

FAST GREEDY ALGORITHMS FOR CONSTRUCTING SPARSE GEOMETRIC SPANNERS*

JOACHIM GUDMUNDSSON[†], CHRISTOS LEVCOPOULOS[‡], AND GIRI NARASIMHAN[§]

Abstract. Given a set V of n points in \mathbb{R}^d and a real constant $t > 1$, we present the first $O(n \log n)$ -time algorithm to compute a geometric t -spanner on V . A geometric t -spanner on V is a connected graph $G = (V, E)$ with edge weights equal to the Euclidean distances between the endpoints, and with the property that, for all $u, v \in V$, the distance between u and v in G is at most t times the Euclidean distance between u and v . The spanner output by the algorithm has $O(n)$ edges and weight $O(1) \cdot wt(MST)$, and its degree is bounded by a constant.

Key words. computational geometry, sparse geometric spanners, cluster graph

AMS subject classifications. 68U05, 65D18

PII. S0097539700382947

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, the 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 (number of edges) $O(n)$ and weight (sum of edge weights) $O(1) \cdot wt(MST)$, where $wt(MST)$ is the total weight of a minimal spanning tree. Given a geometric network $G = (V, E)$, a (generic) weight function wt 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. Levkopoulos and Lingas [10] presented an $O(n \log n)$ -time algorithm that produced a sparse t -spanner for the two-dimensional case. It works by taking any t -spanner which has the form of a (possibly partial) triangulation and achieving almost the same t as that triangulation. However, the problem gets much more difficult in higher

*Received by the editors December 20, 2000; accepted for publication (in revised form) February 18, 2002; published electronically August 1, 2002.

<http://www.siam.org/journals/sicomp/31-5/38294.html>

[†]Department of Computer Science, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands (Joachim@cs.uu.nl). This author's research was funded by the Swedish Foundation for International Cooperation in Research and Higher Education.

[‡]Department of Computer Science, Lund University, Box 118, 221 00 Lund, Sweden (Christos@cs.lth.se).

[§]School of Computer Science, Florida International University, Miami, FL 33199 (giri@cs.fiu.edu). Part of this author's work was done at the University of Memphis and at Lund University.

dimensions. There are several algorithms that run in time $O(n \log n)$ [3, 9, 15, 17]. 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 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, Das, and Smid [5] showed that the lower bound for computing any t -spanner for a given set of points V in \mathbb{R}^d is $\Omega(n \log n)$ in the algebraic computation tree model.

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

Before a correct $O(n \log n)$ -time algorithm was presented, sparse spanners were used in designing efficient approximation schemes for geometric problems. Rao and Smith [14] made a breakthrough by showing 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 multiconnectivity problems in geometric graphs that also depended on the assumption that sparse spanners could be computed in $O(n \log n)$ time. Thus the existence of an $O(n \log n)$ -time algorithm to construct sparse spanners became 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 show the following theorem.

THEOREM 1. *Given a set V of n points in d -dimensional space and any real constant $t > 1$, in the algebraic decision tree model of computation, a sparse t -spanner of the complete Euclidean graph can be constructed in $O(\frac{n \log^2 n}{\log \log n})$ time. If the model is extended with indirect addressing, a sparse t -spanner of the complete Euclidean graph can be constructed in $O(n \log n)$ time. 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 Figure 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 main-

Algorithm STANDARD-GREEDY(G, t)

1. sort the edges in E by increasing weight
 2. $E' := \emptyset$
 3. $G' := (V, E')$
 4. **for** each edge $(u, v) \in E$ **do**
 5. **if** SHORTESTPATH(G', u, v) $> t \cdot d(u, v)$ **then**
 6. $E' := E' \cup \{(u, v)\}$
 7. $G' := (V, E')$
 8. output G'
-

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

tain 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, one can perform clustering very efficiently in only $O(n + N)$ time.

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

2. An improved spanner algorithm. We first describe the previous cluster-based spanner algorithm due to Das and Narasimhan [8]. Then, in section 2.2, we present a simple modification to the DN-clustering algorithm to construct sparse t -spanners.

