Improved 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@cs.lth.se, 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 a $O(n \log n)$-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 have $O(n)$ edges, constant degree and weight $O(wt(MST))$.

1 Introduction

Spanners have applications in the design of geometric networks. Consider a set V of n points in R^d, where the dimension d is a constant. A geometric 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. For $u,v \in V$, let P be a uv-path in G, i.e., a path in G between u and v. The weight of the path P is denoted by $wt(P)$ and is defined as the sum of the weights of the edges of P. 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 uv-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 $O(wt(MST))$. Given a geometric network $G = (V,E)$, a weight function w defined on its edges, and two vertices $u,v \in V$, we let $D_{\{G,w\}}(u,v)$ denote the weight of the shortest path from u to v in G for the weight function w.

The problem of constructing spanners has been investigated by many researchers. Keil and Gutwin [5] showed that for any $t > 1$, and any set V of n points in the plane, a t-spanner for V having $O(n)$ edges can be constructed in $O(n \log n)$ time. Salowe [9], Vaidya [11] and Callahan and Kosaraju [2] showed the same result for any fixed dimension d. Das and Narasimhan [4] gave an $O(n \log^2 n)$-time algorithm that constructs for any set V of n points in R^d, and any constant $t > 1$, a t-spanner for V in which the degree of every point

* Funded by NSF (CCR-940-9752) and Cadence Design Systems, Inc.

is bounded by a constant, and whose total edge weight is proportional to the weight of a minimum spanning tree of V. Chen et al. [3] 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 [6] has shown that a significant result claimed in Arya et al. [1] 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 [8] 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 $O(n \log n)$. Since the claim by Arya et al. [1] 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 [4] 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 [7].

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, constant degree and weight $O(wt(MST))$. The constants in the O-notation depend on t and d.*

It was shown in [4] that the greedy algorithm produces spanners with $O(n)$ edges and weight $O(1) \cdot wt(MST)$. However, a naive implementation of the greedy algorithm had a running time of $O(n^3 \log n)$, mainly due to the fact that $\Omega(n)$ shortest path queries needed to be answered in a "dynamic" graph with at most $O(n)$ edges, each of which could take $O(n \log n)$ time.

Our algorithm is inspired by the algorithm due to Das and Narasimhan [4]. They showed how to use "clustering" in order to speed up shortest path queries. 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 [4] 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 (which takes $O(n \log n)$ time), given a linear-sized edge-weighted graph with integral edge weights in the range $[0..N]$, and given a set of "cluster-centers", then we can perform "clustering" very efficiently in only $O(n + N)$ time. In 1999, Thorup [10] showed that single source shortest path queries could be answered in linear time for undirected graphs with integer weights. The main reasons why this algorithm is not used in this paper is that 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.

2 The DN-Clustering Spanner Algorithm

We first describe the cluster-based spanner algorithm by Das and Narasimhan [4]. It can be roughly described as follows. The algorithm starts with an empty spanner G'. A preprocessing step helps to eliminate all but a linear number of edges. 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 e is added to the graph G', else it is discarded. Whenever shortest path queries are required to be answered, these are not solved on the graph G' being constructed. Instead, they are solved on a "cluster-graph" H, which is simultaneously maintained. The cluster graph H from [4] 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 $O(1)$ time.

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 whose lengths are within a constant factor of $wt(e)$. For all practical purposes, 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. 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 $O(1)$ time, all 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 (using Dijkstra's 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 [4] 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 small length edges, and that clustering can help eliminate all small length 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 make the observation 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 "clustering" is recomputed after processing every batch of edges, since each call to the clustering algorithm takes $O(n \log n)$ time, the total time for cluster graph computations will be $O(\frac{n \log^2 n}{\log \log n})$. 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 [4], in phase i, 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 edges ("inter-cluster" edges) whose weights were in the range $(\delta W, 2W(1 + 2\delta)]$. This meant that for edge $e = (u, v)$ 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 [4] that if there does exist 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 $\frac{2t}{\delta}$ (since $l \leq 2W$). 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 (running Dijkstra's algorithm starting from vertex u suffices). A tighter analysis was unnecessary in the DN-Clustering algorithm of [4] since c, t, and δ 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 $(W, W 2^{\frac{\log \log n}{4 \cdot d}}]$, i.e., they are in the range $(W, W(\log n)^{\frac{1}{4 \cdot d}}]$. This implies that, for edge $e = (u, v)$ 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 could have edges ("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 δW. 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 $O((\frac{t \cdot (\log n)^{\frac{1}{4 \cdot d}}}{\delta^2})^d)$. Thus the maximum number of vertices (and edges, due to the constant degree) that can be reached when performing Dijkstra's algorithm starting from vertex u is $O((\frac{t \cdot (\log n)^{\frac{1}{4 \cdot d}}}{\delta^2})^d)$. Since t, d and δ are constants, $O((\frac{t \cdot (\log n)^{\frac{1}{4 \cdot d}}}{\delta^2})^d) = O((\log n)^{\frac{1}{4}})$. 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 t-spanner of the complete Euclidean graph can be constructed in $O(\frac{n \log^2 n}{\log \log n})$ time such that the spanner has $O(n)$ edges, constant degree and weight $O(1) \cdot wt(MST)$. The constants implicit in the O-notation depend on t and d.*

