# Fast Greedy Algorithms for Constructing Sparse Geometric Spanners ⋆

Joachim Gudmundsson[1], Christos Levcopoulos[1], and Giri Narasimhan[2] †

[1] Department of Computer Science, Lund University, Box 118, 221 00 Lund, Sweden
{Joachim.Gudmundsson,Christos.Levcopoulos}@cs.lth.se
[2] Department of Mathematical Sciences, The University of Memphis, Memphis, TN 38152, USA,
Giri.Narasimhan@memphis.edu

**Abstract.** Let $G$=$(V, E)$ be a connected graph with positive weights and $n$ vertices. A subgraph $G'$ is a $t$-spanner if for all $u, v \in V$, the distance between $u$ and $v$ in the subgraph $G'$ is at most $t$ times the corresponding distance in $G$. We show an $O(n \log n)$-time algorithm which, given a set $V$ of $n$ points in $d$-dimensional space, and any constant $t > 1$, produces a $t$-spanner of the complete Euclidean graph of $G$. The produced spanner has $O(n)$ edges, constant degree, and weight $O(wt(MST))$.

## 1 Introduction

Complete graphs represent ideal communication networks but they are expensive to build; sparse spanners represent low cost alternatives. The weight of the spanner network is a measure of its sparseness; other sparseness measures include the number of edges, maximum degree and the number of Steiner points. 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 considerable research [1, 2, 4, 8, 11].

Consider a set $V$ of $n$ points in $\mathbb{R}^d$, where the dimension $d$ is a constant. A network on $V$ can be modeled as an undirected graph $G$ with vertex set $V$ and with edges $e = (u, v)$ of weight $wt(e)$. A Euclidean network is a geometric network where the weight of the edge $e = (u, v)$ is equal to the Euclidean distance $d(u, v)$ between its two endpoints $u$ and $v$. Let $t > 1$ be a real number. We say that $G'$ is a *$t$-spanner* for $V$, if for each pair of points $u, v \in V$, there exists a path in $G'$ of weight at most $t$ times the Euclidean distance between $u$ and $v$. A *sparse $t$-spanner* is defined to be a $t$-spanner of size $O(n)$ and weight $O(wt(MST))$, where $wt(MST)$ is the total weight of a minimal spanning tree. Given a geometric network $G = (V, E)$, a (generic) weight function $w$ defined on its edges, and two vertices $u, v \in V$, we let $D_{\{G,wt\}}(u, v)$ denote the weight of the shortest path from $u$ to $v$ in $G$ for the weight function $wt$.

The problem of constructing spanners has been investigated by many researchers. Levcopoulos and Lingas [10] presented an $O(n \log n)$-time algorithm for the 2-dimensional case that produced a sparse $t$-spanner. But the problem gets much more difficult in higher dimensions. There are several algorithms that run in time $O(n \log n)$ [3, 9, 14, 16]. However, they only guarantee a linear number of spanner edges and they do not guarantee low weight. Das and Narasimhan [8] gave an $O(n \log^2 n)$-time algorithm that constructs for any set $V$ of $n$

⋆ Parts of this work appeared as an extended abstract in the Proceedings of the Scandinavian Workshop on Algorithm Theory, Bergen, Norway, 2000, pages 314-327.
† Funded by NSF (CCR-940-9752) and Cadence Design Systems, Inc.

points in $\mathbb{R}^d$, and any constant $t > 1$, a sparse $t$-spanner for $V$ in which the degree of every point is bounded by a constant. Chen et al. [5] showed that the lower bound for computing any $t$-spanner for a given set of points $V$ in $R^d$ is $\Omega(n \log n)$ in the algebraic computation tree model.

Mount [12] has shown that a significant result claimed in Arya et al. [2] of an $O(n \log n)$-time algorithm to compute a sparse Euclidean spanner is incorrect. Thus the problem of devising an $O(n \log n)$-time algorithm to produce sparse spanners remained unsolved.

Sparse spanners are also useful in designing efficient approximation schemes for geometric problems. In a startling development, Rao and Smith [13] showed an optimal $O(n \log n)$-time approximation scheme for the well-known Euclidean *traveling salesperson problem*, assuming that it is possible to compute sparse spanners in time $O(n \log n)$. Also, Czumaj and Lingas [6] showed approximation schemes for minimum-cost multi-connectivity problems in geometric graphs that also depend on the assumption that sparse spanners can be computed in $O(n \log n)$ time. Since the claim by Arya et al. [2] was incorrect, the existence of an $O(n \log n)$-time algorithm to construct sparse spanners has become a critical open problem. Note that the most efficient algorithm to construct sparse spanners is due to Das and Narasimhan [8] and runs in $O(n \log^2 n)$ time. In this paper we design an algorithm that produces a $t$-spanner in time $O(n \log n)$, in the standard *real RAM model* defined in Chapter 1.

**Theorem 1.** *Given a set $V$ of $n$ points in $d$-dimensional space, and any real constant $t>1$, a $t$-spanner of the complete Euclidean graph can be constructed in $O(n \log n)$ time such that the spanner has $O(n)$ edges and weight $O(wt(MST))$. The constants implicit in the $O$-notation depend on $t$ and $d$.*

It was shown in [8] that the greedy algorithm produces spanners with $O(n)$ edges and weight $O(wt(MST))$. However, a naive implementation of the greedy algorithm (shown in Fig. 1) takes $O(n^3 \log n)$ time, mainly due to the fact that a quadratic number of shortest path queries are needed to be answered in a "dynamic" graph with $O(n)$ edges. Each of the queries takes $O(n \log n)$ time.

Our algorithm is inspired by the algorithm due to Das and Narasimhan [8]. They showed how to use clustering in order to speed up shortest path queries, i.e., they showed that approximate shortest path queries sufficed to produce sparse spanners. However, their algorithm was not efficient enough because they were unable to maintain the clusters efficiently and the algorithm had to frequently rebuild the clusters. For convenience, we will refer to the $O(n \log^2 n)$-time algorithm from [8] as the **DN-Clustering** spanner algorithm. We retain the general framework of that algorithm. Our main contribution is in developing techniques to efficiently perform clustering. We believe that the techniques that we have developed are likely to be useful in designing other greedy-style "dynamic algorithms", i.e., in situations where only insertions take place and particularly in increasing order of length. What we prove in this paper is that after some preprocessing, given a linear-sized edge-weighted graph with integral edge weights in the range $[0, N]$, and given a set of cluster centers, then one can perform clustering very efficiently in only $O(n + N)$ time.

The terms length and weight are used interchangeably throughout the paper.

## 2 The DN-clustering spanner algorithm

We first describe the previous cluster based spanner algorithm due to Das and Narasimhan [8]. It can be roughly described as follows.

```
Algorithm STANDARD-GREEDY(G, t)
    sort the edges in E by increasing weight
    E' := ∅
    G' := (V, E')
    for each edge (u, v) ∈ E do
        if SHORTESTPATH(G', u, v) > t · d(u, v) then
            E' := E' ∪ {(u, v)}
            G' := (V, E')
    output G'
```

**Fig. 1.** The naive $O(n^3 \log n)$-time greedy spanner algorithm

The algorithm starts with an empty spanner $G'$. A preprocessing step helps to eliminate all but a linear number of edges from further consideration. Among the edges not eliminated, very short edges (i.e., those of length at most $D/n$, where $D$ is the distance between the farthest pair of points) are simply added to $G'$ since their contribution to the overall weight of the spanner cannot be more than the weight of a minimum spanning tree, $wt(MST)$. For the remaining edges, the greedy algorithm is simulated by sorting the edges (by increasing weight) and then processing them in $\log n$ phases. Greedy processing of an edge $e = (u, v)$ entails a shortest path query, i.e., checking whether $D_{\{G',wt\}}(u, v) \leq t \cdot wt(e)$. If the answer to the query is no, then edge $e$ is added to the spanner $G'$, else it is discarded. Whenever shortest path queries are required to be answered, these are not solved on the spanner $G'$ being constructed. Instead, they are solved on a cluster graph $H$, which is simultaneously maintained. The cluster graph $H$ from [8] has the following properties:

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

Item 1. above was demonstrated by showing that corresponding distances in $H$ amd $G'$ differ by only a constant factor. The shortest path query when processing edge $e = (u, v)$ is "specialized" in the sense that, at the instant that this query is processed, the cluster graph $H$ only has edges between clusters (the so-called inter-cluster edges) whose lengths are within a constant factor of $wt(e)$.

In order to understand how shortest path queries can be answered efficiently, we note that in an unweigted graph with bounded degree, checking whether the distance between two vertices $u$ and $v$ is at most a constant $t$, can be achieved in $O(1)$ time (since there are only $O(1)$ vertices at distance $t$ from $u$). Thus, for all practical purposes, the cluster graph $H$ behaves like an unweighted graph of bounded degree for which a bounded radius subgraph around vertex $u$ needs to be searched for the presence of vertex $v$. It is thus easily shown that specialized shortest path queries can be answered in $O(1)$ time.

Since the edges considered have weights in the range $(D/n, D]$ and they are processed in $\log n$ phases, the edges can be sorted into $\log n$ bins, where the $i$-th bin has edges of weight in the range $(2^{i-1} \cdot D/n, 2^i \cdot D/n]$. In order for shortest path queries to be answered quickly, the cluster graph has to be carefully maintained. At the end of each phase, the cluster graph is recomputed from scratch using the graph $G'$. This was deemed necessary since, in order to answer specialized shortest path queries about edge $e{=}(u, v)$ in constant time, all inter-cluster edges in $H$ need to be of length within a constant factor of $d(u, v)$.