2.1. The DN-clustering spanner algorithm. The algorithm by Das and Narasimhan [8] can roughly be described as follows.

The algorithm starts with an empty spanner G' . A preprocessing step helps to eliminate all but a linear number of edges from further consideration. For a given value of t , this step is performed by a call to an $O(n \log n)$ -time algorithm presented by Salowe [15] or Arya et al. [2] to compute a spanner with stretch factor $\sqrt{t/t'}$ (for some $t > t' > 1$). 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 $D_{\{G', wt\}}(u, v) \leq \sqrt{tt'} \cdot wt(e)$. If the answer to the query is no, then edge e is added to the spanner G' , or 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. A set of points $C \subseteq V$ is a cluster of radius r with cluster center $v \in C$ if, for every point $u \in C$, there is a path in G' between v and u of length at most r . A set of clusters C_1, \dots, C_k is a cluster cover of G' if every point in V belongs to at least one cluster. A cluster graph H can be constructed from a cluster cover by adding two types of edges: intracluster edges (edges connecting the cluster center of a cluster C to all other vertices in C) and intercluster edges (edges connecting two cluster centers). The cluster graph H from [8] has the following properties:

1. distances in H “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 was demonstrated by showing that corresponding distances in H and G' differ by only a small 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 has only edges between clusters (the so-called intercluster 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 unweighted graph with bounded degree, checking whether the

distance between two vertices u and v is at most a constant $\sqrt{tt'}$ can be achieved in $O(1)$ time (since there are only $O(1)$ vertices at the 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 intercluster 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 starting from sequentially selected cluster centers). 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.

2.2. 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 the two costs, it is necessary to do fewer than $O(\log n)$ cluster graph computations. This in turn would make the shortest path queries more expensive because it increases the ratio between the lengths of the longest and shortest edges in the cluster graph, which implies that the number of edges along the shortest path between two cluster centers will also increase, and therefore the query time will also increase. 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 intercluster 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 a 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 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 $(W, W(\log n)^{\frac{1}{4 \cdot d}}]$. Thus, 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 $\sqrt{tt'} \cdot l$. During batch i , the cluster graph H can have intercluster edges with weights 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 $\frac{\sqrt{tt'}(\log n)^{\frac{1}{4 \cdot d}}}{\delta}$. 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 $\sqrt{tt'} \cdot l = \sqrt{tt'}W(\log n)^{\frac{1}{4 \cdot d}}$. Packing arguments [8] show that, in \mathbb{R}^d , the number of balls of radius r that can be packed in a ball of radius R is bounded by $O((R/r)^d)$. Thus the number of balls of radius $r = \frac{\delta W}{2}$ that can be packed in a ball of radius $R = \sqrt{tt'}W(\log n)^{\frac{1}{4 \cdot d}}$ is at most $O((\frac{t \cdot (\log n)^{\frac{1}{4 \cdot d}}}{\delta})^d)$. 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 $O((\frac{t \cdot (\log n)^{\frac{1}{4 \cdot d}}}{\delta})^d) = O((\log n)^{\frac{1}{4}})$ (since t , d , and δ are constants). We conclude that Dijkstra’s algorithm for a shortest path query has a time complexity of $O((\log n)^{\frac{1}{4}} \cdot (\log((\log n)^{1/4}))) = O(\log n)$. Thus all $O(n)$ shortest path queries can be answered in $O(n \log n)$ time. Note that, even though the clustering and shortest path costs are not precisely balanced, it is possible to prove (using lengthy but straightforward algebraic calculations) that (asymptotically) it cannot be improved. The obtained spanner satisfies the leapfrog property [8] (also defined in section 5), which implies that the weight of the spanner is $O(wt(MST))$.

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 that uses indirect addressing. 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 2.1, the reason 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 cor-

rect 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 algebraic computation tree model extended with indirect addressing. 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 third and final crucial component in our algorithm is that we show how it is possible to select the cluster centers for each stage of the algorithm in linear time.

The improved spanner algorithm can be roughly described as follows; see Figure 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 (pre-processing) 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. [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 step, 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 $O(wt(MST))$. The greedy algorithm is then simulated on the remaining edges of the initial spanner.

