructed in time $\mathcal{O}(kn \log n)$.*

Implementation. To construct the first directed $\sqrt{t}$-spanner, we use the Θ-graph algorithm, with the modification that we add directed edges instead of undirected edges. Then for each node q in the directed graph, we look at all points which has an edge pointing to q. Let A_q be the set of all such a nodes. We replace all the edges pointing to q by a $\sqrt{t}$-path using the partial sink spanner procedure. In the partial sink spanner procedure we look at k cones with apex at q and we partition the points in A_q based on the cones. Let S_i be the points in the ith cone. For each cone i, add an edge between q and the closest point in S_i, say q_i, and then recurse on the partial sink spanner procedure on q_i and $S_i \setminus \{q_i\}$. In the case that one cone contains more than half of the points, split the points in the cone to two almost equal parts and do the same thing as above. This guarantees that the subproblems half in size.

2.8 Skip-List Spanner

To obtain a spanner with bounded spanner diameter, one can use skip-list spanners as suggested by Arya et al. [4]. The idea is to generalize skip-lists and apply them to the construction of t-spanners.

To construct a t-spanner of V, we construct a sequence of subsets of V, $V = V_0 \supseteq V_1 \supseteq \cdots \supseteq V_k = \emptyset$. To construct V_{i+1}, we flip a fair coin for each element of V_i and then add the point to V_{i+1} if the flip produce heads. The construction ends when the set is empty. Now we construct a t-spanner using the Θ-graph algorithm for each V_i and the union of all these graphs is the skip-list spanner of V.

Theorem 6. *The skip-list spanner is a t-spanner for $V \subset \mathbb{R}^2$ with $\mathcal{O}(kn)$ edges, $\mathcal{O}(\log n)$ spanner diameter and can be constructed in time $\mathcal{O}(kn \log n)$. All the bounds are expected with high probability.*

Implementation. To construct a skip-list spanner, we construct a t-spanner on V using the Θ-graph algorithm. Then for each point in the set we produce a random number between 0 and 10,000 using `random_source` type in LEDA and remove the point if the outcome is less than 5,000. Then again we construct the Θ-graph on the remaining points and we add the generated edges to the previous graph. We continue this procedure until we have no remaining points in the set.

2.9 The Hybrid Algorithms

In [10] it was experimentally shown that the greedy algorithm produced graphs whose size, weight and degree are superior to the graphs produced from the other approaches. However the running time of the greedy algorithm is $\mathcal{O}(n^3 \log n)$. A way to improve the running time while, hopefully, still obtaining the high-quality graphs is to first compute a t^{α}-spanner $(0 < \alpha < 1)$ $G(V, E)$ of the input set and then compute a $(t^{1-\alpha})$-spanner of $G(V, E)$ using the greedy pruning algorithm (in the experiments we use the improved greedy algorithm). The resulting graph will have dilation at most $t^{1-\alpha} \cdot t^{\alpha} = t$. The greedy pruning algorithm is identical to the greedy algorithm, but instead of considering the edges in the complete graph the algorithm only considers the edges in E. The time complexity of the implemented greedy pruning is $\mathcal{O}(mn \log n)$, where m is the number of edges in the input graph.

Table 1. Summarizing the known bounds for the algorithms presented in the paper. The entries marked (*) implies that the values are expected with high probability. The entries marked with (†) indicates that the versions implemented in this paper has an additional $\log n$-factor in their running times.

-	Edges	Weight	Degree	Diameter	Time
Greedy-graph	$\mathcal{O}(\frac{n}{t-1})$	$\mathcal{O}(\frac{1}{(t-1)^4} \cdot wt(MST))$	$\mathcal{O}(\frac{1}{t-1})$	$\Theta(n)$	$\mathcal{O}(n^3 \log n)$
Apx. greedy-graph	$\mathcal{O}(\frac{n}{(t-1)^3})$	$\mathcal{O}(\frac{1}{(t-1)^4} \cdot wt(MST))$	$\mathcal{O}(\frac{1}{(t-1)^3})$	$\Theta(n)$	$\mathcal{O}(\frac{n}{(t-1)^7} \log n)^\dagger$
Θ-graph	$\mathcal{O}(n/\theta)$	$\Theta(n \cdot wt(MST))$	$\Theta(n)$	$\Theta(n)$	$\mathcal{O}(n/\theta \log n)$
O. Θ-graph	$\Theta(n/\theta)$	$\mathcal{O}(n \cdot wt(MST))$	$\mathcal{O}(1/\theta \cdot \log n)$	$\Theta(n)$	$\mathcal{O}(n/\theta \log n)^\dagger$
WSPD-graph	$\Theta(\frac{n}{(t-1)^2})$	$\mathcal{O}(\log n \cdot wt(MST))$	$\Theta(n)$	$\Theta(n)$	$\mathcal{O}((\frac{t}{t-1})^2 n + n \log n)$
Sink-spanner	$\Theta(n/\theta)$	$\mathcal{O}(n \cdot wt(MST))$	$\mathcal{O}(\frac{1}{(t-1)^2})$	$\Theta(n)$	$\mathcal{O}(n/\theta \log n)$
Skip-list spanner	$\Theta(n/\theta)^*$	$\Theta(n \cdot wt(MST))^*$	$\Theta(n)$	$\mathcal{O}(\log n)^*$	$\mathcal{O}(n/\theta \log n)^*$

