Balancing Minimum Spanning Trees and Shortest-Path Trees

Samir Khuller *
University of Maryland

Balaji Raghavachari †
Penn State

Neal Young ‡
University of Maryland

Abstract

We give a simple algorithm to find a spanning tree that simultaneously approximates a shortest-path tree and a minimum spanning tree. The algorithm provides a continuous trade-off: given the two trees and a $\gamma > 0$, the algorithm returns a spanning tree in which the distance between any vertex and the root of the shortest-path tree is at most $1 + \sqrt{2}\gamma$ times the shortest-path distance, and yet the total weight of the tree is at most $1 + \sqrt{2}/\gamma$ times the weight of a minimum spanning tree.

Our algorithm runs in linear time and obtains the best-possible trade-off. It can be implemented on a CREW PRAM to run in logarithmic time using one processor per vertex.

Keywords: minimum spanning trees, graph algorithms, parallel algorithms, shortest paths.

*Computer Science Department, University of Maryland, College Park, MD 20742. E-mail : `samir@cs.umd.edu`. Part of this work was done while this author was at University of Maryland Institute for Advanced Computer Studies (UMIACS) and was supported by NSF grants CCR-8906949, CCR-9103135 and CCR-9111348.

†Computer Science Department, Pennsylvania State University, University Park, PA 16802. E-mail : `rbk@cs.psu.edu`.

‡University of Maryland Institute for Advanced Computer Studies, College Park, MD 20742. E-mail : `young@umiacs.umd.edu`. Supported by NSF grants CCR-8906949 and CCR-9111348.

1 Introduction

A *minimum spanning tree* of an edge-weighted graph is a spanning tree of the graph of minimum total edge weight. A *shortest-path tree* rooted at a vertex r is a spanning tree such that for any vertex v, the distance between r and v is the same as in the graph.

Minimum spanning trees and shortest-path trees are fundamental structures in the study of graph algorithms [10, 11, 15, 19]; fast algorithms for finding each are known [12, 13]. Typically, the edge-weighted graph G represents a feasible network. Each vertex represents a site. The goal is to install links between pairs of sites so that signals can be routed in the resulting network. Each edge of G represents a link that can be installed. The cost of the edge reflects both the cost to install the link and the cost (e.g., time) for a signal to traverse the link once the link is installed. A minimum spanning tree represents the least costly set of links to install so that all sites are directly or indirectly connected, while a shortest-path tree represents the set of links to install so that for each site, the cost for a signal to be sent between the site and the root of the tree is as small as possible.

The goal of a minimum spanning tree is minimum weight, whereas the goal of a shortest-path tree is to preserve distances from the root. We show that a single tree can approximately achieve both goals. That is, the cost to install a set of links so that every site has a short path to the root is only slightly more than the cost just to connect all sites.

In the graph in Figure 1, the weight of the shortest-path tree is much more than the weight of a minimum spanning tree. Conversely, in the minimum spanning tree, the distance between the root and one of the vertices is much larger than the corresponding shortest-path distance. Nonetheless, there is a tree which nearly preserves distances from the root and yet weighs only a little more than the minimum spanning tree. We call such a tree a *Light Approximate Shortest-path Tree* (LAST). The main result of this paper is that such trees exist in all graphs and can be found efficiently.

Let $G = (V, E)$ be a graph with non-negative edge weights and a *root* vertex r. Let G have n vertices and m edges. Let $w(e)$ be the weight of edge $e \in E$. The *distance* $D_G(u, v)$ between vertices u and v in G is the minimum weight of any path in G between them.

Definition 1 *For $\alpha \geq 1$ and $\beta \geq 1$, a spanning tree T of G meeting the following two requirements is called an (α, β)-*LAST *rooted at r.*

- *(Distance) For every vertex v, the distance between r and v in T is at most α times the shortest distance from r to v in G.*
- *(Weight) The weight of T is at most β times the weight of a minimum spanning tree of G.*

Theorem 1 (Section 3) *Let G be a graph with non-negative edge weights; let r be a vertex of G; let $\alpha > 1$ and $\beta \geq 1 + \frac{2}{\alpha - 1}$. Then G contains an (α, β)-LAST rooted at r. The LAST can be computed in linear time given a minimum spanning tree and a shortest-path tree, and in $O(m + n \log n)$ time otherwise.*