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' . Clustering is made more precise in section 3.2. The difference between this and the one in [8] lies in the fact that the cluster centers have to be se-

lected 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 $D_{\{G', wt\}}(u, v) \leq t \cdot wt(e)$. We answer an approximate version of this query, i.e., performing a shortest path query on the simpler graph H and not on the partial spanner graph G' . If the answer to the approximate query is “no,” edge e is added to the graph G' ; otherwise, 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 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 10, 11, 14, and 22) and the computation of the cluster centers (steps 12 and 21). 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 truly time-critical step of this algorithm is the clustering step (step 15) and selecting the new cluster centers for the next phase (step 21). Both of these steps will be closely described in section 3.2. Note that the two values bounding δ are decided in Lemmas 14 and 17.

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 $i = 1, 2, \dots, \log n$
7. $r := \lceil \frac{n}{\epsilon} \rceil$; $R = \lceil \frac{n}{\epsilon\delta} \rceil$; COMMENT: R & r are integral versions of W_i & δW_i .
8. **for** $i := 1$ to $\log n - 1$ **do**
9. $E_i :=$ set of (sorted) edges of E with weights in $(W_i, W_{i+1}]$
10. Build Integer tree with values $\{1, \dots, cn\}$
11. INTEGRALIZE($E', 1$)
12. $C_1 :=$ NAIVE-CENTERS($G', \delta W_1$);
13. **for** $i := 1$ to $\log n$ **do**
14. INTEGRALIZE(E_i, i)
15. $H :=$ CLUSTER-GRAPH(G', Iwt, C_i, r, R)
16. **for** each edge $e = (u, v) \in E_i$ in increasing order **do**
17. **if** not SHORT-PATH($H, u, v, \frac{n\sqrt{tt'}d(u,v)}{\epsilon\delta W_i}$) **then**
18. $E' := E' \cup \{e\}$
19. $G' := (V, E')$
20. INTEREDGESTYPE2(u, v)
21. $C_{i+1} :=$ UPDATE-CENTERS(H, i, C_i, r)
22. REINTEGRALIZE(E')
23. 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 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 , the unit integer length will correspond to the 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.

3.1.1. Error bounds. As defined above, we observe that the integralization function Iwt always involves a rounding up ($Iwt_i(e) \cdot U_i \geq wt(e)$). Thus, in phase i , the error in the length of any single edge 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, 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 3. *In phase i , if $D_{\{G', wt\}}(u, v) > W_i$ for some $u, v \in G'$, then*

$$D_{\{G', wt\}}(u, v) \leq D_{\{G', Iwt\}}(u, v) \cdot U_i < \left(1 + \frac{4t}{c}\right) \cdot D_{\{G', wt\}}(u, v).$$

Proof. Since $wt(e) \leq Iwt_i \cdot U_i = \lceil \frac{wt(e)}{U_i} \rceil \cdot U_i < wt(e) + U_i$, we have

$$\begin{aligned} D_{\{G', wt\}}(u, v) &\leq D_{\{G', Iwt\}}(u, v) \cdot U_i \\ &< D_{\{G', Iwt\}}(u, v) + nU_i \\ &= D_{\{G', wt\}}(u, v) + \frac{4tW_i}{c} \\ &< \left(1 + \frac{4t}{c}\right) D_{\{G', wt\}}(u, v). \quad \square \end{aligned}$$

As a direct consequence, we obtain the following important corollary.

COROLLARY 4. *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 $\frac{nU_i}{\delta W_i} = \frac{4t}{c\delta}$.*

3.1.2. 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 $j = 0, \dots, N$. We assume that the function $\text{INTEGRALIZE}(E_i, i)$ 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 our model but can be easily accomplished by using $O(n)$ preprocessing. One way to accomplish this would be to build a balanced leaf-oriented binary tree including $c \cdot n$ leaves with the values $1, \dots, cn$. Every element in the tree, with value val , also contains a pointer to the element in the tree containing the value $\lceil \frac{val}{2} \rceil$. Assume for simplicity that $c \cdot n = 2^{c'}$. The tree can be built top-down in linear time. The root will have value 1. Consider a node v with value ℓ . The left child of v will have the value $2\ell - 1$, and the right child will have value 2ℓ . This step is repeated until all of the leaves are at level c' . 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 3 still holds. It is clear that $\text{REINTEGRALIZE}(E')$ 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 $O(n \log n)$ time 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 we have 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. Figure 3 illustrates a cluster and a cluster cover.