3 Experimental Results

In this section we discuss the experimental results in more detail by considering the running times of the algorithms. The experiments were done on point sets ranging from 100 to 10,000 points with four different distributions:

– uniform distribution,
– normal distribution with mean 500 and deviation 100,
– gamma distribution with shape parameter 0.75, and
– $\sqrt{n}$ uniformly distributed unit squares with $\sqrt{n}$ uniformly distributed points.

Below we will focus on the uniform distribution and the cluster distribution. A discussion considering all the distributions will be available in the full version of the paper together with a comparison of all the graphs produced by the algorithms. To avoid the effect of specific instances, we ran the algorithms on many different instances and took the average of the results.

3.1 Implementation Details

The algorithms were implemented in C++ using the LEDA 5.01 library. In the cases when LEDA did not contain the required data structure needed for the algorithms, we implemented it ourselves.

The experiments were performed on an AMD Opteron 250 (2.4 GHz), 1GB L2 cache and 4GB RAM. The OS was Fedora 3.4 and it used g++ 3.4.4 for compiling the program using `-O2` option. All sample points sets were generated by NEWRAN03 [9] pseudo random number generator.

3.2 Uniform Distribution

The running times of all the implemented algorithms for $t = 2$ and $t = 1.1$ are depicted in Fig. 1. As the theoretical bounds suggest the original greedy algorithm has the highest time complexity of the implemented algorithms and it shows clearly in the experiments. However, the suggested improvement, the improved greedy algorithm, performed very well in the experimental study and the results corroborate our conjecture that only a linear number of shortest path queries are needed. Figure 1c shows the number of shortest path queries performed by the algorithm for $t = 2$. As an example of the improved running time we constructed a greedy 2-spanner on a set of 4K uniformly distributed points; the original algorithm required 12K seconds while the improved algorithm needed roughly 34 seconds. The improved greedy algorithm performed approximately 13K shortest path queries while the original algorithm performs roughly 8 million queries. Using the improved algorithm we are able to construct greedy graphs for much larger points sets than earlier. For instance for a set of 10K points we can construct 2-spanner greedy graph in about 300 seconds. Figures 3 and 4 illustrates the quality of the obtained graphs using different quality measures.

Based on the experiments, the running time of the improved greedy algorithm is comparable to the running times of the hybrid algorithms using $\alpha = 0.5$ for $t = 2$ and it performs even better for smaller values of t, see Fig. 1 for a comparison. Thus, if high quality networks is a priority the improved greedy algorithm is probably the best choice, especially for small values of t, see Fig. 3 and Fig. 4. Note that the improved greedy algorithm generates the same graph as the original greedy algorithm.

For the hybrid approach, three different values of α were used, $0.1, 0.5$ and 0.9, and the results can be seen in Fig. 2a. By increasing α, less time is used to build the initial graph while more time is needed for the pruning process, see Fig. 2b. However, in [10] it was clearly shown that the decrease in speed gave a better quality network, i.e. small size, low degree and low weight.

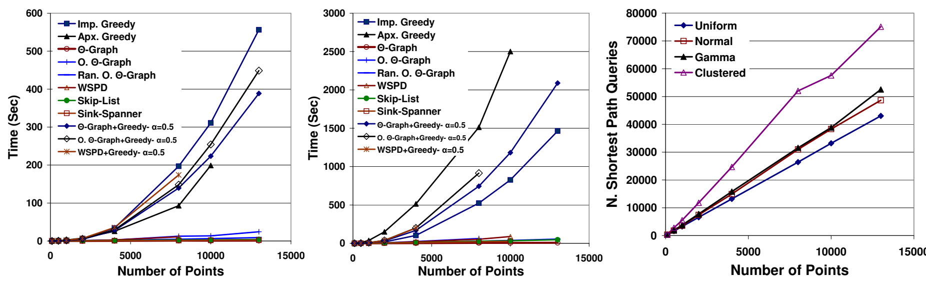

Fig. 1. Comparing the running times of the implemented algorithms for Uniform dist., (a) $t = 2$ and (b) $t = 1.1$. Note the difference between the approximate greedy algorithm and the improved greedy algorithm for the two values of t. (c) Number of shortest paths queries performed by the improved greedy algorithm with $t = 2$.