The time complexity analysis is straightforward. Preprocessing steps ran in $O(n \log n)$ time. The $O(n)$ shortest path queries were processed in $O(n)$ time, since each query took only $O(1)$ time. The cluster graph computation at the start of each phase took $O(n \log n)$ time (since it involved running Dijkstra's shortest path algorithm on linear-sized graphs). Since there were $\log n$ phases, the cluster graph computations took a total of $O(n \log^2 n)$ time. The crucial observation made in [8] was that shortest path queries need not be answered precisely. Instead, approximate shortest path queries suffice to produce low-weight spanners. The second observation was that shortest path queries are expensive if the shortest path involves a number of short edges, and that clustering can help to eliminate all short edges. This, of course, meant that the greedy algorithm, too, was only approximately simulated by the algorithm.

### 2.1 A faster spanner algorithm

In this section, we present a simple modification to the DN-clustering algorithm to construct sparse $t$-spanners. This algorithm improves on the time complexity of the DN-clustering algorithm and runs in time $O(\frac{n \log^2 n}{\log \log n})$ in the algebraic decision tree model of computation.

First we observe that there is wide disparity in the overall time spent by the DN-clustering algorithm on shortest path queries ($O(n)$) and the time spent on the cluster graph computations ($O(n \log^2 n)$). In order to balance out the two costs, it is necessary to do fewer than $O(\log n)$ cluster graph computations, which in turn would make the shortest path queries more expensive. Instead of processing the edges in $\log n$ phases, we process them in $\frac{4 \cdot d \cdot \log n}{\log \log n}$ batches. We use the term batches to distinguish from the word phases used by the earlier DN-clustering algorithm.

If the clustering is recomputed after processing every batch of edges the total time for cluster graph computations will be $O(\frac{n \log^2 n}{\log \log n})$, since each call to the clustering algorithm takes $O(n \log n)$ time. We carefully analyze the cost of the $O(n)$ shortest path queries and show that it can now be answered in a total of $O(n \log n)$ time. In phase $i$ of the DN-Clustering algorithm, edges from the $i$-th bin were processed. These edges had weights in the range $(W, 2W]$, where $W = 2^{i-1}(D/n)$. During phase $i$, the cluster graph $H$ could have inter-cluster edges whose weights were in the range $(\delta W, 2W(1 + 2\delta)]$, where $\delta < \frac{1}{2}$ is a positive constant. This meant that for edge $(u, v)$ of weight $l \in (W, 2W]$, checking whether there is path from $u$ to $v$ of length at most $t \cdot l$ could be done in $O(1)$ time. More precisely, it was observed in [8] that if there exists a path from $u$ to $v$ of length at most $t \cdot l$, then the number of edges on this path can be at most $\frac{2t}{\delta}$. It was further observed that since the vertices of $H$ had a constant degree bound (say $c$), and since there are at most $O(c^{\frac{2t}{\delta}})$ vertices that lie $\frac{2t}{\delta}$ edges away from vertex $u$, this shortest path query could be done in $O(c^{\frac{2t}{\delta}} \log c^{\frac{2t}{\delta}})$ time. A tighter analysis was unnecessary in the DN-Clustering algorithm of [8] since $c$, $t$, and $\delta$ were all constants; below we show an improved analysis of this cost.

Recall that our algorithm works in $\frac{4 \cdot d \cdot \log n}{\log \log n}$ batches. Batch $i$ of our algorithm can be described as follows. For $W = 2^{\frac{(i-1) \cdot \log \log n}{4 \cdot d}}(D/n)$, the edges processed in batch $i$ have weights in the range $(W, W2^{\frac{\log \log n}{4 \cdot d}}]$, i.e., they are in the range $(W, W(\log n)^{\frac{1}{4 \cdot d}}]$. This implies that, for edge $(u, v)$ of weight $l \in (W, W(\log n)^{\frac{1}{4 \cdot d}}]$, we need to check whether there is a path from $u$ to $v$ of length at most $t \cdot l$. During batch $i$, the cluster graph $H$ can have inter-cluster edges whose weights are in the range $(\delta W, (1+2\delta)W(\log n)^{\frac{1}{4 \cdot d}}]$. Thus, if there does exist such a path from $u$

to $v$, then the number of edges on this path can be at most $\frac{t(\log n)^{\frac{1}{4 \cdot d}}}{\delta}$. The crucial observation we make is that the vertices of the cluster graph correspond to clusters of radius $\delta W$. These clusters may overlap, but their centers can lie in only one cluster. In other words, if these clusters are shrunk in half, they do not intersect. Thus the vertices correspond to disjoint clusters of radius $\delta \cdot W/2$. Now, it is possible to bound the number of vertices within distance at most $t \cdot l = tW(\log n)^{\frac{1}{4 \cdot d}}$. A simple packing argument shows that the number of balls of radius $r$ that can be packed in a ball of radius $R$ is bounded by $O((R/r)^d)$, where $d$ is the dimension of the space. In our case, the number of balls of radius $r = \frac{\delta W}{2}$ that can be packed in a ball of radius $R = tW(\log n)^{\frac{1}{4 \cdot d}}$ is at most $O((\frac{t \cdot (\log n)^{\frac{1}{4 \cdot d}}}{\delta})^d)$. Thus, due to the constant degree, the maximum number of vertices and edges that can be reached when performing Dijkstra's algorithm starting from vertex $u$ is $O((\frac{t \cdot (\log n)^{\frac{1}{4 \cdot d}}}{\delta})^d)$. Since $t$, $d$ and $\delta$ are constants, we have that $O((\frac{t \cdot (\log n)^{\frac{1}{4 \cdot d}}}{\delta})^d) = O((\log n)^{\frac{1}{4}})$. We conclude that Dijkstra's algorithm for a shortest path query has a time complexity of $O((\log n)^{\frac{1}{4}} \cdot (\log((\log n)^{1/4}))) = O(\log n)$.

The obvious consequence is that all $O(n)$ shortest path queries can be answered in $O(n \log n)$ time, and hence, we have proved the following theorem:

**Theorem 2.** *In the algebraic decision tree model of computation, given a set $V$ of $n$ points in $d$-dimensional space, and any real constant $t>1$, a sparse $t$-spanner of the complete Euclidean graph can be constructed in $O(\frac{n \log^2 n}{\log \log n})$ time. The constants implicit in the $O$-notation depend on $t$ and $d$.*

## 3 A fast spanner algorithm

In the rest of the paper, we describe an efficient algorithm to construct sparse spanners with a running time of $O(n \log n)$. This algorithm is also inspired by the DN-clustering algorithm in [8]. As explained in Section 2 the reason why their algorithm runs in time $O(n \log^2 n)$ is that the clustering step takes $O(n \log n)$ time per phase. The running time for our algorithm is achieved by designing a linear time algorithm for an "approximate" version of the clustering step, thus executing all the clustering steps in $O(n \log n)$ total time.

One crucial idea that we employ to speed up the clustering is to replace the real-valued edge weights by integral values. As observed in [8], the shortest path queries required by the algorithm need not be answered precisely; approximately correct answers suffice. A convenient way to achieve the *integralization* is to use the *floor/ceiling* function. However, this assumes a more powerful model of computation. In order to get around this problem, we reduce the dependence of the algorithm on the floor/ceiling function and compute the floor/ceiling function by using operations allowed under the real RAM model. The second crucial component of our algorithm is an implementation of the clustering algorithm in $O(n)$ time assuming small integral edge weights for the edges. We also prove that the integralization introduces only a bounded amount of error, and that this error bound helps to prove the correctness of the other required operations.

The improved spanner algorithm can be roughly described as follows; see Fig. 2. It is important to note that the skeleton of the algorithm is similar to the DN-clustering algorithm from [8]. In particular, this improved algorithm also runs in $O(\log n)$ phases. If a fewer number of phases are used, then the error due to integralization could be too large. Even if a fewer number of phases can be used, the running time of the overall algorithm will remain as $O(n \log n)$, since it is dominated by other steps in the algorithm. In particular, the integralization itself has an initial cost of $O(n \log n)$.

The algorithm starts with an empty spanner $G'$ and employs the same first (preprocessing) step to eliminate all but a linear number of edges. For a given value of $t$, this step is performed by a call to an $O(n \log n)$-time algorithm presented by Arya et. al. in [2] to compute a spanner with stretch factor $\sqrt{t/t'}$ (for some $t > t' > 1$) and with bounded degree. As in [8], in the next stepm short edges of length at most $D/n$ are simply added to $G'$; their contribution to the overall weight of the spanner is bounded by $wt(MST)$. The greedy algorithm is then simulated on the remaining edges and the edges are added to the graph $G'$.

The edges of the graph have real-valued weights that are equal to the Euclidean distance between their endpoints. The edges are sorted by increasing weight and then processed in $\log n$ phases. Each of the edges in the spanner graph also have corresponding integer-valued weights that are sufficiently close approximations of the real-valued weights; these integer-valued weights change through the course of the algorithm, becoming coarser and coarser approximations as the algorithm progresses. In order to distinguish between the real- and integer-valued weights, we assume that there are two different weight functions defined on the edges of $G'$. For edge $e = (u, v)$, the real-valued weight function $wt(e)$, as mentioned before, is defined as the Euclidean distance $d(u, v)$ between $u$ and $v$. The integer-valued weight function denoted by $Iwt_i(e)$ is a function of $wt(e)$ and the phase number $i$. It is maintained during the execution of the algorithm as will be described later. Whenever the phase number is clear from the context, we use the simpler notation $Iwt(e)$ instead of $Iwt_i(e)$. Also, unless specified otherwise, we assume that when we refer to the weight of an edge, we are referring to the real-valued weight of the edge.