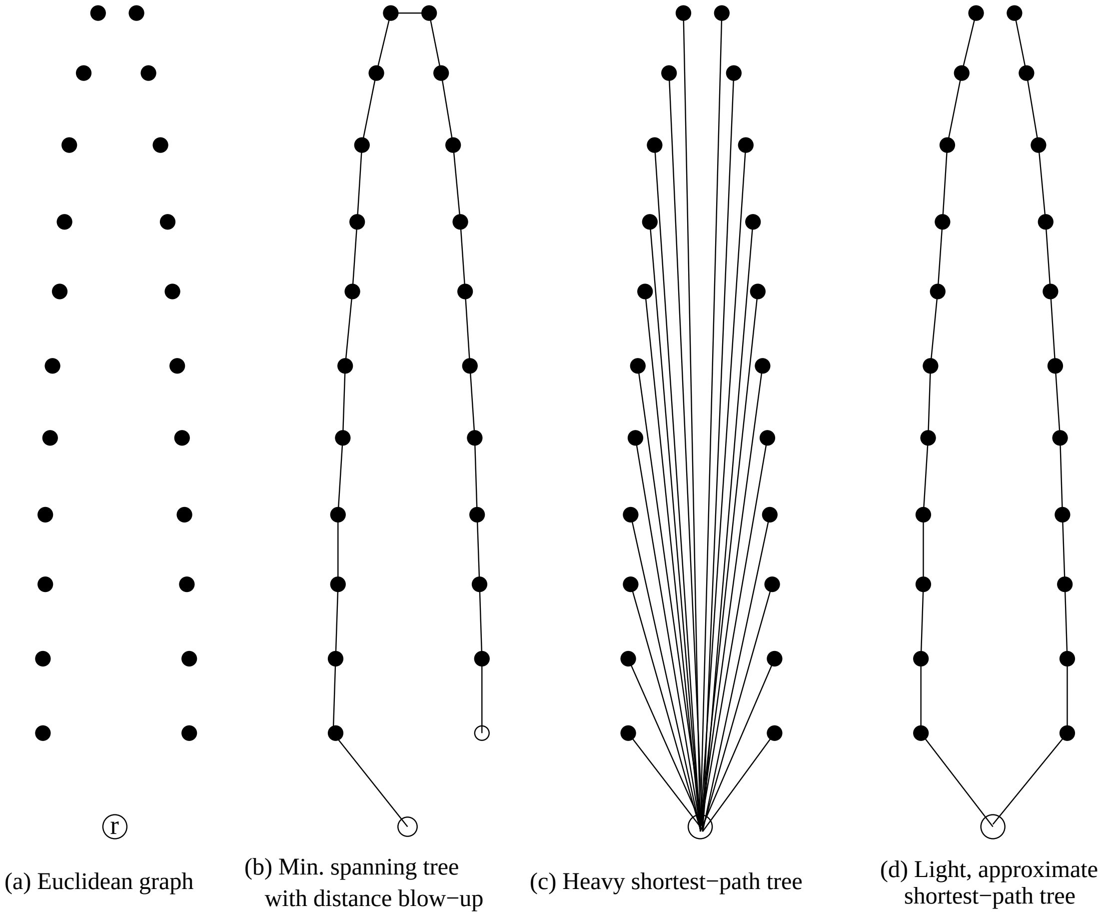

Figure 1: Approximating both a minimum spanning tree and a shortest-path tree.

Note that there is a trade-off between the approximations of the two trees. The trade-off is the best possible:

Theorem 2 (Section 4) *Fix $\alpha > 1$ and $1 \leq \beta < 1 + \frac{2}{\alpha-1}$. There exists a planar graph G with a vertex r such that G contains no (α, β)-LAST rooted at r. Deciding whether a given graph contains an (α, β)-LAST rooted at a given vertex is NP-complete.*

Note that for $\beta = 1$, the problem is to find a minimum spanning tree that best approximates a shortest-path tree. It follows from Theorem 2 that this is NP-complete. When $\alpha = 1$, the problem is to find a minimum-weight shortest-path tree. This problem can be solved in linear time, even in directed graphs:

Theorem 3 (Section 5) *Given any shortest-path tree of a directed or undirected graph rooted at a given vertex, a minimum-weight shortest-path tree can be found in linear time.*

Finally, LAST's can also be found quickly in parallel, given a minimum spanning tree and shortest-path tree (or approximations thereof, see Section 3.4):

Theorem 4 (Section 6) *Given $\alpha > 1$, a minimum spanning tree, and a shortest-path tree, an $(\alpha, 1 + \frac{2}{\alpha-1})$-LAST can be found by n processors in $O(\log n)$ time on a CREW PRAM.*

2 Related Work

Trees realizing tradeoffs between weight and distance requirements were first studied by Bharath-Kumar and Jaffe [4]. The authors' weight requirement was the same as ours, but their distance requirement was that the *sum* of the distances from the root to each vertex should be at most β times the minimum possible sum. They showed the weaker tradeoff that the desired tree exists if $\alpha\beta \geq \Theta(n)$.

Awerbuch, Baratz and Peleg [2], motivated by applications in broadcast-network design, made a fundamental contribution by showing that every graph has a *shallow-light* tree — a tree of *diameter* at most a constant times the diameter of G and of weight at most a constant times the weight of the minimum spanning tree. Our algorithm is modification of their algorithm. Cong, Kahng, Robins, Sarrafzadeh and Wong [7, 8, 9], motivated by applications in VLSI-circuit design, improve the constants in the construction of [2] and consider variations bounding the *radius* of the tree instead of the diameter. Recently and independently, Awerbuch, Baratz and Peleg [3], modified the algorithm from [2]. They obtained the same algorithm as in [8] but a stronger analysis, proving that the algorithm computes an $(\alpha, 1 + \frac{4}{\alpha-1})$-LAST. Their algorithm takes $O(m + n \log n)$ time. Our algorithm achieves a strictly stronger distance requirement than the above algorithms.

Considerable research has been done on finding *spanners* of small size and weight in arbitrary graphs [1, 5, 17] and in Euclidean graphs (graphs induced by points in the plane) [5, 6, 16, 20]. A t-spanner is a low-weight subgraph G' of G such that for any two vertices, the distance between them in G' is at most t times the distance in G. It is known that there are graphs that *do not* have constant-spanners of net weight bounded by a constant times the weight of the minimum spanning tree. Awerbuch, Baratz and Peleg [3] also consider light trees that have low *average* distance-blowup on all non-tree edges. References to most of the work on graph spanners may be found in the paper by Chandra, Das, Narasimhan and Soares [5].

We can reduce the problem of finding an $(\alpha, 1+\frac{2}{\alpha-1})$-LAST to the problem of finding an α-spanner of weight at most $\left(1+\frac{2}{\alpha-1}\right)$ times the minimum spanning-tree weight in a planar graph. An algorithm achieving the latter is given in [1]. This gives an alternate (but less efficient) method of for finding an $(\alpha, 1+\frac{2}{\alpha-1})$-LAST.

3 The Algorithm

The algorithm is given an $\alpha > 1$, a minimum spanning tree, and a shortest-path tree rooted at a vertex r. It returns an $(\alpha, 1+\frac{2}{\alpha-1})$-LAST rooted at r.

The basic idea of the algorithm is to traverse the minimum spanning tree, maintaining a *current tree*, and checking each vertex when it is encountered to ensure that the distance requirement for that vertex is met in the current tree. If it is not met, the edges of the shortest path between the vertex and the root are added into the current tree. Other edges are discarded so that a tree structure is maintained.

After all vertices have been checked and paths added as necessary, the remaining tree is the desired LAST. The final tree is not too heavy because a shortest-path is only added if the path that it replaces is heavier by a factor of $\alpha > 1$. This allows a charging argument bounding the net weight of the added paths.

3.1 Relaxation.

The tree is maintained by keeping a parent pointer $p[v]$ for each non-root vertex v. To avoid recomputing shortest-path distances when a path is added, the algorithm maintains a *distance estimate* $d[v]$ for each vertex v. This distance estimate, which is an upper bound on the true distance in the current tree, is used in deciding whether to add a path to the vertex. The parent pointers and distance estimates are initialized and maintained as in [10, Section 25.1]:

INITIALIZE()
Initialize distance estimates, parent pointers.
for each non-root vertex v do $p[v] \leftarrow$ **nil**; $d[v] \leftarrow \infty$
$d[r] \leftarrow 0$

RELAX(u, v)
Check for shorter path to v through (u, v).
if $d[v] > d[u] + w(u, v)$
  then $d[v] \leftarrow d[u] + w(u, v)$
    $p[v] \leftarrow u$

After executing INITIALIZE, the algorithm builds and updates the tree and maintains the distance estimates by a sequence of calls to RELAX. The important invariant maintained by RELAX is that the edges $\{(p[v], v) : d[v] \neq \infty\}$ form a tree, with $d[v]$ an upper bound on the distance between the root and v in the tree.