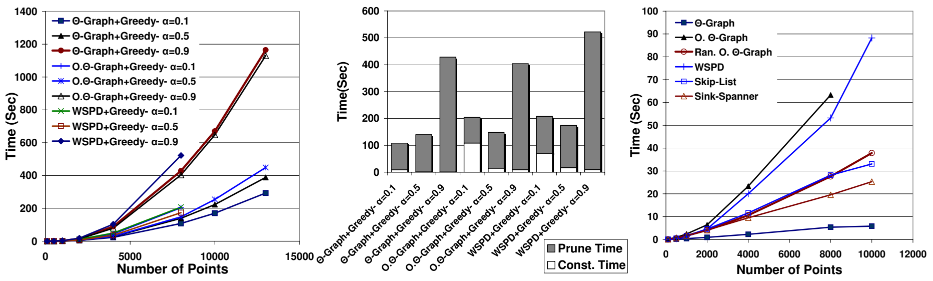

Fig. 2. (a) The performance of the hybrid algorithms for different values of α (Uniform distr. and $t = 2$). (b) Comparing the construction time of the initial graph with the pruning time (Uniform distr., $t = 2$ and $n = 8000$). (c) The running time for the $\mathcal{O}(n \log n)$-time algorithms in the experiments for Uniform distr. with $t = 1.1$.

The remaining algorithms all have a theoretical $\mathcal{O}(n \log^2 n)$, or even $\mathcal{O}(n \log n)$, time complexity. However, the difference in their actual running times is quite substantial and for some a bit surprising. The Θ-graph algorithm is superior to the others with respect to the running time. For sets containing 10K points and for t between 1.5 and 2 the Θ-graph was constructed in less than two seconds. For $t = 1.1$ the running time increased to approximately 6.5 seconds, which is to be expected since its running time is highly dependent on the value of $1/(t-1)^2$. The fastest algorithms after the Θ-graph construction were the sink-spanner algorithm, the skip-list spanner algorithm and the random Ordered Θ-graph algorithm which basically all are modified Θ-graph algorithms. Again for 10K points they required a couple of seconds for $t = 2$ and approximately half a minute for $t = 1.1$. These three algorithms almost show a linear time behavior in our experiments, see Fig. 2c.

For uniform sets the ordered Θ-graph algorithm and the WSPD algorithm clearly show a superlinear behavior but they are still fast enough to handle 8K points with $t = 1.1$ in roughly one minute. For smaller values of t and larger point sets the ordered Θ-graph algorithm ran into memory problems. The simplified

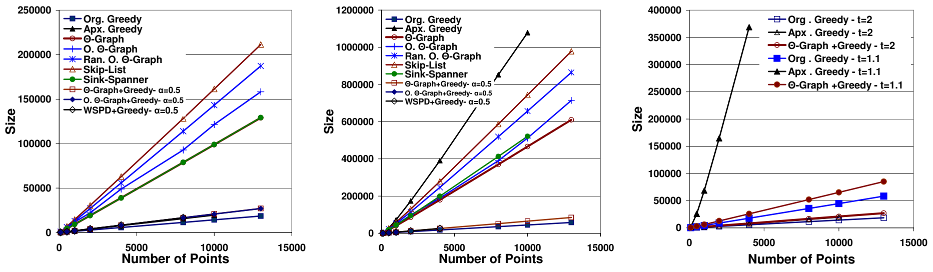

Fig. 3. Comparing the size of the produced graphs for uniform point sets and (a) $t = 2$, and (b) $t = 1.1$. (c) Comparing the size of the high-quality spanners using different values of t.