3 An Improved Spanner Algorithm

In the rest of the paper, we describe an efficient algorithm to construct sparse spanners with a running time of $O(n \log n)$.The running time of $O(n \log n)$ for our algorithm is achieved by designing an $O(n)$-time algorithm for the clustering step, thus executing all the clustering steps in $O(n \log n)$ time. Note that the running time is $O(n \log n)$ even if clustering is executed $O(\log n)$ times.

One crucial idea that we employ to speed up the clustering is to replace the real-valued weights by integral values. As observed in [4], 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 compute the $O(n)$ floor/ceiling functions needed by using operations allowed under the RAM model. The second crucial component of our algorithm is an implementation of the clustering algorithm in $O(n)$ time assuming small integral weights for the edges. We also prove that the integralization introduces only a bounded amount of error, and that this error retains the correctness of the other required operations.

The improved spanner algorithm can be roughly described as follows. It is important to note that the skeleton of the algorithm is similar to the DN-clustering algorithm from [4]. In particular, this improved algorithm also runs in $O(\log n)$

phases. The algorithm starts with an empty graph G' and employs the same preprocessing step to eliminate all but a linear number of edges. This step is done by a call to the t-spanner algorithm presented by Arya et. al. in [1], with $\sqrt{t/t'}$ as input parameter. Note that this algorithm in [1] is correct and runs in time $O(n \log n)$. It also guarantees that the graph has constant degree. As before, 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 now 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 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. 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 is maintained by the algorithm as described later. Whenever the phase number is clear by 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 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 differences of this from the one in [4] 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 a shortest path query, i.e., checking whether $D_{\{G',wt\}}(u, v) \leq t \cdot wt(e)$. As in [4], this query is answered in $O(1)$ time per query by performing an approximate shortest path query on the simpler graph H. If the answer to the query is yes, then 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.

Since the edges that remain to be 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 $(2^{i-1} \cdot D/n, 2^i \cdot D/n]$. At the start of each of the $\log n$ phases, the algorithm calls the "clustering" algorithm, which is required to answer the shortest path queries efficiently. The clustering algorithm is described in section 3.2. Later we show that the running time of our algorithm is $O(n \log n)$. Note that processing is done in $\log n$ phases.

If fewer number of phases are used as in the faster spanner algorithm described in section 2.1, then the error due to integralization could be too large. Even if 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 detailed algorithm is given below in Fig. 1.

Algorithm $\textsc{Improved-Greedy}(V, t, t')$

Compute a $(\sqrt{t/t'})$-spanner $G = (V, E)$ using the algorithm from [1]

$\delta := \min\left(\frac{\sqrt{tt'}-t'}{4(\sqrt{tt'}+3t')}, \frac{\sqrt{tt'+34\sqrt{tt'}+1}-(\sqrt{tt'}+5)}{24}\right)$

sort E; $D :=$ weight of largest edge in E;

$W_0 := 0$; $W_i := 2^{(i-1)}D/n$ for $i = 1, 2, \ldots, \log n$

$I_i := (W_i, W_{i+1}]$ for $i = 0, 1, \ldots, (\log n - 1)$

$E_i :=$ (sorted) edges of E with weights in I_i; $E' := E_0$; $G' := (V, E')$;