At the start of each phase, the integer-valued weight function $Iwt(e)$ is recomputed for this phase. Then a set of vertices of $G'$ are selected as cluster centers and a cluster graph $H$ is constructed from the current spanner graph $G'$, using the weight function $Iwt$. This cluster graph $H$ is a simpler graph than the graph $G'$ and distances between vertices in $H$ are reasonably close to distances between the same pair of vertices in $G'$. The difference of this from the one in [8] lies in the fact that the cluster centers have to be selected before the clustering is done and the clustering is done with the weight function $Iwt$. As mentioned before, we improve on the time complexity of this clustering step and show how it can be implemented to run in $O(n)$ time. Once the cluster graph $H$ is constructed, the algorithm processes the set of edges for that phase. Greedy processing of an edge $e = (u, v)$ entails, as before, a shortest path query, i.e., checking whether $D_{\{G',wt\}}(u, v) \leq t \cdot wt(e)$. This query is answered in constant time per query by performing an approximate shortest path query on the simpler graph $H$, and not on the partial spanner graph $G'$. If the answer to the query is 'no', edge $e$ is added to the graph $G'$, else it is discarded. Each of the steps is described in more detail in the rest of the paper.

In Section 3.1 we describe the integralization process and analyze the error due to it. The clustering algorithm is described in Section 3.2, and finally, in Section 3.3 we describe how to compute the shortest paths in the cluster graph and prove that the total running time of the algorithm is $O(n \log n)$.

The detailed algorithm is given below in figure 2. The inputs are, $V$, which is a set of $n$ points in $d$-dimensional space, and two constants $t$ and $t'$ such that $1 < t' \leq t$. As one can see, it is similar to the DN-clustering algorithm, except for the integralization steps (steps 9, 12 and 13) and the computation of the cluster centers (steps 10 and 20). In Sections 4 and 5 we will show that the output $G'$ indeed is a $t$-spanner, and that a suitable selection of the input parameter $t'$ will guarantee that $G'$ has small weight. Recall that the really time-critical step of this algorithm is the clustering step (step 14) which will be closely described in Section 3.2. Step 19 of the algorithm is needed to maintain the cluster graph $H$, i.e., edges between clusters that lie close to each other are added. A more detailed description of this step is also given in Section 3.2. Note that the two values bounding $\delta$ are decided in Lemma 7 and Lemma 8.

---

**Algorithm** Improved-Greedy($V, t, t'$)
1. Compute a $(\sqrt{t/t'})$-spanner $G = (V, E)$ using the algorithm from [2]
2. $\delta := \min\left(\frac{\sqrt{tt'}-(1+\epsilon)t'}{2(1+\epsilon)(\sqrt{tt'}+3t')}, \frac{\sqrt{tt'}-(1+\epsilon)}{2(\sqrt{tt'}(1+\epsilon)+5+7\epsilon+2\epsilon^2)}\right)$
3. $D :=$ length of longest edge in $E$
4. $E' = \{e \in E \mid wt(e) < D/n\}$
5. $G' := (V, E')$
6. $W_i := 2^{(i-1)} D/n$ for $i = 1, 2, \ldots, \log n$
7. **for** $i := 1$ to $\log n - 1$ **do**
8.     $E_i :=$ set of (sorted) edges of $E$ with weights in $(W_i, W_{i+1}]$
8'.     Build Integer tree with values $\{1, \ldots, cn\}$)
9. Integralize($E', 0$)
10. $C_1 :=$ Naive-Centers($G', r = \delta W_1$);
11. **for** $i := 1$ to $\log n$ **do**
12.     Integralize($E_i, i$)
13.     ReIntegralize($E'$)
14.     $H :=$ Cluster-Graph($G', Iwt, C_i, r = \delta W_i, R = W_i$)
15.     **for** each edge $e = (u, v) \in E_i$ in increasing order **do**
16.         **if** not Short-Path($H, u, v, \sqrt{tt'} \cdot d(u, v)$) **then**
17.             $E' := E' \cup \{e\}$
18.             $G' := (V, E')$
19.             AddIntraEdgesType2($u, v$)
20.     $C_{i+1} :=$ Update-Centers($H, i, C_i, r = \delta W_i$)
21. output $G'$

---

**Fig. 2.** The $O(n \log n)$-time spanner algorithm

### 3.1 Integralization

As mentioned before, in order to speed up the cluster graph computation, we replace the real-valued edge weights by integral values. The integralization changes in every phase. It is done in such a way that the edge weights and distances encountered in that phase are always in the range $[0, N]$, where $N = c \cdot n$ for some constant integer $c$. The choice of $c$ will dictate the errors introduced in the distance computations; this will be discussed later.

A closer inspection of a phase leads to the following simple observations. At the start of phase $i$, the spanner graph constructed so far has edges of weight at most $W_i$. During phase $i$, the edges considered for inclusion by the greedy algorithm are in the range $(W_i, 2W_i]$. The shortest path query for an edge of length $l$ involves checking whether the distance between a given pair of vertices is at most $t \cdot l$. Hence the longest paths that need to be dealt with during phase $i$ are of weight $t \cdot 2W_i$. The idea is to make the largest distance we consider in phase $i$ to correspond to the integer $c \cdot n$. To be on the safe side, since there are small errors in the distance computations, we set $2(t \cdot 2W_i)$ to correspond to $c \cdot n$. Thus, in phase $i$, unit integer length will correspond to real length of $U_i = \frac{4 \cdot t \cdot W_i}{c \cdot n}$.

Although a constant-time floor/ceiling function is not used in the algorithm, a convenient way to describe the integralization is as follows:

$$Iwt_i(e) := \left\lceil \frac{wt(e)}{U_i} \right\rceil.$$

We will describe below how the integralization step is performed.

**Error bounds:** Assuming the integralization defined above, we observe that the function $Iwt$ always involves a rounding up. Hence, $Iwt_i(e) \cdot U_i \geq wt(e)$. It follows that in phase $i$, the error in the length of any single edge of the spanner graph is at most $U_i$. In other words, $Iwt_i(e) \cdot U_i - wt(e) \leq U_i$. Note that this error is an additive or an absolute error. Since any simple path can use at most $n-1$ edges, it is also easy to see that the error in the length of any simple path of the spanner graph is less than $nU_i$. Another consequence is that given two simple paths $P_1$ and $P_2$, if $Iwt(P_1) = Iwt(P_2)$, then $|wt(P_1) - wt(P_2)| < nU_i$. It follows that $nU_i$ is also a bound on the error that can be introduced when running Dijkstra's single-source-shortest-path algorithm using the integral weights instead of the real weights. The following lemma formalizes this statement:

**Lemma 1.** *In phase $i$, given any $\epsilon > 0$ and two vertices $u$ and $v$ in $G'$ such that $D_{\{G',wt\}}(u,v) > W_i$, it holds that*

$$D_{\{G',wt\}}(u,v) \leq D_{\{G',Iwt\}}(u,v) \cdot U_i < (1+\epsilon) \cdot D_{\{G',wt\}}(u,v).$$

*Proof.* We give a sketch of the proof. Recall that the cluster radius is $r = \delta W_i$, where $\delta < 1/2$ is some positive constant. In phase $i$, for a path $P$ such that $wt(P) \geq W_i$, the error in computing its weight is at most $nU_i$. Thus the relative error is at most $nU_i/W_i = \frac{4t}{c}$. The proof follows by setting $\epsilon = \frac{4t}{c\delta} > \frac{4t}{c}$ and using the well-known property of Dijkstra's algorithm that the minimum value in the priority queue is monotonically non-decreasing. Note that $\epsilon$ can be made as small as desired by choosing an appropriate value of $c$. □

As a direct consequence we obtain the following important corollary.

**Corollary 1.** *For a path $P$ in $G'$ with $wt(P) \geq \delta W_i$, the absolute error in computing its weight is at most $nU_i$, and the relative error is at most $\frac{nU_i}{\delta W_i} = \epsilon$, for any $\epsilon > 0$.*

**Computing the integralization** Here we show how to compute the integer values of the weights of the edges over all phases in $O(n \log n)$ total time without using the floor/ceiling function.

We first observe that the spanner graph has at most $O(n)$ edges at the start of any phase. Consider a specific phase $i$. In this phase, for a specific edge $e$, since its integer value is in the range $[0, N]$ (where $N = c \cdot n$), $Iwt(e)$ can be computed in $O(\log n)$ time without the use of the floor/ceiling function by performing a binary search on the set of real values $j \cdot U_i$, for $j = 0, \ldots, N$. We assume that the function INTEGRALIZE$(E_i, i)$ performs this operation for each edge in the set $E_i$ in $O(\log n)$ time per edge.

If the above observations are used in a naive fashion for all edges, then the cost of integralization is $O(n \log n)$ just for one phase. Since the number of phases is not constant, the integralization would turn out to be too expensive. We have to show that the algorithm spends $O(\log n)$ time for computing the integralization of an edge weight over all the phases. The idea is to compute the integral value in $O(\log n)$ time when the edge is encountered for the first time. Integralizations of an edge for subsequent phases is done by calling REINTEGRALIZE, and are computed in constant time from the integer weights of the edge computed in the previous phase. If the integral weight of an edge is $I$ in phase $i$, then the integral weight of the edge in phase $i+1$ will be $I/2$ if $I$ is even, and $(I+1)/2$ if it is odd. This is correct since $U_{i+1} = 2U_i$, i.e., the integralization in phase $i+1$ is twice as coarse as that in phase $i$. Checking if an integer is odd or even cannot be done in constant time in the real RAM model, but can be easily accomplished by using $O(n)$ preprocessing. One way to accomplish this would be to build a balanced binary tree including $c \cdot n$ elements with the values $1, \ldots, cn$. Every element in the tree, with value $val$, also contains a pointer to the element in the tree containing the value $\lceil \frac{val}{2} \rceil$. Hence, by using $O(n)$ time preprocessing, the integral weight of an edge for the next phase can be computed in constant time. Another way to handle this problem would be to extend the model of computation with trigonometric functions, i.e., the sine function.