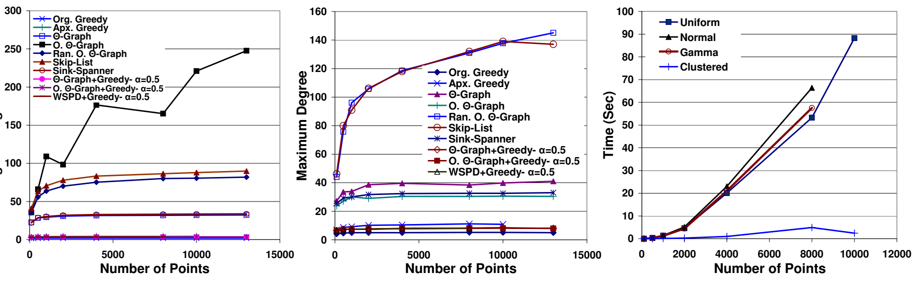

Fig. 4. (a) The weight of the produced graphs for uniform point sets and $t = 2$. (b) The average maximum degree of the produced graphs for uniform distributions and $t = 2$. (c) The running time for the WSPD algorithm for $t = 1.1$ for different distributions.

version that we implemented uses $\Omega(\frac{2\pi}{\theta} n \log n)$ space (instead of $\Omega(\frac{2\pi}{\theta} n)$ space) and for small values of t and large values of n this function grows rapidly.

The approximate greedy algorithm works well for large values of t. For $t = 2$ the running time is comparable to the fastest hybrid algorithms but the produced graphs can be shown to have slightly better quality. When t decreases the running time of the algorithm deteriorates rapidly and for $t = 1.1$ the algorithm performs even worse than the improved algorithm which is conjectured to have a running time of $\mathcal{O}(n^2 \log n)$ (see Fig. 1). The reason is that the approximate greedy approximates the greedy algorithm in two steps; first the complete graph is approximated using a dense t'-spanner G' and then the shortest path queries in G' are approximated using a cluster graph H. This works well for large values of t, and in theory for any constant, however, in practice t becomes too small at some point and the error when doing the approximation becomes too large. As a result the initial graph G' will be very dense (but still linear in n) and the approximation factor used for the approximate shortest path query will be so small that it is equivalent to the exact shortest path query in many cases.

Finally, the produced graphs most often contain many "redundant" edges, i.e., edges that could be removed while still keeping the dilation bounded by t. Table 2 clearly shows this, e.g., the skip-list spanner, sink-spanner and Θ-graph all produce spanners of dilation approximately 1.2 in the case when $t = 2$.

3.3 Clustered Distributions

Most of the algorithms perform slightly better on the clustered point sets, except the WSPD-algorithm and the approximate greedy algorithm which both show a considerable improvement. For example, to construct a 2-spanner on a uniformly distributed set which contains 8K points, the WSPD algorithm needs roughly 11 seconds while the corresponding running time for the clustered set is about 1.6 seconds. For $t = 1.1$ the improvement is even bigger; 88 seconds compared to 2.5 seconds, see Fig. 4c. The WSPD algorithm was expected to perform slightly better for clustered sets since it uses a clustering approach, but the improvement was greater than predicted. Especially for small values of t the

Table 2. The maximum dilation of graphs generated by different algorithms

Maximum Dilation (Uniform distribution)										
	$t = 2$					$t = 1.1$				
n	500	1000	2000	4000	8000	500	1000	2000	4000	8000
Original greedy	1.99	2	2	2	2	1.1	1.1	1.1	1.1	1.1
Improved greedy	1.99	2	2	2	2	1.1	1.1	1.1	1.1	1.1
Approximate greedy	1.68	1.68	1.68	1.68	1.68	1.07	1.07	1.07	1.07	
Θ-graph	1.18	1.2	1.21	1.23	1.22	1.02	1.02	1.03	1.03	1.03
O. Θ-graph	1.37	1.4	1.43	1.46	1.47	1.06	1.07	1.07	1.07	1.07
Random O. Θ-graph	1.35	1.38	1.42	1.41	1.47	1.06	1.06	1.07	1.07	1.07
WSPD-graph	1.35	1.39	1.44	1.49	1.47	1.04	1.04	1.05	1.05	1.05
Skip-list	1.17	1.18	1.21	1.21	1.21	1.02	1.02	1.03	1.03	1.03
Sink-spanner	1.19	1.19	1.21	1.23	1.23	1.03	1.03	1.03	1.03	1.04

algorithm performs better, it is even comparable to the Θ-graph algorithm for the clustered set with 10K points and $t = 1.1$. A similar observation can be made for the approximate greedy where the corresponding running times for $t = 1.1$ and $n = 8K$ are 1500 seconds and 128 seconds. As for the WSPD-approach the approximate greedy algorithm also uses a clustering approach however the main gain comes from the fact that the algorithm does not process any edges in the initial graph G' of length at most D/n (they are just added to the partial spanner graph), where D is the diameter of the point set. In the clustered case there will be many such edges and thus only "long" edges has to be processed.