$\textsc{Integralize}(E_0, 0)$

$C_1 := \textsc{Naive-Centers}(G', \delta \cdot W_1)$; $M_1 := \emptyset$

for $i := 1$ to $\log n$ **do**

    $\textsc{Integralize}(E_i, i)$

    $\textsc{ReIntegralize}(E_0 \cup E_1 \cup \ldots \cup E_{i-1})$

    $H := \textsc{Cluster-Graph}(G', Iwt, C_i, r, R)$

    for each edge $e = (u, v) \in E_i$ **do**

        if not $\textsc{Short-Path}(H, u, v, \sqrt{tt'} \cdot d(u, v))$ **then**

            $E' := E' \cup \{e\}$; $G' := (V, E')$

    $C_{i+1} := \textsc{Update-Centers}(H, i, C_i, r)$

output G'

Fig. 1. 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 queries 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 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 $2t \cdot 2W_i$ 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: $Iwt_i(e) := \lceil \frac{wt(e)}{U_i} \rceil$.

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 is also easy to see 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 at most nU_i. Another consequence is that given two simple paths P_1 and P_2, if $Iwt(P_1) = Iwt(P_2)$, then $|wt(P_1) - wt(P_2)| \leq nU_i$. 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 given vertices u and v in G' such that $D_{\{G',wt\}}(u,v) \geq W_i$, $D_{\{G',wt\}}(u,v) \leq D_{\{G',Iwt\}}(u,v) \cdot U_i < (1+\epsilon) \cdot D_{\{G',wt\}}(u,v)$.*

Proof. We give a sketch of the proof. 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 (i.e., the ratio of the error to the weight of the path) 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 ϵ can be made as small as desired by choosing an appropriate value of c. $\square$

Corollary 1. *For a path P in G' with $wt(P) \geq \delta W_i$ (i.e., $Iwt(P) \geq R$), 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 $O(n \log n)$ 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, since its integer value is in the range $[0..N]$ (where $N = c \cdot n$), it 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) performs this operation for each edge in the set E 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 a constant, the integralization would turn out to be too expensive. Our 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 $O(1)$ 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 quite easily be accomplished by using $O(n \log n)$ preprocessing. One way to accomplish this would be to build a balanced binary tree including $c \cdot n$ elements with the values $1 \dots c \cdot n$. Every element in the tree also contains a pointer to the element in the tree containing the value $\lceil \frac{val}{2} \rceil$. This value can be computed in time $O(\log n)$ and searching the tree for the value is also done in $O(\log n)$. Hence, by using $O(n \log n)$ time preprocessing, the integral weight of an edge for the next phase can be computed in constant time. 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(F) performs its operation for each edge in the edge set F 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

First for some definitions. Here we assume that $G = (V, E)$ is an arbitrary weighted graph, with weight function w defined on the edges in E. The following definition of a cluster is modified from the one in [4] 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 Euclidean graph $G = (V, E)$, a vertex $v \in V$, a radius r, and a weight function w defined on the edges in E, Cluster(G, v, r, w) *is defined as the set of all vertices u such that $D_{G,w}(v, u) \leq r$. 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 Euclidean graph $G = (V, E)$, a set $C = \{v_1, v_2, \dots, v_m\} \subseteq V$, a radius r, and a weight function w defined on the edges in E, the Cluster-Cover(G, C, r, w) *is defined as a set of clusters $\mathcal{K} = \{K_1, K_2, \dots, K_m\}$ such that K_i is a cluster with radius r and cluster-center at v_i $(i = 1, 2, \dots, m)$, and such that $K_1 \cup K_2 \cup \dots \cup K_m = V$.*

During the course of the algorithm, clustering is performed on the spanner graph G' with the weight function Iwt. Also, the set C and the value 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 of a cluster-graph so that it is a bit more general and it is defined for a given set of clusters and for an arbitrary weight function.

Definition 3. Cluster graph
Assume that $G = (V, E)$ is a Euclidean graph with a weight function w defined on its edges. Assume that $C = \{v_1, v_2, \ldots, v_m\} \subseteq V$ is a given set of cluster-centers. For a given radius r, we assume that $\mathcal{K} = \{K_1, K_2, \ldots, K_m\}$ is equal to CLUSTER-COVER(G, C, r, w). *Given $R \geq r$, the* 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 $D_{\{G,w\}}(u, v)$ 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 $u_i \in K_i, u_j \in K_j$ (Type 2).*

Computing the Cluster graph Here we describe how the cluster graph is computed efficiently under some assumptions. We first describe how a cluster-cover is computed. Once a cluster-cover is computed we show that the cluster graph can be easily computed. 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 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 $[0 \mathinner{..} R]$. 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 [4], i.e., to run Dijkstra's SSSP 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 faster priority queue, which we denote by PQ. The priority queue we use is an array of size R, indexed from 0 to R. 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, 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 is an array, EXTRACT-MIN can be implemented as a scan through the array for the "next" largest item. This means that the $O(n)$ calls to EXTRACT-MIN needed by Dijkstra's algorithm can be implemented in $O(n + R)$ time.