Note also that the relative error for an edge with newly computed weight is less than $U_{i+1}$, hence Lemma 1 still holds. It is clear that REINTEGRALIZE($E'$) performs its operation for each edge in the edge set $E'$ in $O(1)$ time per edge.

The above explanation proves that the integralization is computed in time $O(n \log n)$ for all edges over all phases. The integer weights are then used directly in the clustering algorithms described below.

### 3.2 Clustering the graph

Now we turn our attention to the main contribution of this paper, namely, how to construct a cluster graph in linear time. First some definitions. Here we assume that $G = (V, E)$ is a metric graph with a weight function $w$ defined on its edges $E$. The following definition of a cluster is modified from the one in [8] to allow for arbitrary weight functions. The definition of a cluster cover is also modified and is defined for a given set of cluster centers.

**Definition 1. Cluster, cluster center, and radius**
*Given a vertex $v \in V$ and a non-negative real value $r$,* CLUSTER$(G, v, r, w)$ *is defined as the set of all vertices $U \subseteq V$ such that $D_{\{G,w\}}(v, u) \leq r$ for all $u \in U$. The vertex $v$ is called the* cluster center *of this cluster and $r$ is called the* radius *of the cluster.*

**Definition 2. Cluster-cover**
*Given a set of cluster centers $C = \{v_1, v_2, \ldots, v_m\} \subseteq V$ and a radius $r$, the* CLUSTER-COVER$(G, C, r, w)$ *(if it exists) is a set of clusters $K = \{K_1, \ldots, K_m\}$ such that $K_i$, for $1 \leq i \leq m$, is a cluster with radius $r$ and cluster center $v_i$ and such that $K_1 \cup K_2 \cup \cdots \cup K_m = V$.*

The set $C$ and the radius $r$ will be chosen in such a way that the cluster cover always exists. In general, clusters in a cluster cover may overlap. We also modify the definition from [8] of a cluster graph so that it is a bit more general and is defined for a given set of clusters and for an arbitrary weight function.

**Definition 3. Cluster graph**
*Assume that $C = \{v_1, v_2, \ldots, v_m\} \subseteq V$ is a given set of cluster centers. For a given radius*

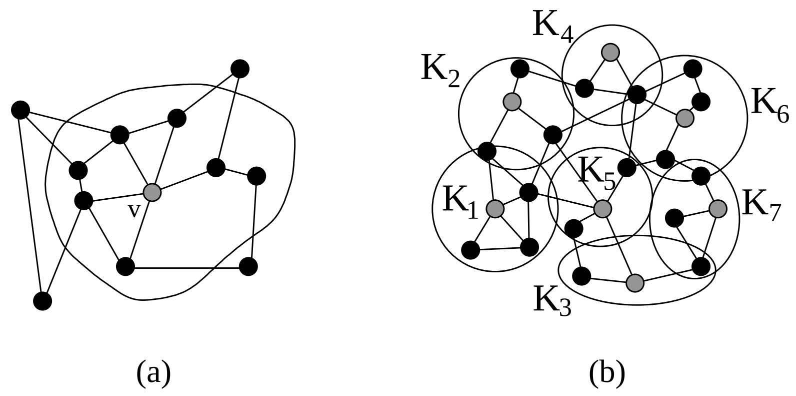


**Fig. 3.** (a) A cluster with center at $v$ and radius $r$. (b) The clusters $K_1, \ldots, K_7$ form a cluster cover.

*$r$, we assume that $K = \{K_1, K_2, \ldots, K_m\}$ is equal to* Cluster-Cover$(G, C, r, w)$. *Given a second radius $R > r$,* Cluster-Graph$(G, w, C, r, R)$ *is defined as a graph $H = (V, E_H)$ with a weight function $w$ defined on its edges $E_H$. The weight of an edge $[u, v]$ in $E_H$ is defined to be equal to $D_{\{G,w\}}(u, v)$ (we use square brackets to distinguish cluster graph edges from the edges of $G$). The edges of $H$ are defined as follows.*

**Intra-cluster edges:** *For all $K_i$, and for all $u \in K_i$, $[u, v_i] \in E_H$.*
**Inter-cluster edges:** *For all $v_i, v_j \in C$, $[v_i, v_j]$ is an inter-cluster edge if either:*

1. *$v_i \notin K_j$ and $v_j \notin K_i$ and $D_{\{G,w\}}(v_i, v_j) \leq R$ (Type 1), OR*
2. *there exists $e = (u_i, u_j) \in E$ such that $u_i \in K_i$ and $u_j \in K_j$ (Type 2).*

**Computing the cluster cover:** Here we describe how the cluster cover is computed efficiently under some assumptions. Once a cluster cover is computed we show later that it is straight-forward to construct the cluster graph.

Note that the input to the cluster cover computation is a weighted graph $G = (V, E)$ with a weight function $w$ defined on its edges, a set $C \subseteq V$ of cluster centers and a radius $R$. We will assume that $|V| = n$, $|E| = O(n)$, the weight function $w$ is integral, and the radius $R$ is an integer. Since we do not have to deal with distances greater than $R$, we can safely assume that the weight of any edge is an integer value in the range $[0, R]$. We will further assume that the cluster centers are chosen in such a way that a cluster cover exists, which will be shown in Section 3.3. The obvious way to implement this algorithm is as was done in [8], i.e., to run Dijkstra's Single-Source-Shortest-Path algorithm from all the cluster centers and to compute the clusters in the cluster cover. However, this has a running time of $O(n \log n)$. In order to speed it up, we run Dijkstra's algorithm in **parallel** from all the cluster centers and use a simple and fast priority queue, which we denote by $PQ$. The priority queue we use is an array of size $R$, indexed from 1 to $R$, as shown in Fig. 4. This is sufficient for our purposes because of the following reasons. Firstly, the weight function is integral and the array contains all possible distance values from the cluster centers to vertices in the clusters. Secondly, it is well-known that in Dijkstra's algorithm, once a vertex has been extracted from the priority queue, its distance from the source will never be updated again and the distance from the source at the time of the extraction is the correct distance from the source. In other words, the minimum value of the items in the priority queue is monotonic. Since the priority queue

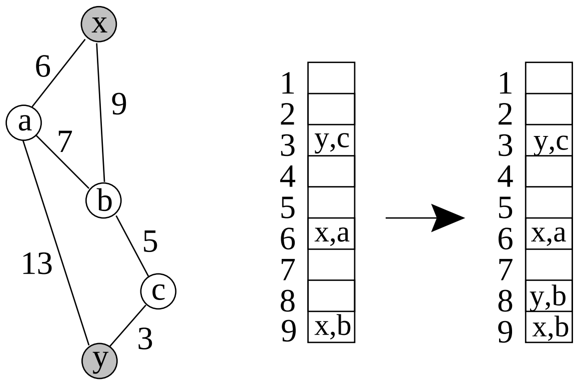


**Fig. 4.** An example of how the cluster cover is computed with $x$ and $y$ as cluster centers and radius $R = 9$. The array to the left is the initial priority queue after all edges leading out of $x$ and $y$ have been processed. The array to the right shows the final priority queue. Note that after vertex $c$ was processed, the edge $(c, b)$ was relaxed and $(y, b)$ was inserted into the priority queue with length 8.

is an array, EXTRACT-MIN can be implemented as a scan through the array for the "next" largest item.

One problem is that clusters can overlap and that vertices may have entries in the priority queue with distances from several cluster centers. Let $c_v$ denote the maximal number of clusters that a vertex may belong to. The problem can be taken care of by augmenting the priority queue entries to be a pointer to a linked list where every entry in the list also stores information about the vertex as well as the corresponding cluster center. Since a vertex at most belongs to $c_v$ clusters the space complexity of the priority queue will be $O(n \cdot c_v + R)$. Also, every vertex contains a list of the clusters it belongs to.

It should be noted that Dijkstra's algorithm, as shown below, needs to perform a number of RELAX steps and that in each such step the priority queue may need to be updated. The process of RELAXing an edge $(u, v)$ consists of testing whether we can improve the shortest path to $v$ found so far by going through $u$ and, if so, updating the value for $v$, i.e., adding a new entry and removing an old entry. Since every vertex contains information about which clusters it belongs to and the distance to each cluster center, each update is performed in time $O(c_v)$. It should be pointed out that this is the only place where we are unable to eliminate the use of *Random Access* since it is critical that this update be performed efficiently, i.e., in constant time. Also note that an edge $(u, v)$ may be RELAXed several times ($O(c_v)$ times), each time with respect to a different cluster center.

**Algorithm** PARALLELDIJKSTRA

| | |
|---|---|
| Q=INITIALIZE($G'$, cluster centers) | $O(n)$ |
| **while** $Q \neq \emptyset$ **do** | $O(n \cdot c_v)$ |
| $u$ =EXTRACTMIN($Q$) | |
| **for** each vertex $v$ adjacent to $u$ **do** | $O(1)$ |
| RELAX($u, v, Iwt$) | $O(c_v)$ |

**end**

Thus the time and space complexity of the algorithm is affected by the amount of overlap of the clusters in the cluster cover. The space complexity of the data structure is $O(n \cdot c_v +$

$R$). Furthermore, the RELAX-operation has a running time of $O(c_v)$ and, the total number ($O(n \cdot c_v)$) of EXTRACTMIN-operations can be performed in total time $O(n \cdot c_v + R)$. A careful implementation of cluster cover can be made to run in time $O(n \cdot c_v^2 + R)$.