DEFINITION 5 (cluster, cluster center, and radius). *Given a vertex $v \in V$ and a nonnegative real value r , $\text{CLUSTER}(G, v, r, w)$ 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.*

FIG. 3. (a) A cluster with center at v and radius r . (b) The clusters K_1, \dots, K_7 form a cluster cover.

DEFINITION 6 (cluster-cover). *Given a set of cluster centers $C = \{v_1, \dots, v_m\} \subseteq V$ and a radius r , the $\text{CLUSTER-COVER}(G, C, r, w)$ (if it exists) is a set of clusters $K = \{K_1, \dots, K_m\}$ such that K_i , for $1 \leq i \leq m$, is a cluster with radius r and cluster center v_i and such that $K_1 \cup K_2 \cup \dots \cup K_m = V$.*

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. In our algorithm, the cluster centers will be reasonably far apart so that the amount of overlap is limited. 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 7 (cluster graph). *Assume that $C = \{v_1, v_2, \dots, v_m\} \subseteq V$ is a given set of cluster centers. For a given radius r , we assume that $K = \{K_1, K_2, \dots, K_m\}$ is equal to $\text{CLUSTER-COVER}(G, C, r, w)$. Given a second radius $R > r$, $\text{CLUSTER-GRAPH}(G, w, C, r, R)$ 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.*

Intracluster edges. For all K_i and for all $u \in K_i$, $[u, v_i] \in E_H$.

Intercluster edges. For all $v_i, v_j \in C$, $[v_i, v_j]$ is an intercluster 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).

3.2.1. 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 straightforward 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 an 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 it 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 Figure 4. This is sufficient for our purposes because of the following reasons. First, the weight function is integral, and the array contains all possible distance values from the cluster centers to vertices in the clusters. Second, 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 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 σ 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 belongs to at most σ clusters, the space complexity of the priority queue will be $O(n \cdot \sigma + R)$. Also, every vertex contains a list of the clusters it belongs to.

It should be noted that this version of Dijkstra's algorithm, as shown in Figure 5, 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 $O(\sigma)$. It should be pointed out that this is the only place where we are unable to eliminate the use of indirect addressing since it is critical that this update be performed efficiently, i.e.,

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.

Algorithm PARALLELDIJKSTRA

```

1.  $Q := \text{INITIALIZE}(G', \text{ClusterCenters})$   $O(n)$ 
2. while  $Q \neq \emptyset$  do  $O(n \cdot \sigma)$ 
3.    $u := \text{EXTRACTMIN}(Q)$ 
4.   for each vertex  $v$  adjacent to  $u$  do  $O(1)$ 
5.      $\text{RELAX}(u, v, lwt)$   $O(\sigma)$ 

```

FIG. 5. *The parallel Dijkstra algorithm.*

in constant time. Also note that an edge (u, v) may be RELAXed several times ($O(\sigma)$ 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. The space complexity of the data structure is $O(n \cdot \sigma + R)$. Furthermore, the RELAX operation has a running time of $O(\sigma)$, and the total number ($O(n \cdot \sigma)$) of EXTRACTMIN operations can be performed in total time $O(n \cdot \sigma + R)$. A careful implementation of cluster cover can be made to run in time $O(n \cdot \sigma^2 + R)$. The value of R in our applications will be $O(n)$; hence the time complexity will be $O(n \cdot \sigma^2)$.

3.2.2. 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 $C = \{v_1, \dots, v_m\}$, 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.

- $\text{FINDCENTERS}(v, K)$: 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.
- $\text{COMPUTEDISTANCE}(v_i, v)$: Given $v \in V$ and a cluster center v_i , it returns the quantity $D_{\{G,w\}}(v, v_i)$ if $D_{\{G,w\}}(v, v_i) \leq R$; otherwise, it returns the value ∞ .