One problem is that clusters can overlap and that vertices may have entries in the priority queue with distances from several cluster-centers. This can be taken care of by augmenting the priority queue entries to also store information about the vertex as well as the corresponding cluster-center. It should also be noted that Dijkstra's algorithm 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. 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 $O(1)$ time. Also note that an edge (u, v) may be RELAXed several times, each time with respect to a different cluster-center. Thus the time and space complexity of the algorithm is affected by the amount of overlap of the clusters in the cluster-cover. A careful implementation of cluster-cover can be made to run in time $O(m \cdot c_v \cdot c_e + R)$, where m is the number of cluster-centers, c_v is the maximum number of clusters that contain a vertex, and c_e is the maximum number of clusters that contain one of the endpoints of an edge. (In Section 3.3. we show that for our purposes c_v and c_e are constants.)

We now 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. 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 $\mathcal{K}_r$ and $\mathcal{K}_R$ respectively. We augment the cluster-cover procedure to also produce a data structure that supports the following queries for both the cluster-covers: (a) FINDCENTERS$(v, \mathcal{K})$: Given $v \in V$, it returns all cluster-centers v_i such that v is in a cluster from $\mathcal{K}$ centered at v_i, i.e., $D_{G,Iwt}(v, v_i)$ is at most the radius of the clusters in $\mathcal{K}$. it also returns $D_{G,w}(v, v_i)$ for these cluster-centers, and (b) COMPUTEDISTANCE(v_i, v): Given $v \in V$, and a cluster-center v_i, it returns $D_{G,Iwt}(v, v_i)$ if $D_{G,Iwt}(v, v_i) \le R$; otherwise, it returns the value ∞.

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 $\mathcal{K}_r$ and adding the corresponding edges. The inter-cluster edges can be of two types. An edge $[v_i, v_j]$ of type 1 is added if $v_i \notin K_j$ and $v_j \notin K_i$ and $D_{\{G,Iwt\}}(v_i, v_j) \le R$. Note that K_i and K_j are clusters of radii r with centers at v_i and v_j respectively. For every cluster-center v_i, we use the FINDCENTERS query to list all the clusters from $\mathcal{K}_R$ that it is contained in. The centers v_j of these clusters satisfy the condition that $D_{\{G,w\}}(v_i, v_j) \le R$. Now we use the COMPUTEDISTANCE queries to make sure that $v_i \notin K_j$ and $v_j \notin K_i$. A careful consideration of all the steps above shows that the time complexity of computing the cluster graph is $O(n \cdot c_v^2)$. Having the cluster centers 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 An edge 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, 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 $e = (u, v)$, 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 $\mathcal{K}_r$ and join the corresponding cluster-centers by an inter-cluster edge 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 $w(u_i, u_j)$. The above function runs in $O(1)$ time.

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 $G' = (V, E_0)$ with real-valued edge weights, and using a radius of r. This is referred to as NAIVE-CENTERS in the algorithm given in Fig. 1. 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 $r/2$.

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 deletes all cluster-centers that are within distance r from it. (It is important to note that since the integralization changes in every iteration, vertices that are distance r' in one iteration are at distance $r'/2$ in the next iteration.) This is easily implemented by using the FINDCENTERS query that is available after the cluster-cover for phase i has been computed. The next cluster-center is then picked and the process is continued until all centers are either picked or marked. Clearly this process runs in time $O(m \cdot c_v)$. 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 $r/2$ 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. Lemma 2 specifies conditions under which vertices belong to at most a constant number of clusters.