3.2 The algorithm as a sequence of relaxations.

The algorithm performs a depth-first search of the minimum spanning tree starting at the root. For a tree, a depth-first search is simply an edge-by-edge walk from the root vertex through the vertices of the tree. Each edge is traversed twice: once in each direction. At any time in the search, the sequence of edges traversed so far forms a walk (a non-simple path) through the visited vertices. The walk starts at the root and ends at a vertex that we call the *current vertex*. We also say the algorithm is *visiting* this vertex.

The relaxations done by the algorithm are of two kinds. The first kind adds shortest paths. The first time vertex v is visited, if $d[v]$ exceeds α times the distance from the root to v in the shortest-path tree, then the edges of the shortest path are relaxed as needed to lower $d[v]$ to the shortest-path distance.

The second kind extends or modifies the current tree to use a minimum-spanning-tree edge if it is useful. Specifically, when an edge (u, v) is traversed from u to v, RELAX(u, v) is called. This guarantees inductively that $d[v]$ is bounded by the weight of the shortest path from the root r to vertex v' plus the weight of the minimum-spanning-tree path from v' to v, where v' is the most recent vertex to have its shortest path added. This invariant is what allows the weight of the added paths to be bounded.

When the depth-first search finishes, the current tree is the desired LAST. The full algorithm is given in Figure 2.

3.3 A sample execution.

Figure 3 shows a sample execution of the algorithm with $\alpha = 2$ on the graph given in Frame (a). Frames (b) and (c) give, respectively, a minimum spanning tree (of weight 60) and a shortest-path tree.

Initially all parent pointers are **nil** and each $d[v]$ is infinite. The depth-first search of the minimum spanning tree visits the vertices in increasing order of their labels and traverses the edges of the minimum-spanning tree in the following order:

$$(1,2),(2,3),(3,4),(4,5),(5,4),(4,6),(6,7),(7,6),(6,4),(4,3),(3,2),(2,1),(1,8),(8,1)$$

Recall that when an edge is traversed, it is relaxed. When a vertex is visited, if its current distance estimate is not small enough to guarantee the distance requirement then the edges on the shortest path to the vertex are relaxed, modifying the current tree.

Frame (d) shows the state of the algorithm just after vertex 5 has been visited: the edges $(1, 2)$, $(2, 3)$, $(3, 4)$, and $(4, 5)$ were relaxed as they were traversed. Because $d[5]$ was equal to 40 (more than twice the shortest-path distance) when vertex v was visited, edge $(1, 5)$ was relaxed, changing vertex 5's parent to vertex 1, and changing $d[5]$ to 15.

Frame (e) shows the state after vertex 7 — the next vertex to have its shortest path added — has been visited. Note that when edge $(5, 4)$ was traversed, from 5 to 4, its

```
FIND-LAST(T_M, T_S, r, α)
Input: Min. spanning tree T_M, shortest-path tree T_S, vertex r, α > 1.
Output: an (α, 1 + 2/(α−1))-LAST rooted at r.
INITIALIZE()
DFS(r)
return tree T = {(v, p[v]) | v ∈ V − {r}}

DFS(u)
Traverse the subtree of T_M rooted at u, relaxing edges as they are tra-
versed, and adding paths from T_S as needed.
if d[u] > α D_{T_S}(r, u)
  then ADD-PATH(u)
for each child v of u in T_M
    do RELAX(u, v)
       DFS(v)
       RELAX(v, u)

ADD-PATH(v)
Relax edges along path from r to v in T_S.
if d[v] > D_{T_S}(r, v)
  then ADD-PATH(parent_{T_S}(v))
       RELAX(parent_{T_S}(v), v)
```

Figure 2: Algorithm to compute a LAST.

relaxation changed vertex 4's parent to vertex 5 and updated $d[4]$ to reflect the new shorter path $(1,5),(5,4)$. The algorithm then traversed and relaxed edges $(4,6)$ and $(6,7)$, bringing vertices 6 and 7 into the tree. When vertex 7 was encountered, its distance estimate (40) exceeded twice the shortest-path distance (15), so the edges on the shortest path $(1,8),(8,7)$ to vertex 7 were relaxed in that order. This added these edges to the current tree and brought down the distance estimates of these vertices.

Frame (f) shows the final state of the algorithm. The parent pointers give the final tree. Note that the relaxation of edge $(7,6)$ from 7 to 6, changed vertex 6's parent. This was the final change made to the tree. Subsequent relaxations made by the traversal had no effect. Remaining distance estimates were small enough to guarantee that the distance requirements were met, so that ADD-PATH was not called.