Now the cluster graph $H = (V, E_H)$ is computed easily as follows. The intracluster 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 in pseudocode in Figure 6 and has a running time of $O(n \cdot \sigma)$.

From Definition 7, we have that each intercluster edge can be one of two types—either type 1 or type 2. The type 2 edges are only added after the initial construction

Algorithm INTRAEDGES

```

1. for every vertex  $v \in V$  do
2.    $T := \text{FINDCENTERS}(v, K_r)$ 
3.   for every element  $t \in T$  do
4.      $\text{ADDEDGE}(H, v, t.u, t.d)$ ; COMMENT: add edge  $(v, t.u)$  of weight  $t.d$  to  $H$ 

```

FIG. 6. *Algorithm to add intracluster edges.*

Algorithm INTEREDGESTYPE1

-
1. **for** every cluster center v in K_r **do**
 2. $T := \text{FINDCENTERS}(v, K_R)$
 3. **for** every element $t \in T$ **do**
 4. **if** $(\text{COMPUTEDISTANCE}(v, t.u) \geq r)$ **then**
 5. $\text{ADDEGE}(H, v, t.u, t.d)$
-

FIG. 7. Algorithm to add intercluster edges of type 1.

to maintain the cluster graph and will be considered in the next paragraph. An edge $[v_i, v_j]$ of type 1 is added if $v_i \notin K_j$, $v_j \notin K_i$, 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 intercluster edges of type 1 is done in time $O(n \cdot \sigma^2)$, as shown in the algorithm in Figure 7.

The time complexity of computing the cluster graph is $O(n \cdot \sigma^2)$. 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.

3.2.3. 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 intracluster edges and intercluster 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 intercluster 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 intercluster edges in H . The weight of such edges is 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 shown in Figure 8 runs in $O(\sigma^2)$ time, and it is performed $O(n)$ times.

3.2.4. Selecting the cluster centers for a phase. In order for the `CLUSTERGRAPH` function to be implemented efficiently, it needs to have the set of cluster

Algorithm INTEREDGESTYPE2(u_i, u_j)

-
1. $T_1 := \text{FINDCENTERS}(u_i, K_r)$
 2. $T_2 := \text{FINDCENTERS}(u_j, K_r)$
 3. **for** every $t_1 \in T_1$ **do**
 4. **for** every $t_2 \in T_2$ **do**
 5. $\text{ADDEGE}(H, t_1.u, t_2.u, t_1.d + w(u_i, u_j) + t_2.d)$
-

FIG. 8. Algorithm to add intercluster edges of type 2.

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, and 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 Figure 2. NAIVE-CENTERS runs in $O(\sigma n \log n)$ 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. As a first approximation, the set of cluster centers is always chosen as a subset of the cluster centers used in the previous phase. It may turn out that this choice of cluster centers does not give us a cluster cover. Later we describe how to augment the set of cluster centers to ensure that a cluster cover is obtained.

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 $r/2$.

At the end of phase i , the set of cluster centers for phase $i + 1$ is computed. Initially we set $C_{i+1} := C_i \setminus M_i$; i.e., a subset M_i of the cluster centers is deleted from the list of cluster centers. Later C_{i+1} is augmented appropriately. 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 iteratively 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. We will refer to this process as a "thinning" of centers in C_i . 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. Now set $C_{i+1} := C_i \setminus M_i$. Clearly this process runs in time $O(m \cdot \sigma)$. 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 at least $r'/2$ in the next iteration. Since the integers are rounded up, it is possible that clustering from centers C_{i+1} with radius r in phase $i + 1$ may not give us a cluster cover.

Steps 1 through 3 (described below) are repeated until a clustering from cluster centers C_{i+1} with radius r gives us a cluster cover in phase $i + 1$.