**Computing the cluster graph:** Now we are ready to describe how to compute the cluster graph. The input is a weighted graph $G$ with a weight function $w$, a set of cluster centers $C = \{v_1, \ldots, v_m\}$, and two different radii $r$ and $R$, where $R > r$. In order to compute the cluster graph, the algorithm computes a cluster cover from the same set of cluster centers but with the two radii, $r$ and $R$. Let the cluster covers with radii $r$ and $R$ be denoted by $K_r$ and $K_R$ respectively. We augment the cluster cover procedure to also produce a data structure that supports the following queries for both the cluster covers:

**FINDCENTERS$(v, K)$:** Given $v \in V$, it returns all cluster centers $v_i$ such that $v$ is in a cluster from $K$ centered at $v_i$, i.e., $D_{\{G,w\}}(v, v_i)$ is at most the radius of the clusters in $K$. It also returns $D_{\{G,w\}}(v, v_i)$ for these cluster centers.

**COMPUTEDISTANCE$(v_i, v)$:** Given $v \in V$, and a cluster center $v_i$, it returns $D_{\{G,w\}}(v, v_i)$ if $D_{\{G,w\}}(v, v_i) \leq R$; otherwise, it returns the value $\infty$.

Now the cluster graph $H = (V, E_H)$ is computed easily as follows. The intra-cluster edges of $H$ are computed by performing FINDCENTERS queries for each vertex $v \in V$ in the cluster cover $K_r$ and adding the corresponding edges. Recall that FINDCENTERS returns a set $T$ of 2-tuples, where each tuple $t$ consists of a vertex $t.u$ and its distance $t.d$ to $v$. The algorithm is described below in pseudo code, and has a running time of $O(n \cdot c_v)$.

```
Algorithm IntraEdges
  for every vertex v ∈ V do
    T :=FindCenters(v, K_r)
    for every element t ∈ T do
      AddIntraEdge(v, t.u, t.d)
end
```

From Definition 3 we have that the inter-cluster edges can be of two types, type 1 and 2. An edge $[v_i, v_j]$ of type 1 is added if $v_i \notin K_j$ and $v_j \notin K_i$ and $D_{\{G,w\}}(v_i, v_j) \leq R$, where $K_i, K_j \in K_r$ are clusters with centers at $v_i$ and $v_j$. For every cluster center $v_i$, we use the FINDCENTERS query to list all the clusters from $K_R$ that it is contained in. The centers $v_j$ of these clusters satisfy the condition that $D_{\{G,w\}}(v_i, v_j) \leq R$. Now we use the COMPUTEDISTANCE queries to make sure that $v_i \notin K_j$ and $v_j \notin K_i$. Adding the inter-cluster edges of type 1 is done in time $O(n \cdot c_v^2)$.

```
Algorithm InterEdgesType1
  for every cluster center v in K_r do
    T :=FindCenters(v, K_R)
    for every element t ∈ T do
      If ComputeDistance(v, t.u) ≥ r then
        AddInterEdge(v, t.u, t.d)
end
```

The time complexity of computing the cluster graph is $O(n \cdot c_v^2)$. Having the cluster centers selected before performing the clustering enables clusters to be grown in "parallel" and thus the above algorithm is able to use one common priority queue to grow all the clusters, and is consequently able to perform the clustering efficiently.

**Maintaining the cluster graph during a phase:** An edge $[v_i, v_j]$ of type 2 is added if there exists an edge $e = (u_i, u_j) \in E$ such that $u_i \in K_i$ and $u_j \in K_j$. During the computation of the cluster graph $H$ at the start of a phase, only intra-cluster edges and inter-cluster edges of type 1 are added. Additional edges may be added during a phase of the greedy algorithm. Every time the greedy algorithm decides to add an edge $e = (u, v)$ to the partial spanner graph, several inter-cluster edges of type 2 may be added to $H$. This is achieved as follows: for every edge $e = (u_i, u_j)$ that is to be added to $G'$, perform FINDCENTERS queries for $u_i$ and $u_j$ from $K_r$ and join the corresponding cluster centers by inter-cluster edges in $H$. The weight of such edges are computed by performing two COMPUTEDISTANCE queries for $u_i$ and $u_j$ with the corresponding cluster centers and adding it to the weight of $(u_i, u_j)$. Note that this gives a safe upper bound on the true distance between $u_i$ and $u_j$. The ratio of $R$ to $r$ determines the maximum error represented by the bound. It is clear that the function below runs in $O(c_v^2)$ time, and it is performed $O(n)$ times.

**Algorithm** INTEREDGESTYPE2$(u_i, u_j)$
  $T1$ :=FINDCENTERS$(u_i, K_r)$
  $T2$ :=FINDCENTERS$(u_j, K_r)$
  **for** every $t1 \in T1$ **do**
    **for** every $t2 \in T2$ **do**
      ADDINTEREDGE$(t1.u, t2.u, t1.d + w(u_i, u_j) + t2.d)$
**end**

**Selecting the cluster centers for a phase:** In order for the CLUSTER-GRAPH function to be implemented efficiently, it needs to have the set of cluster centers as input. For the first phase, the cluster centers $C_1$ are identified in a greedy fashion using the weighted graph $G' = (V, E')$ with real-valued edge weights, and using a radius of $r = \delta W_1$. That is, select any point $v \in V$, not belonging to any clusters already computed, as a cluster center, compute the cluster with center at $v$. Continue this procedure until all points in $V$ belong to a cluster. This is referred to as NAIVE-CENTERS in the algorithm given in Fig. 2. NAIVE-CENTERS runs in $O(n \log n)$ time, since this can be implemented using the standard Dijkstra's algorithm. For subsequent phases, cluster centers are identified (using UPDATECENTERS) in a different way. The set of cluster centers are always chosen as a subset of the cluster centers used in the previous phase.

At the end of each phase, the algorithm selects a set of cluster centers for the next phase. These centers are guaranteed to be sufficiently far apart from each other. More specifically, the cluster centers $C_i$ used in phase $i$ are guaranteed to be at a distance of at least $r_i/2$.

In phase $i$, the set of cluster centers for phase $i+1$ is computed as $C_{i+1} := C_i \backslash M_i$, i.e., a subset $M_i$ of the cluster centers are deleted from the list of cluster centers. We now describe how the set $M_i$ is chosen. $M_1$ is the empty set, implying that $C_2$ is identical to $C_1$. For $i > 1$, the algorithm picks a cluster center from $C_i$ and marks all cluster centers that are within distance $r$ from it. The cluster centers that are marked are inserted into $M_i$ and hence, deleted in the next phase. This is easily implemented by calling the FINDCENTERS after the cluster cover for phase $i$ has been computed. The next cluster center is then picked and the process continues until all centers have been processed. Finally, $C_{i+1} := C_i \backslash M_i$. Clearly this process runs in time $O(m \cdot c_v)$. (It is important to note that since the integralization changes in every iteration, vertices that are at distance $r'$ in one iteration are at distance $r'/2$ in the next iteration.)

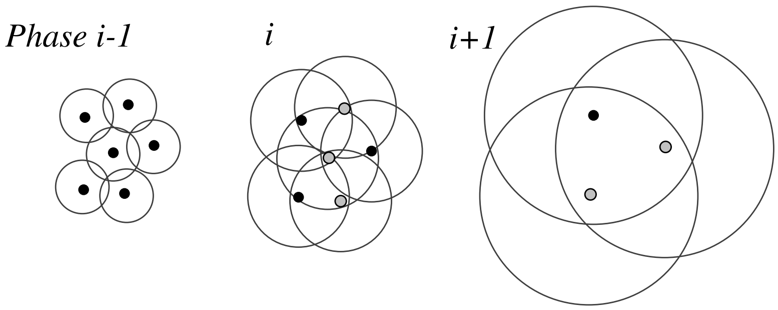


**Fig. 5.** If a cluster contains another cluster center then this cluster center is marked for deletion in the next phase. The figure shows an example of a set of cluster centers where the cluster centers that are marked for deletion for the next phase are marked in grey.

We now show that in phase $i$ the cluster centers are guaranteed to be at a distance of at least $r/2$ from each other. In phase 1, since cluster centers are identified by using a radius of $r$, all cluster centers are at a distance of at least $r$ from each other. In phase $i-1$, if two cluster centers are at a distance of $r$ or less, then one of them will get marked, and will subsequently be deleted from the list $C_i$ for phase $i$, as shown in Fig. 5. Lemma 2 specifies conditions under which vertices belong to at most a constant number of clusters.

It follows from this lemma that no vertex of $H$ is in more than a constant number of clusters of radius $r$ or of radius $R$ (since $\frac{R}{r} = \frac{1}{\delta}$).

**Lemma 2.** *Let $C = \{v_1, \ldots, v_m\} \subseteq V$ be a set of vertices such that for any pair of vertices $v_i, v_j \in C$, $D_{\{G,w\}}(v_i, v_j) > r'$. If $K = \{K_1, \ldots, K_m\}$ is returned by* CLUSTER-COVER$(G, C, c' \cdot r')$*, where $c'$ is a constant, then each vertex $v \in V$ is contained in at most a constant (which depends on the dimension $d$ and $c'$) number of clusters from $K$.*

The conditions of the lemma are true for the cluster graph as constructed above with $r' = r/2$ and $c' = 2$ or $c' = 4t/\delta$. Hence any vertex in $H$ is part of at most a constant number of clusters in $K_r$ or $K_R$. The proof follows from standard packing arguments, see also Section 2.1. Similar arguments also show that the number of inter-cluster edges incident to a cluster center is also a constant (although it might have a large number of intra-cluster edges). It follows that the degree of any vertex in $H$ that is not a cluster center must be a constant, and the size of $H$ is $O(n)$. Thus $c_v$ and $c_e$ are also bounded by a constant. Note that IMPROVED-GREEDY use integralized weights so the resulting clusters are approximate clusters; they are a little bit larger (since integers are always rounded up) than the exact clusters. It is clear that this does not affect the correctness of Lemma 2.