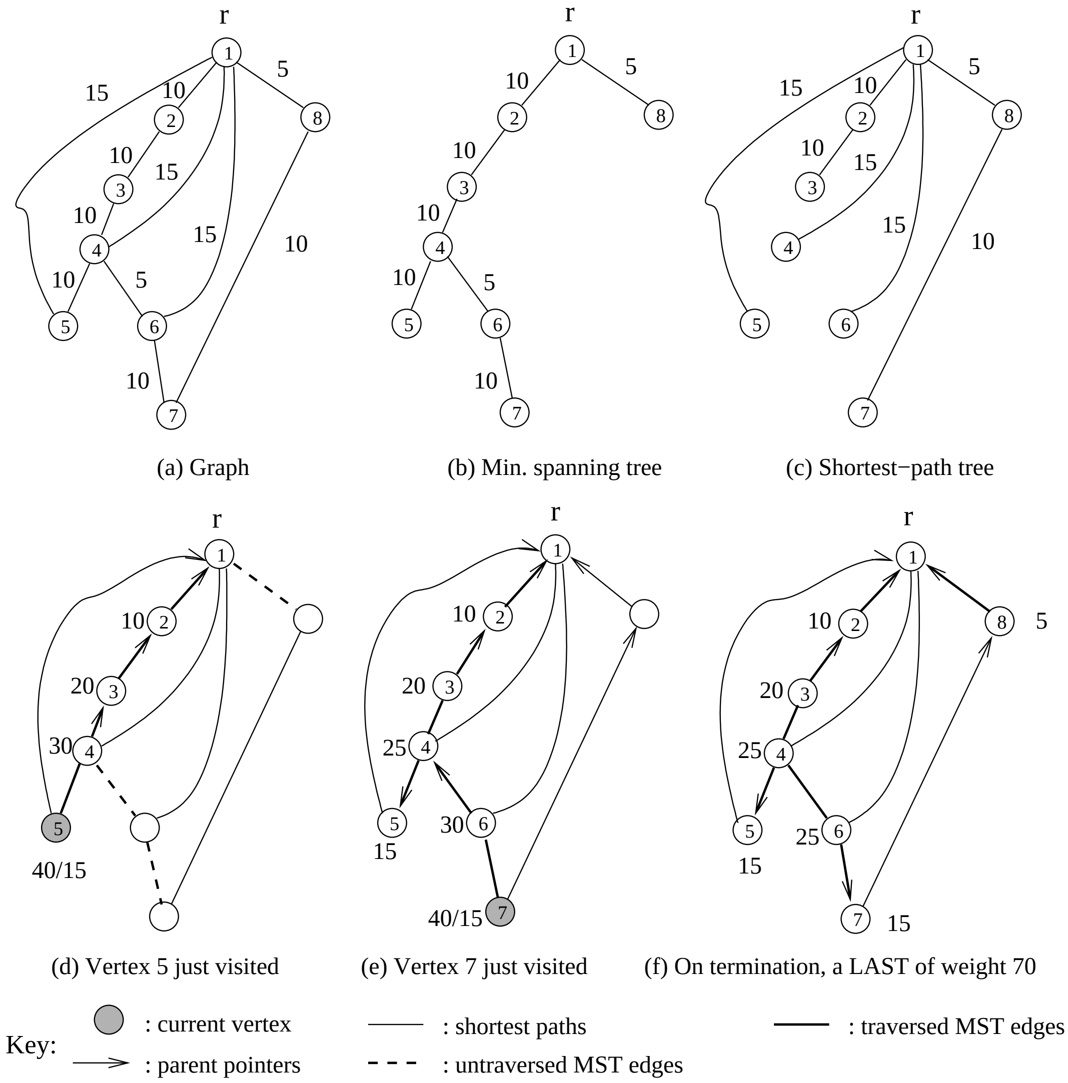

Figure 3: A sample execution of the algorithm.

3.4 Analysis of the algorithm.

Next we prove that FIND-LAST(T_M, T_S, r, α) returns an $(\alpha, 1+\frac{2}{\alpha-1})$-LAST in linear time. Let T be the tree returned.

Lemma 3.1 *The distance between v and r in T is at most α times the shortest-path distance.*

Proof. When a vertex v is visited, if $d[v]$ exceeds α times the distance in the shortest-path tree then ADD-PATH is called, after which $d[v]$ equals the shortest-path distance. In any case, after v is visited, $d[v]$ is at most α times the shortest-path distance and subsequently never increases. On termination it bounds the distance in T. □

An amortized analysis establishes that the total weight of T is not too large.

Lemma 3.2 *The weight of T is at most $\left(1+\frac{2}{\alpha-1}\right)$ times the minimum spanning-tree weight.*

Proof. Let $v_0 = r$ and let $v_1, v_2, ..., v_k$ be the vertices that caused shortest paths to be added during the traversal, in the order they were encountered. When the shortest path from r to v_i $(i \geq 1)$ was added, the net weight of the added edges was at most $D_{T_S}(r, v_i)$. Also, the edges on the path to v_i consisting of the shortest path to v_{i-1} followed by the path in the minimum spanning tree from v_{i-1} to v_i had been relaxed in order, so that $d[v_i] \leq D_{T_S}(r, v_{i-1}) + D_{T_M}(v_{i-1}, v_i)$. The shortest path to v_i was added because $\alpha D_{T_S}(r, v_i) < d[v_i]$. Combining the inequalities,

$$\alpha D_{T_S}(r, v_i) < D_{T_S}(r, v_{i-1}) + D_{T_M}(v_{i-1}, v_i).$$

Summing over i bounds the net weight of the added paths:

$$\alpha \sum_{i=1}^{k} D_{T_S}(r, v_i) < \sum_{i=1}^{k} \left(D_{T_S}(r, v_{i-1}) + D_{T_M}(v_{i-1}, v_i)\right)$$

and therefore

$$(\alpha - 1) \sum_{i=1}^{k} D_{T_S}(r, v_i) < \sum_{i=1}^{k} D_{T_M}(v_{i-1}, v_i).$$

The DFS traversal traverses each edge exactly twice, and hence the sum on the right-hand side is at most twice the weight of T_M, i.e.,

$$\sum_{i=1}^{k} D_{T_M}(v_{i-1}, v_i) \leq 2\, w(T_M).$$

Hence the net weight of the added paths is less than $\frac{2}{\alpha-1} w(T_M)$. □

The following alternate proof of Lemma 3.2 may also be of interest.

Alternate Proof of Lemma 3.2. As the algorithm executes, define the potential function Φ to be the distance estimate of the current vertex. When a shortest path of length p to the current vertex v is added, $\Phi = d[v] > \alpha\, p$. Adding the path lowers $d[v]$ to p, decreasing

Φ by at least $(\alpha-1)\,p$. Hence the total weight of the added paths is bounded by the sum of the decrements to Φ during the course of the algorithm, divided by $\alpha-1$.

Since Φ is initially 0 and always non-negative, the sum of the decreases is at most the sum of the increments. Φ increases only when the current vertex changes from some vertex u to a vertex v after the edge (u,v) was relaxed. This ensures that $d[v] \leq d[u] + w(u,v)$ and that Φ increases by at most $w(u,v)$. Since each edge is traversed twice, the total of the increases to Φ during the course of the algorithm is bounded by twice the weight of the minimum spanning tree.

This establishes that the total weight of the added paths is bounded by $\frac{2}{\alpha-1}$ times the weight of the minimum spanning tree. □

The running time is proportional to the number of relaxations. This is $O(n)$ because each edge in T_M or T_S is relaxed at most twice by DFS and at most once by ADD-PATH. If the shortest-path tree and the minimum spanning tree are not given, they can be computed in $O(m + n\log n)$ time [12, 13]. This establishes Theorem 1.