- *Step 1.* Run PARALLELDIJKSTRA from all vertices of C_{i+1} using the integralization of phase $i + 1$. Let K_i be the set of vertices not covered by the clusters centered at C_{i+1} .
- *Step 2.* Greedily pick a subset $K' \subseteq K_i$ such that no two vertices in K' belong to the same cluster from the cluster cover of phase i .
- *Step 3.* If K_i is empty, then **Stop**. Else let K be the result of "thinning" out of the centers in K' , and set $C_{i+1} := C_{i+1} \cup K$.

It is easy to see that, after the above process, the new centers are guaranteed to remain at a distance of at least $r/2$. Second, because the process continues until all vertices are covered, it produces a set of cluster centers that will produce a cluster cover in the next phase. Finally, we argue that the number of times steps 1 through 3 are executed is $O(1)$ per phase. Let \mathcal{C} be a cluster from the cluster cover of phase i . Let \mathcal{P} be the region composed of the circular discs of radius r centered at vertices of \mathcal{C} . In each iteration of steps 1 through 3, at least one vertex of \mathcal{C} is picked for K_i . This vertex is either chosen for K , or else a large constant fraction of its area

FIG. 9. 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.

is covered by some circular discs of radius r centered at vertices of K . Clearly the region \mathcal{P} will be covered by at most a constant number (exponentially proportional to the dimension of the space d) of circular discs of radius r . Thus steps 1 through 3 will only be executed a constant number of times, each of which takes $O(n)$ time.

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 Figure 9. Lemma 8 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 8. Let $C = \{v_1, \dots, v_m\} \subseteq V$ 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 $K = \{K_1, \dots, K_m\}$ is returned by $\text{CLUSTER-COVER}(G, C, c' \cdot r')$, 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 2.2. Similar arguments also show that the number of intercluster edges incident to a cluster center is also a constant (although it might have a large number of intracluster 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 σ is also bounded by a constant. Note that IMPROVED-GREEDY uses 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 8.

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 $D_{\{H,Iwt\}}(u,v) \leq \sqrt{tt'} \cdot d(u,v)/U_i$. 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 9. *IMPROVED-GREEDY runs in time $O(n \log n)$.*

Proof. Following the steps of the IMPROVED-GREEDY algorithm shown in Figure 2, we have that step 1 in the initialization and steps 8 and 11 take time $O(n \log n)$; step 2 takes constant time; step 6 runs in $O(\log n)$ time, while steps 3 and 10 take linear time. The integralization of the weight of the edges in steps 11, 14, and 22 takes a total of $O(\log n)$ time per edge. Since each edge is considered exactly once, the total time spent on integralizing and reintegralizing the weight of the edges is $O(n \log n)$ according to section 3.1. Step 15 requires linear time since σ is bounded by a constant. On line 16, 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 $O(n \log n)$. \square

In 1999, Thorup [16] 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 is not allowed in the computational model used in our algorithm.

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 α close to 1. This is done in Lemmas 12 and 13. Second, we need to show, in Lemma 14, that H is always a valid cluster graph of G' . From these results, we easily obtain Theorem 15, 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 δW_i and the larger radii R_i is W_i , where δ is a positive constant decided in Lemmas 14 and 17. 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 10. *Let K be equal to $\text{CLUSTER}(G',v,\delta W_i,Iwt_i)$, i.e., a cluster with cluster center v and radius δW_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 4 and the fact that a cluster from K with center at v consists of all vertices within integer distance $\delta W_i/U_i$ from v . \square

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 11. *If u is a cluster center and $[u, v]$ is an intracluster edge in H , then*

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

If $[v_j, v_k]$ is an intercluster edge in H , then

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

Proof. The first statement is a direct consequence of Lemma 10; the same holds for the left inequality in (2). The right inequality in the second statement follows from Definition 7 since an intercluster 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 $x \in K_j$, $y \in K_k$, and $Iwt(x, y) < W_i/U_i$. \square

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

LEMMA 12. *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 be *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 intracluster edges but all intermediate edges are intercluster edges.

The next lemma is the approximate converse of Lemma 12.

LEMMA 13. *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. The proof is similar to the proof of Lemma 4 in [8]. 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 intracluster 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 intercluster 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 intercluster 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 intracluster edge $[v_m, v]$, as shown in Figure 10. Three cases arise, and the lemma is proved in each of these cases.