### 3.3 Answering shortest path queries

When the algorithm IMPROVED-GREEDY considers an edge $e = (u, v)$ for inclusion in the spanner graph, it needs to answer a shortest path query. It needs to check if $D_{\{G',wt\}}(u, v) \leq t \cdot d(u, v)$, where $G'$ is the spanner graph constructed so far. As noted in [8], it is sufficient for this query to be answered approximately. So, it is sufficient to devise a procedure to efficiently check if $D_{\{G',wt\}}(u, v) \leq t(1+\epsilon') \cdot d(u, v)$, for some small $\epsilon' > 0$. In other words, it is sufficient to check if $D_{\{G',Iwt\}}(u, v) \leq t(1+\epsilon'') \cdot d(u, v)/U_i$, for some small $\epsilon'' > 0$. In fact, the algorithm will check if $D_{\{H,Iwt\}}(u, v) \leq t \cdot d(u, v)/U_i$. The time complexity of this test is constant if $D_{\{H,Iwt\}}(u, v) < c' \cdot r$, for some constant $c'$. Hence, we conclude this section with the following theorem, which follows from the above arguments.

**Theorem 3.** IMPROVED-GREEDY *runs in time* $O(n \log n)$.

*Proof.* Following the steps of the IMPROVED-GREEDY algorithm shown in Fig. 2, we have that step 1 in the initialization and steps 8 and 9 take time $O(n \log n)$; step 2 take constant time; step 6 run in $O(\log n)$ time, while step 3 takes $O(n)$ time. The integralization of the weight of the edges from steps 9, 12 and 13, takes a total of $O(\log n)$ time per edge. Since each edge is considered exactly once the total time spent on integralizing and re-integralizing the weight of the edges is $O(n \log n)$ according to Section 3.1. Step 14 requires linear time since $c_v$ is bounded by a constant. On line 15 every edge in the input spanner is considered once. For each edge the algorithm performs one shortest path query in the cluster graph. As mentioned above, each query takes constant time. Hence the total time complexity for computing a linear number of shortest path queries is $O(n)$. Finally, updating the centers is easily done in linear time. From this it follows that IMPROVED-GREEDY runs in time $O(n \log n)$.

□

In 1999, Thorup [15] showed that single source shortest path queries could be answered in linear time for undirected graphs with integer edge weights. However this algorithm was not used in this paper since it does not visit the vertices in order of increasing distance, which is crucial for our algorithm. Also, it uses bit-shift for computing the floor function in constant time, which we do not allow in the real RAM model.

## 4 The graph produced by IMPROVED-GREEDY is a $t$-spanner

In order to show that the produced spanner graph $G'$ is a $t$-spanner we need two main results. First, we need to show that the cluster graph $H$ approximates the spanner graph $G'$, i.e., $D_{\{G',wt\}}(v, u) \leq D_{\{H,Iwt\}}(v, u) \cdot U_i \leq \alpha D_{\{G',wt\}}(v, u)$ for some constant $\alpha$ close to 1. This is done in Lemmas 5 and 6. Secondly, we need to show, Lemma 7, that $H$ is always a valid cluster graph of $G'$. From these results we easily obtain Theorem 4, which says that the produced spanner $G'$ is a $t$-spanner of the complete Euclidean graph.

Since the clusters are computed using the function $Iwt(\cdot)$ instead of $wt(\cdot)$, clusters are not as precise as they were in [8]. In this section we will assume that the smaller radii $r_i$ is $\delta W_i$ and the larger radii $R_i$ is $W_i$, where $\delta$ is a positive constant decided in Lemma 7 and 8. Finally, we set $\epsilon = \frac{nU_i}{\delta W_i}$. Some of the results in this section, and the next, are modified versions of analogous results in [8].

**Lemma 3.** *Let $K$ be equal to* CLUSTER$(G', v, \delta W_i, Iwt_i)$, *i.e., a cluster with cluster center $v$ and radius $\delta W_i$ computed in iteration $i$ of the algorithm. If $u$ is a vertex in $K$, then $D_{\{G',wt\}}(v, u) \leq (1 + \epsilon)\delta W_i U_i$. Otherwise, if $u \notin K$, then $D_{\{G',wt\}}(v, u) > \delta W_i U_i$.*

*Proof.* The lemma follows from Corollary 1 and the fact that a cluster from $K$ with center at $v$ consists of all vertices within integer distance $\delta W_i$ from $v$.

□

Consider the cluster graph $H$ that results from the clustering performed on $G'$ at the start of phase $i$. The following results apply to edges and paths in $H$.

**Lemma 4.** *If $u$ is a cluster center and $[u, v]$ is an intra-cluster edge in $H$, then*

$$D_{\{G',wt\}}(u,v) \leq (1+\epsilon)\delta W_i U_i. \tag{1}$$

*If $[v_j, v_k]$ is an inter-cluster edge in $H$, then*

$$\delta W_i U_i < D_{\{G',wt\}}(u,v) \leq (1+\epsilon)\cdot(W_i + 2\delta W_i)U_i. \tag{2}$$

*Proof.* The first statement is a direct consequence of Lemma 3, the same holds for the left inequality in (2). The right inequality in the second statement follows from Definition 3 since an inter cluster edge of type 2 may be constructed to connect two cluster centers $v_j$ and $v_k$ since there exists an edge $(x, y) \in G'$ such that $x \in K_j$ and $y \in K_k$ and $Iwt(x, y) < W_i$. □

For simplicity in the rest of this section we will leave out the unit length $U_i$. The following lemma is straight-forward since $H$ is an approximation of $G'$.

**Lemma 5.** *If there exists a path $P_H$ in $H$ between vertices $u$ and $v$ such that $Iwt(P_H) = L$, then there exists a path $P_{G'}$ in $G'$ between vertices $u$ and $v$ such that $Iwt(P_{G'}) \leq L$.*

We first introduce some definitions. A vertex $u$ is defined to *sufficiently far* from a vertex $v$ if, (1) no single cluster contains both $u$ and $v$, and (2) $D_{\{G',wt\}} \geq W_i$. Define a *cluster path* in $H$ to be a path where the first and last edges may be intra-cluster edges, but all intermediate edges are inter-cluster edges.

The next lemma is the approximate converse of Lemma 5.

**Lemma 6.** *Let $u$ be sufficiently far from $v$. Let $P_{G'}$ be a path between $u$ and $v$ in $G'$ such that $wt(P_{G'}) = L_1$. Then there exists a cluster path $P_H$ between $u$ and $v$ in $H$ such that*

$$Iwt(P_H) = L_2 < L_1 \cdot \frac{(1+\epsilon)(1+6\delta)}{1-2\delta(1+\epsilon)}.$$

*Proof.* Let the path from $u$ to $v$ having weight $L_1$ in $G'$ be $P$. We shall use the notation $P(y, x)$ to denote the vertices of $P$ between vertices $y$ and $x$, not including $y$. We construct a cluster path $Q$ from $u$ to $v$ in $H$ with weight $L_2$ as follows. Let $C_0$ be any cluster, with center $v_0$, containing $u$. The first edge of $Q$ is the intra-cluster edge $[u, v_0]$. Next, among all clusters with centers adjacent to $v_0$ in $H$, let $C_1$, with center $v_1$, intersect the furthest vertex along $P(u, v)$, say $w_1$. Add the inter-cluster edge $[v_0, v_1]$ to $Q$. Next, among all clusters with centers adjacent to $v_1$ in $H$, let $C_2$, with center $v_2$, intersect the furthest vertex along $P(w_1, v)$, say $w_2$. Add the inter-cluster edge $[v_1, v_2]$ to $Q$. This process continues until we reach a cluster center, $v_m$, whose cluster contains $v$. At this stage complete $Q$ by adding the intra-cluster edge $[v_m, v]$, as shown in Fig 6. We now prove that the weight of $Q$ satisfies the lemma statement.