Observation 1: In metric graphs (complete graphs with edge weights satisfying the triangle inequality, such as Euclidean graphs) the shortest-path tree is trivial and can be found in $O(n)$ time. For Euclidean graphs (induced by points in the plane) the minimum spanning tree can be computed in $O(n\log n)$ time [18]. In these cases the LAST can be found more quickly.

Observation 2: If the algorithm is given an *a-approximate* shortest-path tree and a *b-approximate* minimum spanning tree, the tree returned by the algorithm will be an $(a\alpha, b + 2b/(\alpha-1))$-LAST. If such trees can be found more quickly then a LAST can also be found more quickly.

Observation 3: In the *multiple-root* variant, the distance requirement is that in the final tree (or forest) the distance between each vertex and its *nearest* root should be at most α times the distance to any root in the original graph. This variant can be easily reduced to the original problem by adding an artificial root at distance 0 from the multiple roots.

4 Optimality of the Algorithm

Next we show that the algorithm is optimal in the following sense. Fix $\alpha > 1$ and $1 \leq \beta < 1 + \frac{2}{\alpha-1}$. There is a planar graph not containing an (α,β)-LAST rooted at a particular vertex. Further, it is NP-complete to decide whether a given graph contains an (α,β)-LAST from a given root.

4.1 Non-existence of LAST's when $\beta < 1 + \frac{2}{\alpha-1}$.

Lemma 4.1 *If $\alpha > 1$ and $1 \leq \beta < 1 + \frac{2}{\alpha-1}$, then there exists a planar graph containing no (α,β)-LAST rooted at a particular vertex.*

Proof. The graph is shown in Figure 4. The structure of the graph is as follows. The root r is connected to a central vertex c by a path of weight A of edges of weight some small δ. The central vertex is connected through similar paths of weight B to the ℓ leaves. The

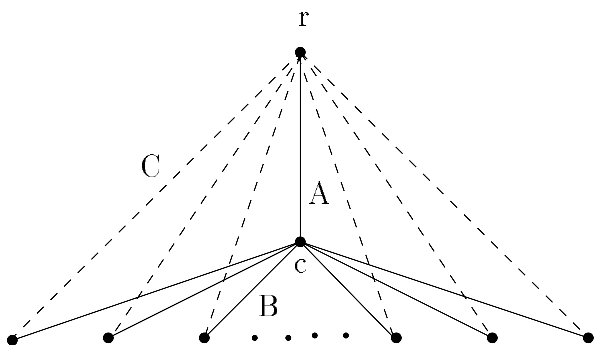

Figure 4: A graph with no (α, β)-LAST for $\beta < 1 + \frac{2}{\alpha-1}$, ($A = \alpha + 1$, $B = \alpha + \epsilon - 1$ and $C = 2$).

root is connected to each leaf with an edge of weight C. Let $A = \alpha + 1$, $B = \alpha + \epsilon - 1$ and $C = 2$, where ϵ is an arbitrarily small constant. For small enough δ, the minimum spanning tree is formed by using all edges except those of weight C. Notice that this graph is planar.

Consider the paths from the root to any leaf. The shortest path is the direct edge of weight 2. Any other path weighs more than 2α: the path through the center vertex weighs $A + B = 2\alpha + \epsilon$; any path through another leaf weighs at least $2 + 2B = 2(\alpha + \epsilon)$. This means that in any (α, β)-LAST all ℓ edges of weight 2 are present. In addition, all but ℓ of the remaining edges are present. Therefore the weight of any (α, β)-LAST is at least $2\ell + T_M - \ell\delta$, where $T_M = (\alpha+1) + \ell(\alpha - 1 + \epsilon)$ is the weight of the minimum spanning tree. Hence the ratio of the weight of the (α, β)-LAST to the weight of the minimum spanning tree is at least

$$1 + \frac{\ell(2 - \delta)}{\alpha + 1 + \ell(\alpha - 1 + \epsilon)}$$

If $\beta < 1 + \frac{2}{\alpha-1}$, then the above exceeds β for sufficiently small ϵ and δ and sufficiently large ℓ. $\square$

4.2 NP-completeness of LAST queries.

Next we show that for any fixed $\alpha > 1$ and $1 \leq \beta < 1 + \frac{2}{\alpha-1}$ is NP-hard to decide whether a given graph contains an (α, β)-LAST rooted at a given vertex. Thus, it is unlikely that a polynomial-time algorithm exists for finding (α, β)-LAST's when $\beta < 1 + \frac{2}{\alpha-1}$.

Clearly the problem is in NP. The proof of NP-hardness is in two parts. We first show NP-hardness for $\beta = 1$ and fixed $\alpha > 1$. We then reduce this problem to the fixed $\beta < 1 + \frac{2}{\alpha-1}$ case.

Lemma 4.2 *For fixed $\alpha > 1$, deciding the existence of an $(\alpha, 1)$-LAST rooted at a given vertex of a given graph is NP-hard.*

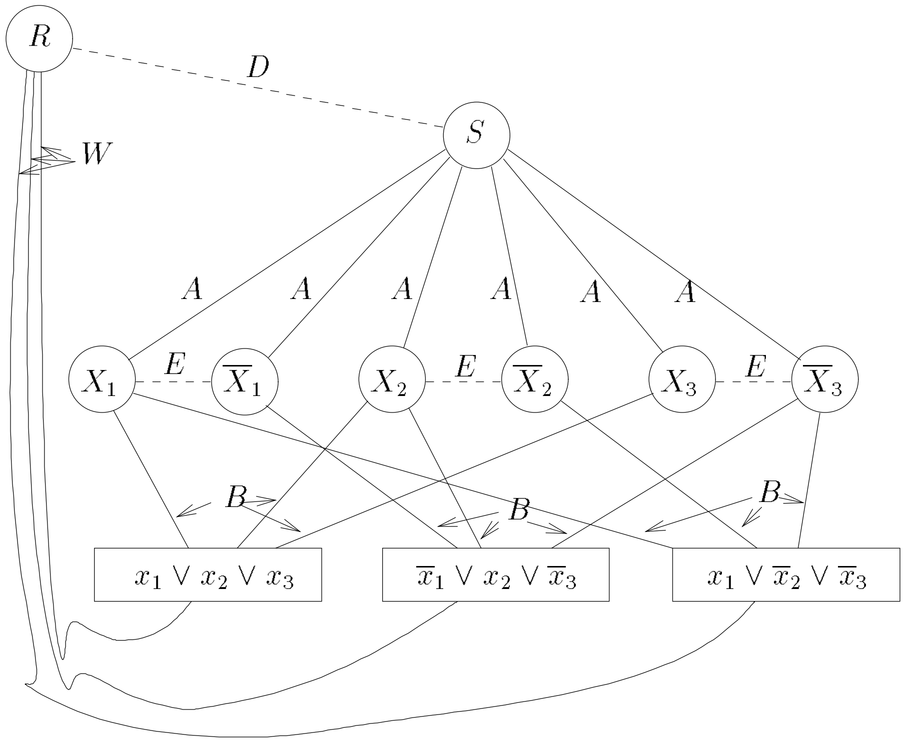

Figure 5: Reduction From 3-SAT.