Lemma 2. *If* $C{=}\{v_1, v_2, \ldots, v_m\} \subseteq V(G)$ *vertices* $v_i, v_j{\in}C$, $D_{\{G,Iwt\}}(v_i, v_j) > r'$, *and if* $\mathcal{K}_{r''} = \{K_1, K_2, \ldots, K_m\}$, *is returned by* CLUSTER-COVER(G, C, r'') *and if* $r'' \leq c' \cdot r'$ *for some constant* c', *then each vertex* $v \in V(G)$ *is contained in at most a constant (which depends on* d *and* c'*) number of clusters from* $\mathcal{K}_{r'}$.

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 $\mathcal{K}_r$ or $\mathcal{K}_R$. The proof follows from standard packing arguments. Similar arguments also show that the number of inter-cluster edges incident on 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 $O(n)$. Note that since the weights have been integralized, the resulting clusters are approximate clusters; they are a little bit larger (since integers are always rounded up) than the exact clusters.

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 earlier, 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 a constant if $D_{\{H,Iwt\}}(u, v)$ is bounded by some constant multiple of r is a constant. Hence, we conclude this section by noting that the Improved-Greedy algorithm runs in time $O(n \log n)$.

3.4 The graph produced by IMPROVED-GREEDY is a t-spanner

In order to show that a valid cluster graph H approximates the graph G', we need to prove lemmas that are analogous to Lemmas 3 and 4 from [4] modified to account for the error introduced by the integralization. Next, we need to show that the G' is a t-spanner for V. Since the clusters are computed using the function $Iwt(\cdot)$ instead of $wt(\cdot)$, clusters are not as precise as they were in [4]. The following claims are stated without proof, which will be provided in a full version of the paper. Consider the cluster graph H that results from the clustering performed on G' at the start of phase i. The following claims apply to edges and paths in H. Many of them are modified versions of corresponding lemmas in [4]:

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

2. If u is a cluster-center and $[u,v]$ is an intra-cluster edge in the cluster-graph H, then $D_{\{G',wt\}}(u,v) \leq (1+\frac{\epsilon}{\delta})rU_i$.
3. If $[u,v]$ is an inter-cluster edge, then $r_i < D_{\{G',wt\}}(u,v) \leq (1+\epsilon)\cdot(R_i+2r_i)U_i$.
4. 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 $P_{G'}$ between u and v such that $Iwt(P_{G'}) \leq L$.
5. Let H be a valid Cluster graph of G' with cluster radii r and $R = r/\delta$. Let $P_{G'}$ be a path between u and v in G' of weight $Iwt(P_{G'})$ such that $D_{\{G',wt\}}(u,v) > (1+\epsilon)W - 2\delta W$. Then there exists a path P_H between u and v in H such that $Iwt(P_H) \leq \left(\frac{1+6\delta}{1-2\delta}\right) \cdot Iwt(P_{G'})$.

The above claims are enough to prove that H is a cluster graph for G', and consequently that G' is a t-spanner. As argued in Section 3.2, the resulting spanner graph has constant degree. The weight of the spanner is $O(1)\cdot wt(MST)$ because of the *leapfrog property* from [4], the proof is omitted in this version.

This concludes the proof of Theorem 1.

4 Conclusions and Acknowledgments

We present improved algorithms for the sparse spanner problem. In the process, we design linear-time algorithms for a clustering problem, which is likely to be of independent interest.

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

References

1. S. Arya, G. Das, D. M. Mount, J. S. Salowe, and M. Smid. Euclidean spanners: short, thin, and lanky. In *Proc. ACM STOC'95*, pages 489–498, 1995.
2. P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM*, 42:67–90, 1995.
3. D. Z. Chen, G. Das, and M. Smid. Lower bounds for computing geometric spanners and approximate shortest paths. In *Proc. CCCG'96*, pages 155–160, 1996.
4. G. Das and G. Narasimhan. A fast algorithm for constructing sparse Euclidean spanners. *Internat. J. Comput. Geom. Appl.*, 7(4):297–315, 1997.
5. J. M. Keil and C. A. Gutwin. Classes of graphs which approximate the complete Euclidean graph. *Discrete Comput. Geom.*, 7:13–28, 1992.
6. D. Mount. Personal communication, 1998.
7. F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1985.
8. S. B. Rao and W. D. Smith. Improved approximation schemes for traveling salesman tours. In *Proc. ACM STOC'98*, 1998.
9. J. S. Salowe. Construction of multidimensional spanner graphs with applications to minimum spanning trees. In *Proc. ACM SoCG'91*, pages 256–261, 1991.
10. M. Thorup. Undirected single-source shortest path with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999.
11. 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.