**Case 1.** $[m = 1]$. In this case there is only one inter-cluster edge along $Q$. Since $u$ is sufficiently far from $v$, we know that $L_1 > W_i - 2(1+\epsilon)\delta W_i$. Now $L_2 = Iwt([u, v_0]) +$

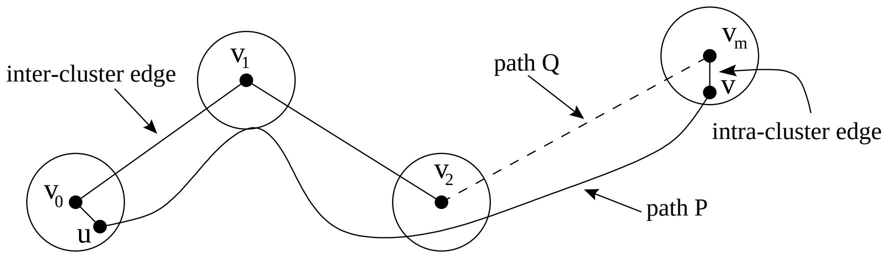


**Fig. 6.** Paths in $H$ approximate paths in $G'$.

$Iwt([v_0, v_1]) + Iwt([v_1, v])$. But $Iwt([u, v_0]) \leq (1+\epsilon)\delta W_i$ and $Iwt([v, v_1]) \leq (1+\epsilon)\delta W_i$, while $Iwt([v_0, v_1]) \leq 2(1+\epsilon)\delta W_i + D_{\{G', Iwt\}}(v, u) \leq 2(1+\epsilon)\delta W_i + (1+\epsilon)L_1$. This result follows from the procedure AddInterEdgesType2, since $wt([v_0, v_1])$ is at most $2(1+\epsilon)\delta W_i$ plus the length of the shortest edge connecting vertices of the two clusters to which $u$ and $v$ belong. So $L_2 \leq (1+\epsilon)L_1 + 4(1+\epsilon)\delta W_i$, and we have that $W_i < \frac{(1+\epsilon)L_1}{1-2(1+\epsilon)\delta}$. Combining these inequalities, we get

$$L_2 \leq (1+\epsilon)L_1 + \frac{4(1+\epsilon)\delta}{1-2(1+\epsilon)\delta} \cdot L_1 < \frac{(1+\epsilon)(1+2\delta)}{1-2\delta(1+\epsilon)} \cdot L_1.$$

**Case 2.** [$m \geq 2$ and $m$ is even.] Suppose $[v_i, v_{i+1}]$ and $[v_{i+1}, v_{i+2}]$ are any two consecutive inter-cluster edges on $Q$. Observe that the sum of their weights is greater than $W_i$. If this were not so, then the edge $[v_i, v_{i+2}]$ would instead have been added to $Q$ while $Q$ was being constructed. Divide $Q$ into portions $Q_0, Q_1, \ldots$, where $Q_{2i}$ is the portion between $v_{2i}$ and $v_{2i+2}$. Similarly, divide $P$ into portions $P_0, P_1, \ldots$, where $P_{2i}$ is the portion between the last vertex intersecting $C_{2i}$ and the first vertex intersecting $C_{2i+2}$. We shall first prove that for any even $i$, the weight of $Q_{2i}$ is no more than a constant times the weight of $P_{2i}$.

Let the weight of $P_{2i}$ be $p_{2i}$ and that of $Q_{2i}$ be $q_{2i}$. Since there cannot be an inter-cluster edge between $v_{2i}$ and $v_{2i+2}$, we have that $p_{2i} > W_i - 2\delta(1+\epsilon)W_i$. Select $r$ to be any vertex of $P_{2i}$ within the intermediate cluster $C_{2i+1}$. The vertex $r$ splits $P_{2i}$ into two portions. Let $p^1_{2i}$ (respectively $p^2_{2i}$) be the initial (respectively final) portions; thus $p_{2i} = p^1_{2i} + p^2_{2i}$. From the procedure AddInterEdgesType2 we have $Iwt([v_{2i}, v_{2i+1}]) \leq (1+\epsilon)p^1_{2i} + 2\delta(1+\epsilon)W_i$, and similarly $Iwt([v_{2i+1}, v_{2i+2}]) \leq (1+\epsilon)p^2_{2i} + 2\delta(1+\epsilon)W_i$. Adding the two we get $q_{2i} \leq (1+\epsilon)p_{2i} + 4\delta(1+\epsilon)W_i$. We now have two inequalities relating $p_{2i}$, $q_{2i}$ and $W_i$. We have that $W_i < \frac{p_{2i}}{1-2(1+\epsilon)\delta}$, which gives us

$$q_{2i} < (1+\epsilon)p_{2i} + 4\delta(1+\epsilon) \cdot \frac{p_{2i}}{1-2\delta(1+\epsilon)} < p_{2i} \cdot \frac{(1+\epsilon)(1+2\delta)}{1-2\delta(1+\epsilon)}.$$

Summing over all even values of $i$, and taking into account the two intra-cluster edges at either ends of $Q$, we get

$$L_2 < L_1 \cdot \frac{(1+\epsilon)(1+2\delta)}{1-2\delta(1+\epsilon)} + 2\delta(1+\epsilon)W_i.$$

Since $u$ is sufficiently far from $v$, we know that $L_1 > W_i - 2\delta(1+\epsilon)W_i$. That is, $\frac{L_1}{1-2\delta(1+\epsilon)} > W_i$. Substituting this in the above inequality we obtain

$$L_2 < L_1 \cdot \frac{(1+\epsilon)(1+4\delta)}{1-2\delta(1+\epsilon)}.$$

**Case 3.** [$m \geq 3$ and $m$ is odd.] The analysis will be exactly the same as in the previous case, except that we have to account for the last inter-cluster edge along $Q$ and correspondingly the portion of $P$ between the last two clusters. Let $q_{m-1}$ be the integer weight of $[v_{m-1}, v_m]$, and let $p_{m-1}$ be the weight of the portion of $P$ between the last vertex intersecting $C_{m-1}$ and the first vertex intersecting $C_m$. Clearly $q_{m-1} \leq (1+\epsilon)p_{m-1} + 2\delta(1+\epsilon)W_i$. This inequality does not change if we rewrite it as $q_{m-1} < (\frac{(1+\epsilon)(1+4\delta)}{1-2\delta(1+\epsilon)})p_{m-1} + 2\delta(1+\epsilon)W_i$. We then sum up as above, and get

$$L_2 < L_1 \cdot \frac{(1+\epsilon)(1+2\delta)}{1-2\delta(1+\epsilon)} + 4\delta(1+\epsilon)W_i.$$

Since $L_1 > W_i - 2\delta(1+\epsilon)W_i$, we have that $L_1 \cdot \frac{4\delta(1+\epsilon)}{1-2\delta(1+\epsilon)} > 4\delta(1+\epsilon)W_i$. Substituting this in the above inequality we obtain

$$L_2 < L_1 \cdot \frac{(1+\epsilon)(1+6\delta)}{1-2\delta(1+\epsilon)}.$$

□

Before processing any group $E_i$, the algorithm constructs a fresh cluster graph $H$. It uses a radius of $\delta W_i$, where $(W_i, 2W_i]$ is the interval that bounds the edge weights of $E_i$.

**Lemma 7.** *During the processing of any group $E_i$, the graph $H$ always represents a valid cluster graph of $G'$.*

*Proof.* Let the edges in $E_i$ be ordered by increasing weight as $e_{i1}, \ldots, e_{il}$. The proof is by induction. In the base case, when none of the edges have been processed, the lemma is obviously true. Now assume that the lemma is true just before the algorithm decides to examine edge $e_{ij}=(u,v)$. If this edge is not added to $G'$, then the lemma still holds. Now, suppose this edge is added to $G'$. Since $Iwt(u,v) > W_i$ and $\delta<1/2$, the distance between any two previous cluster centers in the new $G'$ will remain greater than $\delta W_i$, and thus the previous cluster cover will remain valid. Also the previous intra-cluster edges, and the inter-cluster edges of type 1 (see the definition of inter-cluster edges) will remain the same. We only have to make sure that we add new inter-cluster edges of type 2, and it is easily seen, that this is done by the algorithm. It remains to decide what weights are to be assigned to these new inter-cluster edges in $H$. Consider one such edge $[x,y]$, where $x$ (respectively $y$) is the center of the cluster to which $u$ (respectively $v$) belongs. The weight of this edge should be assigned $D_{\{G',Iwt\}}(x,y)$ (the shortest path in the new graph $G'$ between $x$ and $y$). However, it will be too time consuming to compute this directly. Instead the algorithm assigns the weight as $Iwt([x,u]) + Iwt(u,v) + Iwt([y,v])$ (see Section 3.2). We now show that our choice of $\delta$ makes this acceptable.

Assume the contrary, i.e., that a shorter alternate path $P$ exists between $x$ and $y$. Let $Iwt(P)$ denote its integral weight in this phase. Since $P$ cannot involve the edge $(u,v)$, it only contains edges of the previous $G'$. But we know that $(u,v)$ was selected to be added to $G'$, thus no cluster path existed between $u$ and $v$ of weight within $\sqrt{tt'} \cdot Iwt(u,v)$ in $H$. Furthermore, since $u$ is sufficiently far from $v$, we may use Lemma 6 to get

$$Iwt(P) + 2\delta W_i(1+\epsilon) > \frac{1-2\delta(1+\epsilon)}{(1+\epsilon)(1+6\delta)} \cdot \sqrt{tt'} \cdot Iwt(u,v).$$

We have that $Iwt(P) < Iwt([x,u]) + Iwt(u,v) + Iwt([v,y])$ which is at most $Iwt(u,v) + 2\delta W_i(1+\epsilon)$. Putting these two results together we obtain

$$4\delta(1+\epsilon)W_i + Iwt(u,v) > \frac{1-2\delta(1+\epsilon)}{(1+\epsilon)(1+6\delta)} \cdot \sqrt{tt'} \cdot Iwt(u,v).$$

Using the fact that $Iwt(u,v) > W_i$ we get:

$$4\delta(1+\epsilon) > \frac{1-2\delta(1+\epsilon)}{(1+\epsilon)(1+6\delta)} \cdot \sqrt{tt'} - 1.$$

If we solve this inequality for $\delta$, we see that the only positive solutions are

$$\delta > \frac{\sqrt{tt'} - (1+\epsilon)}{2(\sqrt{tt'}(1+\epsilon) + 5 + 7\epsilon + 2\epsilon^2)}.$$

According to our choice of $\delta$ in algorithm Improve-Greedy, the above inequality will never be satisfied (see Fig. 2). Hence, we have a contradiction. $\square$

The following theorem now concludes this section.