Proof. The proof is by reduction from 3-SAT. Let F be a 3-SAT formula in conjunctive normal form — each clause consists of three literals from $\{x_1, \ldots, x_n\} \cup \{\overline{x}_1, \ldots, \overline{x}_n\}$. We build a graph in which the $(\alpha, 1)$-LAST's correspond to satisfying assignments of F.

A, B, D, E and W are constants to be determined later. The graph has a root vertex R, a vertex S, and a path connecting R to S of weight D consisting of edges small enough to ensure that the path is in any minimum spanning tree.

For each pair of literals x_i and $\overline{x}_i$, there are two vertices X_i and $\overline{X}_i$, each having an edge to S of weight A. A path of weight E connects X_i and $\overline{X_i}$. This path is also constructed so as to be in any minimum spanning tree.

For each clause c_j there is a vertex C_j with an edge to R of weight W. From C_j to each vertex corresponding to a literal in c_j there is an edge of weight B.

This defines the graph. Observe that, provided $0 < A < B < W$, the minimum spanning trees are exactly characterized by the following. In any minimum spanning tree, the path from R to S and each path from X_i to $\overline{X}_i$ are present. For each variable x_i, exactly one of the two edges $\{(S, X_i), (S, \overline{X}_i)\}$ is present. For each clause c_j, exactly one edge of the form (X_i, C_j) or $(\overline{X}_i, C_j)$ for some i is present. No other edges are present.

Next we use the distance requirement to ensure that any minimum spanning tree is an $(\alpha, 1)$-LAST if and only if the edge to each clause vertex comes from some variable vertex X_i or $\overline{X}_i$ that has an edge in the minimum spanning tree directly to S. This is all that is needed, for then the $(\alpha, 1)$-LAST's will correspond to satisfying assignments in the original formula, and vice versa, as follows: for each variable x_i, choose the edge (S, X_i) iff x_i is true, otherwise choose the edge $(S, \overline{X}_i)$; for each clause c_j, choose the edge (X_i, C_j) (or $(\overline{X}_i, C_j)$), where x_i (or $\overline{x}_i$) is a variable (or negated variable) satisfying c_j.

It suffices to choose A, B, D, E and W so that

$$0 < A < B < W$$

$$D + A + E \leq \alpha \min\{A + D, B + W\}$$

$$D + A + B \leq \alpha \min\{D + A + B, W\} < D + A + E + B.$$

To achieve this, let $A = 1$, $B = \alpha$, $D = 2\alpha$, $E = (\alpha - 1)(2\alpha + 1)$ and $W = 1 + 2\alpha + \frac{1}{\alpha}$. □

Next we will reduce the $(\alpha, 1)$-LAST problem to the (α, β)-LAST problem, for any fixed α and β such that $\alpha > 1$ and $1 \leq \beta < 1 + \frac{2}{\alpha - 1}$.

Proof. (Theorem 2) Let G^* be the graph for which we want to determine the existence of an $(\alpha, 1)$-LAST rooted at a given vertex r^*. By Lemma 4.1, there exists a graph G' with no (α, β)-LAST rooted at some vertex r'. Assume without loss of generality that the minimum spanning tree of G^* has weight 1 and the minimum spanning tree of G' is of weight c (a constant to be determined later). Define the graph G to be the union of G^* and G' by identifying r^* and r' into a single root r.

Let β' be the minimum β such that G' has an (α, β)-LAST. Define β^* analogously for G^*. Take $c = \frac{\beta - 1}{\beta' - \beta}$.

The weight of the minimum spanning tree in G is $1 + c$; similarly the lightest tree in G meeting the distance requirement is of weight $\beta^* + \beta' c$. Thus G has an (α, β)-LAST iff $\beta^* + \beta' c \leq \beta(1 + c)$. By our choice of c, this is equivalent to $\beta^* \leq 1$. Thus G has an (α, β)-LAST iff G^* has an $(\alpha, 1)$-LAST. □

5 Minimum-Weight Shortest-Path Trees

Next we consider the case when $\alpha = 1$, i.e., an (α, β)-LAST is a shortest-path tree of weight at most β times the weight of the minimum spanning tree. In this case, no algorithm can guarantee any fixed β for all graphs. Instead, we show how to find a $(1, \beta)$-LAST with minimum β in a given graph, i.e., a minimum-weight shortest-path tree.

In fact, we solve a more general problem: finding a minimum-weight shortest-path tree in a rooted *directed* graph. The undirected case reduces to this case by the standard trick of replacing each undirected edge (u, v) by two new directed edges (u, v) and (v, u) of the same weight as the original edge.

The directed problem reduces in turn to the problem of finding a *minimum-weight branching* in the *shortest-path subgraph* of the given directed graph. A branching is a directed spanning tree with all edges directed away from the root. The shortest-path subgraph is the spanning subgraph consisting of all directed edges (u, v) on some shortest path from the root, i.e., those for which $D_G(r, u) + w(u, v) = D_G(r, v)$. It is easy to show that the shortest-path trees in a directed graph are exactly the branchings from the root in its shortest-path subgraph. Consequently, it suffices to find a minimum-weight branching in the shortest-path subgraph.

A polynomial-time algorithm for finding a minimum-weight branching in *any* given graph is known [13]. However, a shortest-path subgraph of a non-negatively weighted graph has the property that any edge on a cycle has weight zero. This allows the following linear-time algorithm. First, identify the strongly-connected components in the subgraph induced by the edges of weight zero. This can be done in linear time [10]. For each component not containing the root, choose the minimum-weight incoming edge and call the vertex with an incoming chosen edge the *base* vertex of the component. For the component containing the root, call the root vertex the base vertex. For each component, find a branching of weight zero edges rooted at the base in the subgraph induced by the component. Finally, return the chosen edges together with the edges of the components' branchings.