Case 1 ($m = 1$). In this case, there is only one intercluster edge along Q . Since u is sufficiently far from v , we know that $L_1 > W_i - 2(1 + \epsilon)\delta W_i$. Now $L_2 = Iwt([u, v_0]) + Iwt([v_0, v_1]) + Iwt([v_1, v])$. However, $Iwt([u, v_0]) \leq (1 + \epsilon)\delta W_i$ 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

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

FIG. 10. Paths in H approximate paths in G' .

Case 2 ($m \geq 2$ and m even). Suppose $[v_i, v_{i+1}]$ and $[v_{i+1}, v_{i+2}]$ are any two consecutive intercluster 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, \dots , where Q_{2i} is the portion between v_{2i} and v_{2i+2} . Similarly, divide P into portions P_0, P_1, \dots , 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 intercluster edge between v_{2i} and v_{2i+2} , we have that $p_{2i} > W_i - 2\delta(1 + \epsilon)W_i$. Thus $W_i < \frac{p_{2i}}{1 - 2\delta(1 + \epsilon)}$. 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'_{2i} (respectively, p''_{2i}) be the initial (respectively, final) portions; thus $p_{2i} = p'_{2i} + p''_{2i}$. From the procedure INTEREDGESTYPE2, we have $Iwt([v_{2i}, v_{2i+1}]) \leq (1 + \epsilon)p'_{2i} + 2\delta(1 + \epsilon)W_i$, and, similarly, $Iwt([v_{2i+1}, v_{2i+2}]) \leq (1 + \epsilon)p''_{2i} + 2\delta(1 + \epsilon)W_i$. 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 . Thus

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

Summing over all even values of i and taking into account the two intracluster edges at either end 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 odd). The analysis will be exactly the same as in the previous case, except that we have to account for the last intercluster 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 can be rewritten as $q_{m-1} < (\frac{(1 + \epsilon)(1 + 2\delta)}{1 - 2\delta(1 + \epsilon)})p_{m-1} + 2\delta(1 + \epsilon)W_i$. We then sum up as above to 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

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

That completes the proof of the lemma. \square

Before processing group E_i (which contains edges with weights in the range $(W_i, 2W_i]$), the algorithm constructs a fresh cluster graph H using a radius of δW_i .

LEMMA 14. *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}, \dots, 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 $wt(u, v) > W_i$ and $\delta < 1/2$, the distance between any two previous cluster centers in the new G' will remain greater than δW_i , and thus the previous cluster cover will remain valid. Also, the previous intracluster edges and the intercluster edges of type 1 (see the definition of intercluster edges) will remain the same. We have only to make sure that we add new intercluster 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 intercluster 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 3.2.3). We now show that our choice of δ 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 contains only edges of the previous G' . However, 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 13 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 δ , we see that the only positive solutions are

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

FIG. 11. *Illustration of the definition of the leapfrog property.*

According to our choice of δ in the algorithm IMPROVE-GREEDY, the above inequality will never be satisfied (see Figure 2). Hence we have a contradiction. \square

The following theorem now concludes this section.

THEOREM 15. *The graph produced by IMPROVED-GREEDY is a t -spanner of the complete Euclidean graph.*

Proof. The 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 nonintuitive.

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 $S = \{(u_1, v_1), \dots, (u_m, v_m)\}$, which is a subset of E' :

$$t \cdot wt(u_1, v_1) < \sum_{i=2}^m wt(u_i, v_i) + t \cdot \left(\sum_{i=1}^{m-1} wt(v_i, u_{i+1}) + wt(v_m, u_1) \right).$$

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 $t' \cdot wt(u_1, v_1)$, as shown in Figure 11. The following fact was shown by Das and Narasimhan [8].

FACT 16 (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 .*

Now suppose that we construct a t -spanner such that, for every spanner edge (u, v) , the second shortest path is not necessarily 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 $wt(E' \setminus E_0)$, where E' is the set of edges produced by the algorithm.