An interesting observation that can be seen in Fig. 1c is that the number of shortest path queries performed by the improved greedy algorithm in uniformly distributed sets is considerably smaller than for the clustered points set, while the running time is almost the same. Consider the case when the input contains 10K points. The number of shortest path queries performed on the uniform set is approximately 33K while it is about 57K for the clustered set. The number of clusters is 100, with 100 points per cluster. From the experiments it follows that the number of shortest-path queries performed between two points within the same cluster of uniformly distributed points is approximately 300. Since there are 100 clusters the number of shortest path queries needed for the "intra-cluster" edges in the clustered set is approximately 30K. These queries are all performed on very small graphs and are therefore processed extremely fast. Next approximately 27K "inter-cluster" queries are performed. We believe that the smaller number of "inter-cluster" queries together with the fact that the 2-spanner of the clustered set is slightly smaller than for the uniform set explains why the running times for the two different distributions are almost identical.

4 Concluding Remarks and Acknowledgements

We studied the running time of the most common construction algorithm for t-spanners. In addition to the algorithms presented in [10] we also tested sink-spanners, skip-list spanners and, most importantly, the approximate greedy spanner. Unfortunately, the approximate greedy algorithm performs worse than expected in most cases, even though the theoretical bounds are very good.

In general the Θ-graph construction algorithm is the fastest algorithm, however if it is important to obtain a high quality network then the improved greedy algorithms seems to be the most suitable choice. The main question that remains to be answered experimentally is the dependency on the number of dimensions, i.e. how the algorithms and the quality of the produced graphs depends on the number of dimensions.

The authors would like to thank the anonymous reviewers for comments on a earlier version of this paper.

References

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

[2] Arya, S., Das, G., Mount, D.M., Salowe, J.S., Smid, M.: Euclidean spanners: short, thin, and lanky. In: Proc. 27th ACM Symposium on Theory of Computing, pp. 489–498. ACM Press, New York (1995)

[3] Arya, S., Mount, D.M., Smid, M.: Randomized and deterministic algorithms for geometric spanners of small diameter. In: Proc. 35th IEEE Symposium on Foundations of Computer Science, pp. 703–712 (1994)

[4] Arya, S., Mount, D.M., Smid, M.: Dynamic algorithms for geometric spanners of small diameter: Randomized solutions. Computational Geometry: Theory and Applications 13(2), 91–107 (1999)

[5] Bose, P., Gudmundsson, J., Morin, P.: Ordered theta graphs. Computational Geometry: Theory and Applications 28, 11–18 (2004)

[6] Callahan, P.B., Kosaraju, S.R.: 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)

[7] Clarkson, K.L.: Approximation algorithms for shortest path motion planning. In: Proceedings of the 19th ACM Symposium on the Theory of Computing, pp. 56–65 (1987)

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

[9] Davis, R.B.: `http://www.robertnz.net/nr03doc.htm` (2005)

[10] Farshi, M., Gudmundsson, J.: Experimental study of geometric t-spanners. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 556–567. Springer, Heidelberg (2005)

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

[12] Keil, J.M.: Approximating the complete Euclidean graph. In: Karlsson, R., Lingas, A. (eds.) SWAT 1988. LNCS, vol. 318, pp. 208–213. Springer, Heidelberg (1988)

[13] Keil, J.M., Gutwin, C.A.: Classes of graphs which approximate the complete Euclidean graph. Discrete and Computational Geometry 7, 13–28 (1992)

[14] Li, X.-Y.: Applications of computational geometry in wireless ad hoc networks. In: Cheng, X.-Z., Huang, X., Du, D.-Z. (eds.) Ad Hoc Wireless Networking, Kluwer Academic Publishers, Dordrecht (2003)

[15] Narasimhan, G., Smid, M.: Geometric spanner networks. Cambridge University Press, Cambridge (2007)

[16] Navarro, G., Paredes, R.: Practical construction of metric t-spanners. In: Proc. 5th Workshop on Algorithm Engineering and Experiments, pp. 69–81. SIAM Press (2003)

[17] Rao, S., Smith, W.D.: Approximating geometrical graphs via spanners and banyans. In: Proc. 30th ACM Symposium on the Theory of Computing, pp. 540–550. ACM, New York (1998)

[18] Sigurd, M., Zachariasen, M.: Construction of minimum-weight spanners. In: Albers, S., Radzik, T. (eds.) ESA 2004. LNCS, vol. 3221, Springer, Heidelberg (2004)