**Theorem 4.** *The graph produced by* Improved-Greedy *is a $t$-spanner of the complete Euclidean graph.*

*Proof.* Follows from the fact that $G'$ is a $\sqrt{tt'}$ spanner of $G$ and that $G$ is a $\sqrt{t/t'}$ spanner of the complete graph. $\square$

## 5 The weight of $G'$ is $O(wt(MST))$

In [4] it was shown that the greedy algorithm produces a spanner that has $O(n)$ edges and a total weight of $O(\log n) \cdot wt(MST(V))$, where $MST(V)$ is the minimum spanning tree of $V$. The analysis of the greedy algorithm was then improved in [7]. The proof relies on a property known as the *leapfrog property.* This property restricts how a set of line segments may be positioned in space. Here we provide a definition, which is technical and non-intuitive.

Let $t \geq t' > 1$. A set of line segments, denoted $E'$, in $d$-dimensional space satisfy the $(t', t)$-*leapfrog* property if the following is true for every possible subset $S = \{(u_1, v_1), \ldots, (u_m, v_m)\}$ of $E'$:

$$t' \cdot wt(u_1, v_1) < \sum_{i=2}^{m} wt(u_i, v_i) + t \cdot (\sum_{i=1}^{m-1} wt(v_i, u_{i+1}) + wt(v_m, u_1)).$$

Informally, this definition says that if there exists an edge between $u_1$ and $v_1$ then any path, not including $(u_1, v_1)$ must have length greater than $t' \cdot wt(u_1, v_1)$.
The following fact was shown by Das and Narasimhan [8].

**Fact 5** *(Theorem 3 in [8])*
*There exists a constant $0 < \phi < 1$ such that the following holds: if a set of line segments $E'$ in $d$-dimensional space satisfies the $(t', t)$-leapfrog property where $t \geq t' \geq \phi t + 1 - \phi > 1$, then $wt(E') = O(wt(MST))$, where $MST$ is a minimum spanning tree connecting the endpoints of $E'$. The constant implicit in the $O$-notation depends on $t$ and $d$.*

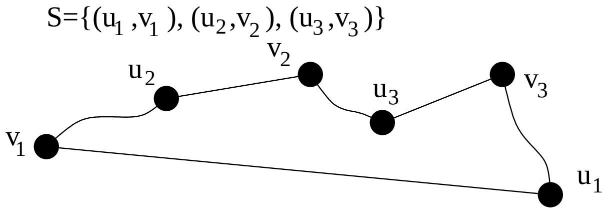


**Fig. 7.** Illustration of the definition of the leapfrog property.

Now, suppose that we construct a $t$-spanner such that for every spanner edge $(u, v)$, the second shortest path is not necessary longer than $t{\cdot}wt(u, v)$, but longer than $t'{\cdot}wt(u, v)$, for some $t'$ such that $t \geq t' > 1$. In this case the $t$-spanner satisfies the $(t, t')$-leapfrog property, as can be proved by using arguments similar to those used in Lemma 2.4 in [7]. Hence, the produced spanner will then have total weight $O(wt(MST(V)))$.

So, it remains to prove that the weight of the second shortest path between $u$ and $v$ is greater than $t'{\cdot}wt(u, v)$. First note that the edges in $E_0$ do not contribute much, because their total length is at most equal to the length of the longest edge ($<n{\cdot}D/n$), which is less than the weight of the minimum spanning tree. We estimate $wt(E' \backslash E_0)$, where $E'$ is the set of edges produced by the algorithm.

**Lemma 8.** *Let $e{=}(u, v) \in E' \backslash E_0$. The weight of the second shortest path between $u$ and $v$ is greater than $t'{\cdot}wt(u, v)$.*

*Proof.* Let $C$ be the shortest simple cycle in $G'$ containing $e$. We have to estimate $wt(C) - wt(u, v)$. Let $e_1 = (u_1, v_1)$ be the longest edge on the cycle. Then $e_1{\in}E' \backslash E_0$, and among the cycle edges it is examined last by the algorithm. What happens while the algorithm is examining $e_1$?

Assume that $e_1$ is examined in phase $i$. There is an alternate path in $G'$ from $u_1$ to $v_1$ of weight $wt(C){-}wt(u_1, v_1)$. But since the algorithm eventually decides to add $e_1$ to the spanner, at that moment the weight of each cluster path from $u_1$ to $v_1$ is larger than $\sqrt{tt'}{\cdot}Iwt(u_1, v_1){\cdot}U_i$. Notice that $Iwt(u_1, v_1) \cdot U_i$ and $wt(u_1, v_1)$ are larger than $W$. This implies that $u_1$ and $v_1$ are not contained in the same cluster. Thus $u_1$ is sufficiently far from $v_1$. Lemma 6 implies that the weight of each path in $G'$ between $u_1$ and $v_1$ is large, i.e.,

$$wt(C) - wt(u_1, v_1) > \sqrt{tt'} \cdot Iwt(u_1, v_1) \cdot U_i > \sqrt{tt'} \cdot wt(u_1, v_1) \cdot \frac{1 - 2\delta(1 + \epsilon)}{(1 + \epsilon)(1 + 6\delta)}.$$

But we know, according to the algorithm, that $\delta \leq \frac{\sqrt{tt'} - (1+\epsilon)t'}{2(1+\epsilon)(\sqrt{tt'} + 3t')}$. Substituting, we obtain $wt(C){-}wt(u_1, v_1) > t'{\cdot}wt(u_1, v_1)$. □

Finally, since Lemma 8 holds, we can use the following observation which together with Fact 5 concludes the proof of Theorem 1.

**Observation 6** *$E'/E_0$ satisfies the $(t', t)$-leapfrog property.*

*Proof.* Consider any subset of the edges, $S = \{(u_1, v_1), \ldots, (u_m, v_m)\}$ of $E'$. By Lemma 8 we know that $t' \cdot d(u_1, v_1)$ is smaller than the weight of the second shortest path between $u_1$ and $v_1$ in $G'$. Consider a path $P$ from $v_1$ to $u_1$, composed of the shortest path from $v_1$ to

$u_2$ (of weight $\leq t \cdot d(v_1, u_2)$), the edge $(u_2, v_2)$, the shortest path from $v_2$ to $u_3$ (of weight $\leq t \cdot d(v_2, u_3)$), and so on, until the final portion is the shortest path from $v_m$ to $u_1$. Clearly $wt(P)$ is at least as large as the weight of the second shortest path between $u_1$ and $v_1$. But $wt(P)$ is also equal to the right hand side of the definition of the leapfrog property. The fact follows. □

This concludes the proof of Theorem 1.

## 6 Conclusions and open problems

This paper represents an important advancement in the study of spanners; we present the first correct $O(n \log n)$-time algorithm to construct low weight spanners (weight $O(1) \cdot wt(MST)$) and small number of edges (only $O(n)$ edges). The implementation of clustering techniques are of independent interest in the design of efficient algorithms.

Our algorithms are also interesting since they employ the practical *real RAM* model of computation. The main theoretical open problem that remains unsolved is to design an algorithm to construct a sparse $t$-spanner in time $O(n \log n)$ in the algebraic decision tree model of computation.


## 7 Acknowledgments

We are grateful to Professor Michiel Smid for helpful discussions and for pointing out errors in an earlier draft.


## References


1. I. Althöfer, G. Das, D. P. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete Comput. Geom.*, 9:81–100, 1993.
2. S. Arya, G. Das, D. M. Mount, J. S. Salowe, and M. Smid. Euclidean spanners: short, thin, and lanky. In *Proc. 27th Annu. ACM Sympos. Theory Comput.*, pages 489–498, 1995.
3. P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to $k$-nearest-neighbors and $n$-body potential fields. *J. ACM*, 42:67–90, 1995.
4. B. Chandra, G. Das, G. Narasimhan, and J. Soares. New sparseness results on graph spanners. *Internat. J. Comput. Geom. Appl.*, 5:125–144, 1995.
5. D. Z. Chen, G. Das, and M. Smid. Lower bounds for computing geometric spanners and approximate shortest paths. In *Proc. 8th Canad. Conf. Comput. Geom.*, pages 155–160, 1996.
6. A. Czumaj and A. Lingas. Linear-time heuristics for minimum weight rectangulation. In *Proc. ICALP*, LNCS, page ?, 2000.
7. G. Das, P. Heffernan, and G. Narasimhan. Optimally sparse spanners in 3-dimensional euclidean space. In *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. *Internat. J. Comput. Geom. Appl.*, 7:297–315, 1997.
9. J. M. Keil and C. A. Gutwin. Classes of graphs which approximate the complete Euclidean graph. *Discrete Comput. Geom.*, 7:13–28, 1992.
10. C. Levcopoulos and A. Lingas. There are planar graphs almost as good as the complete graphs and almost as cheap as minimum spanning trees. *Algorithmica*, 8:251–256, 1992.
11. C. Levcopoulos, G. Narasimhan, and M. Smid. Efficient algorithms for constructing fault-tolerant geometric spanners. In *Proc. 30th Annu. ACM Sympos. Theory Comput.*, pages 186–195, 1998.
12. D. Mount. Personal communication, 1998.
13. S. B. Rao and W. D. Smith. Improved approximation schemes for traveling salesman tours. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, 1998.

14. J. S. Salowe. Construction of multidimensional spanner graphs with applications to minimum spanning trees. In *ACM Symposium on Computational Geometry*, pages 256–261, 1991.
15. M. Thorup. Undirected single-source shortest path with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999.
16. P. M. Vaidya. A sparse graph almost as good as the complete graph on points in $K$ dimensions. *Discrete Comput. Geom.*, 6:369–381, 1991.