LEMMA 17. Let $e=(u, v) \in E' \setminus E_0$. 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 $e_1 \in E' \setminus E_0$, 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 $wt(C) - wt(u_1, v_1)$. However, 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_i . 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 13 implies that the weight of each path in G' between u_1 and v_1 is large; i.e.,

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

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

Finally, since Lemma 17 holds, we can use the following observation which, together with Fact 16, concludes the proof of Theorem 1.

OBSERVATION 18. $E' \setminus E_0$ satisfies the (t', t) -leapfrog property.

Proof. Consider any subset of the edges $S = \{(u_1, v_1), \dots, (u_m, v_m)\}$ of E' . By Lemma 17, 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 . However, $wt(P)$ is also equal to the right-hand side of the definition of the leapfrog property. The observation follows. \square

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 a small number of edges (only $O(n)$ edges). The implementation of clustering techniques is of independent interest in the design of efficient algorithms.

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.

Acknowledgments. We are grateful to Michiel Smid for numerous helpful discussions. We are also very grateful to a diligent referee for many good comments and for helping us fix a subtle error in section 3.2.4.

REFERENCES

[1] I. ALTHÖFER, G. DAS, D. P. DOBKIN, D. JOSEPH, AND J. SOARES, *On sparse spanners of weighted graphs*, Discrete Comput. Geom., 9 (1993), pp. 81–100.

- [2] S. ARYA, G. DAS, D. M. MOUNT, J. S. SALOWE, AND M. SMID, *Euclidean spanners: Short, thin, and lanky*, in Proceedings of the 27th Annual ACM Symposium on Theory of Computing, ACM, New York, 1995, pp. 489–498.
- [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 (1995), pp. 67–90.
- [4] B. CHANDRA, G. DAS, G. NARASIMHAN, AND J. SOARES, *New sparseness results on graph spanners*, Internat. J. Comput. Geom. Appl., 5 (1995), pp. 124–144.
- [5] D. Z. CHEN, G. DAS, AND M. SMID, *Lower bounds for computing geometric spanners and approximate shortest paths*, in Proceedings of the 8th Annual Canadian Conference on Computational Geometry, Ottawa, Canada, 1996, pp. 155–160.
- [6] A. CZUMAJ AND A. LINGAS, *Linear-time heuristics for minimum weight rectangulation*, in Proceedings of the 27th Colloquium on Automata, Languages and Programming, Lecture Notes in Comput. Sci. 1853, Springer-Verlag, New York, 2000, pp. 856–868.
- [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, ACM, New York, 1993, pp. 53–62.
- [8] G. DAS AND G. NARASIMHAN, *A fast algorithm for constructing sparse Euclidean spanners*, Internat. J. Comput. Geom. Appl., 7 (1997), pp. 297–315.
- [9] J. M. KEIL AND C. A. GUTWIN, *Classes of graphs which approximate the complete Euclidean graph*, Discrete Comput. Geom., 7 (1992), pp. 13–28.
- [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 (1992), pp. 251–256.
- [11] C. LEVCOPOULOS, G. NARASIMHAN, AND M. SMID, *Efficient algorithms for constructing fault-tolerant geometric spanners*, in Proceedings of the 30th Annual ACM Symposium on Theory of Computing, ACM, New York, 1998, pp. 186–195.
- [12] D. MOUNT, Private communication, Department of Computer Science, University of Maryland, College Park, MD, 1998.
- [13] F. P. PREPARATA AND M. I. SHAMOS, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.
- [14] S. B. RAO AND W. D. SMITH, *Improved approximation schemes for traveling salesman tours*, in Proceedings of the 29th Annual ACM Symposium on Theory of Computing, ACM, New York, 1998, pp. 540–550.
- [15] J. S. SALOWE, *Construction of multidimensional spanner graphs with applications to minimum spanning trees*, in Proceedings of the 7th Annual ACM Symposium on Computational Geometry, ACM, New York, 1991, pp. 256–261.
- [16] M. THORUP, *Undirected single-source shortest path with positive integer weights in linear time*, J. ACM, 46 (1999), pp. 362–394.
- [17] P. M. VAIDYA, *A sparse graph almost as good as the complete graph on points in K dimensions*, Discrete Comput. Geom., 6 (1991), pp. 369–381.