This set of edges forms a branching: each non-root vertex has an incoming edge and there are no cycles. The branching is of minimum weight because in *any* branching every non-root component has at least one incoming edge. It is straightforward to implement the algorithm to run in $O(n + m)$ time. This proves Theorem 3.

6 Finding LAST's in Parallel

Given $\alpha > 1$, a minimum spanning tree, and a shortest-path tree, an $(\alpha, 1 + \frac{2}{\alpha-1})$-LAST can be found using n processors in $O(\log n)$-time. The model of computation we use is the *Concurrent-Read, Exclusive-Write Parallel RAM*, in which independent, synchronized parallel processors share a common memory [14]. Multiple simultaneous accesses to the same memory location are allowed only if all of the accesses are read operations.

The algorithm is as follows. Let $C = (e_1, e_2, ..., e_{2n-2})$ be the (directed) edges of the walk through the graph implicit in the depth-first search of the minimum spanning tree, as in Section 3.2. This tour can be constructed in $O(\log n)$ time by n processors using standard techniques [14]. Let $(u_i, u_{i+1}) = e_i$. Using the terminology of Section 3.2, after edge (u_i, u_{i+1}) is traversed from u_i to u_{i+1}, vertex u_{i+1} is the current vertex.

The parallel algorithm emulates the serial algorithm except that the distance estimates are looser in two ways. First, while a vertex may occur several times in C, the parallel algorithm treats each occurrence as a distinct vertex. Second, when a shortest path is added, only the distance estimate of the destination vertex is lowered. These looser distance estimates can be computed in parallel, but still suffice to imply the weight requirement.

Let $D_C(u_i, u_j)$ denote $\sum_{k=i}^{j-1} w(e_k)$, the distance from u_i to u_j along C. Let $m(i,j)$ be the relation

$$i < j \ \text{ and } \ D_{T_S}(r, u_i) + D_C(u_i, u_j) > \alpha\, D_{T_S}(r, u_j).$$

The meaning of $m(i,j)$ is the following. Suppose we modify the original algorithm to use the looser distance estimates. When the modified algorithm enountered a vertex u_j, if the algorithm had not added shortest paths to any of the vertices $u_{i+1}, u_{i+2}, ..., u_{j-1}$, then it would add the shortest path from the root to u_j. Thus, if the modified algorithm adds a shortest path to a vertex u_i, then the next shortest path it adds will be to vertex u_k, where $k = \min\{j : m(i,j)\}$.

The parallel algorithm will emulate the modified algorithm. Define $J(i) = \min\{j : m(i,j)\}$. The parallel algorithm will compute the function J and then add shortest paths to vertices in the set

$$S = \{u_1, u_{J(1)}, u_{J(J(1))}, ..., u_{J(J(\ldots J(1)\ldots))}\}.$$

Once J has been computed, S can be computed by n processors in $O(\log n)$ times on a CREW PRAM using standard techniques [14]. Once S has been computed, the set S^* of ancestors of S in the shortest-path tree can also be computed by n processors in $O(\log n)$ time using tree contraction techniques [14]. The final tree is formed by each non-root vertex choosing as its parent either the parent in the shortest-path tree (if the vertex is in S^*) or the parent in the minimum spanning tree (otherwise). It can easily be shown that every vertex has a path to the root using this set of $n-1$ edges, so that they do indeed form a tree.

It remains to compute $J(i)$. First, note that $m(i,j)$ is monotone in i for fixed j.

Lemma 6.1 *If $i' < i$ and $m(i,j)$ is true then $m(i',j)$ is true.*

Proof. If $m(i,j)$ is true then $D_{T_S}(r,u_i) + D_C(u_i,u_j) > \alpha\, D_{T_S}(r,u_j)$. For any $i' < i$, $D_C(u_i,u_j) \le D_C(u'_i,u_j)$. The shortest path from r to i is no longer than any other path from i to r in the graph, and hence $D_{T_S}(r,u_i) \le D_{T_S}(r,u_{i'}) + D_C(u'_i,u_i)$. Combining these inequalities, we get

$$\begin{aligned} D_{T_S}(r,u_{i'}) + D_C(u'_i,u_j) &= D_{T_S}(r,u_{i'}) + D_C(u'_i,u_i) + D_C(u_i,u_j) \\ &\ge D_{T_S}(r,u_i) + D_C(u_i,u_j) \\ &> \alpha\, D_{T_S}(r,u_j) \end{aligned}$$

Hence $m(i',j)$ is true by definition. □

The function J can be computed efficiently because of this monotonicity property:

Lemma 6.2 *Suppose $m(i,j)$ implies $m(i',j)$ for $0 \le i' \le i \le n$, $0 \le j \le n$. Then the function $J(i) = \min\{j : m(i,j)\}$ can be computed in $O(\log n)$ time by n processors on a CREW PRAM.*

Proof. Define $I(j) = \max\{i : m(i,j)\}$. For each j, compute $I(j)$ using binary search. Define $I^*(j) = \max\{I(j') : 1 \le j' \le j\}$. Compute function I^* from function I using a standard prefix-maxima computation. Finally, define $J'(i) = \min\{j : I^*(j) \ge i\}$. Compute function J', again using binary search, from monotone function I^*. (See Figure 6.) Each of these computations can be done by n processors in $O(\log n)$ time on a CREW PRAM using standard techniques [14].

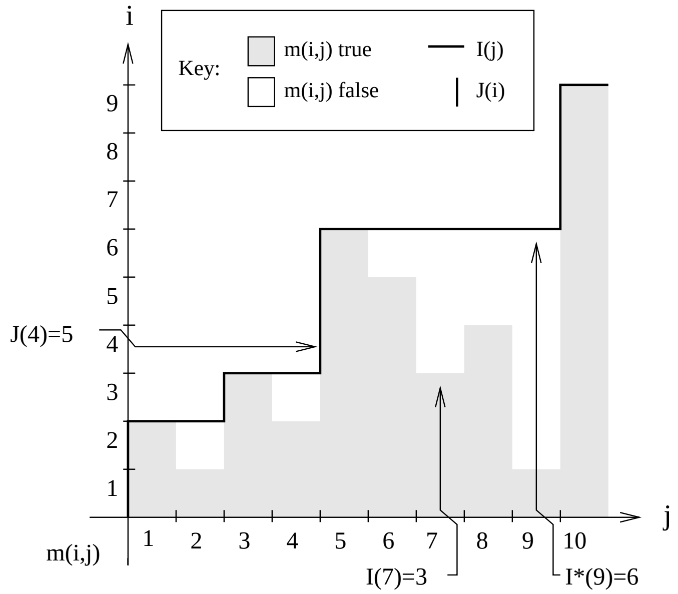

Figure 6: Computing J from m to find a LAST in parallel.

We prove that $J'(i) = J(i)$ for each i. The proof is in two steps.

1. $J'(i) = \min\{j : I^*(j) \ge i\} = \min\{j : I(j) \ge i\}$, i.e., the smallest j such that $I^*(j)$ exceeds i is equal to the smallest j such that $I(j)$ exceeds i. This is because the latter depends only on the maxima of I — those j such that $I(j) > I(j')$ for all $j' \le j$.

2. $\min\{j : I(j) \ge i\} = \min\{j : m(i,j)\}$ because $I(j) \ge i$ is equivalent to $m(i,j)$ by the monotonicity property of m.

□

The analyses of Lemmas 3.1 and 3.2 can easily be adapted to prove that the final tree produced by the parallel algorithm is an $(\alpha, 1 + \frac{2}{\alpha-1})$-LAST. This establishes Theorem 4 — an $(\alpha, 1 + \frac{2}{\alpha-1})$-LAST can be computed by n processors in $O(\log n)$ time on a CREW PRAM.

7 Conclusions

Every graph contains trees that offer a continuous tradeoff between minimum spanning trees and shortest-path trees. Trees achieving the optimal trade-off can be found in (sequential) linear time or in logarithmic time by linearly many processors.

In the following cases, is it possible to obtain a better trade-off?

- In Euclidean graphs (i.e., points in the plane). Note that the proof of Lemma 3.2 requires only that the algorithm walk around the graph from the root visiting every vertex once, i.e., that the algorithm traverse a Traveling Salesman path starting at the root. In Euclidean graphs, perhaps such a path of weight at most $(2-\epsilon)$ times the minimum spanning-tree weight always exists and can be found in polynomial time.
- If the distance requirement is replaced by the requirement that the *sum* of distances from the root is within α times the minimum possible?
- If the root is not fixed? This would correspond to the problem of installing a low-cost network *and* choosing a root site so that distances from the root are near-minimum.

Clearly any (α, β)-LAST also meets these looser requirements, but our lower bounds no longer show that the trade-off is optimal.

For directed graphs, it is easy to show that for any fixed α and β, (α, β)-LAST's may not exist and that finding the minimum β such that an (α, β)-LAST exists is NP-hard. Can one *approximate* this minimum β?

Acknowledgments

We would also like to thank Seffi Naor, Dheeraj Sanghi and Moti Yung for useful discussions. We would like to thank Shay Kutten for telling us about [2]. We would like to thank Baruch Awerbuch, Alan Baratz and David Peleg for sending us a copy of their manuscript [3]. We would like to thank Andrew Kahng and Jeff Salowe for telling us about [7, 8, 9].

References

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

[2] B. Awerbuch, A. Baratz, and D. Peleg, Cost-sensitive analysis of communication protocols, Proc. of 9th Symp. on Principles of Distributed Computing (PODC), pp. 177–187, (1990).

[3] B. Awerbuch, A. Baratz, and D. Peleg, Efficient broadcast and light-weight spanners, Manuscript, (1991).

[4] K. Bharath-Kumar and J. M. Jaffe, Routing to multiple destinations in computer networks, *IEEE Transactions on Communications* 31 (3), pp. 343–351, (1983).

[5] B. Chandra, G. Das, G. Narasimhan and J. Soares, New sparseness results on graph spanners, Proc. of 8th Symp. on Computational Geometry, (CG), pp. 192–201, (1992).

[6] L. P. Chew, There are planar graphs almost as good as the complete graph, *Journal of Computer and System Sciences*, 39(2), pp. 205–219, (1989).

[7] J. Cong, A. B. Kahng, G. Robins, M. Sarrafzadeh and C. K. Wong, Performance-driven global routing for cell based IC's, Proc. IEEE Intl. Conference on Computer Design, pp. 170–173, (1991).

[8] J. Cong, A. B. Kahng, G. Robins, M. Sarrafzadeh and C. K. Wong, Provably good performance-driven global routing, *IEEE Transactions on CAD*, pp. 739-752, (1992).

[9] J. Cong, A. B. Kahng, G. Robins, M. Sarrafzadeh and C. K. Wong, Provably good algorithms for performance-driven global routing, *Proc. IEEE Intl. Symp. on Circuits and Systems*, San Diego, pp. 2240-2243 (1992).

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, *The MIT Press*, (1989).

[11] E. W. Dijkstra, A note on two problems in connexion with graphs, *Numerische Mathematik*, 1, pp. 269–271 (1959).

[12] M. L. Fredman and R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *Journal of the ACM*, 34 (3), pp. 596–615, (1987).

[13] H. N. Gabow, Z. Galil, T. Spencer and R. E. Tarjan, Efficient algorithms for finding minimum spanning trees in undirected and directed graphs, *Combinatorica*, 6 (2), pp. 109–122, (1986).

[14] J. JáJá, Introduction to Parallel Algorithms, *Addison-Wesley*, Reading, MA, (1991).

[15] J. B. Kruskal, On the shortest spanning subtree of a graph and the traveling salesman problem, *Proc. Amer. Math. Soc.*, 7, pp. 48–50, (1956).

[16] 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 (3), pp. 251–256, (1992).

[17] D. Peleg and J. D. Ullman, An optimal synchronizer for the hypercube, Proc. of 6th Symp. on Principles of Distributed Computing (PODC), pp. 77–85, (1987).

[18] F. P. Preparata and M. I. Shamos, Computational Geometry, *Springer Verlag*, (1985).

[19] R. C. Prim, Shortest Connection Networks and Some Generalizations, *Bell System Tech. J.*, 36, pp. 1389–1401 (1957).

[20] P. M. Vaidya, A sparse graph almost as good as the complete graph on points in K dimensions, *Discrete and Computational Geometry*, 6, pp. 369–381, (1991).