# A 3-approximation for the minimum tree spanning $k$ vertices

*Naveen Garg*
Max-Planck-Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany

## Abstract

*In this paper we give a 3-approximation algorithm for the problem of finding a minimum tree spanning any $k$-vertices in a graph. Our algorithm extends to a 3-approximation algorithm for the minimum tour that visits any $k$-vertices.*

## 1. Introduction

Given an undirected graph $G = (V, E)$ with edge costs $c : E \rightarrow \mathbf{R}^+$ we consider the problem of finding a minimum tree spanning $k$ vertices; we call such a tree a $k$-tree. The problem is known to be **NP**-hard. It has received much attention in recent years with a sequence of papers improving the approximation guarantee from an initial $3\sqrt{k}$ [11] to $O(\log^2 k)$ [1] to $O(\log k)$ [10] and finally to 35 [4]. In this paper we provide a 3-approximation algorithm to this problem.

For the special case when we are given $n$ points in the plane and the cost of an edge is just the Euclidean distance between its end-points, it was believed that the minimum $k$-tree was easier to approximate. Here too a sequence of results improved the approximation guarantee from $O(k^{1/4})$ [11] to $O(\log k)$ [6] to $O(\log k / \log\log n)$ [5] to $O(1)$ [3] and finally to $2\sqrt{2}$ [9]. Our approximation factor for the general case compares well with the best known for this seemingly simpler setting.

An important technical contribution of this paper is that it makes explicit the lower bounds on the optimum value used in arriving at the claimed approximation guarantee. We use the maximum of two lower bounds as our lower bound. One of these is the distance of the farthest vertex from the root. While this is not really a lower bound on the optimum value it is a lower bound if we assume that the farthest vertex from the root is included in the minimum $k$-tree; this it turns out is easy to ensure. The problem of computing the minimum $k$-tree can be formulated as an integer program. Relaxing the integrality constraints then gives us a linear program the optimum value to which is a lower bound on the cost of the minimum $k$-tree. Our second lower bound relates to the optimum value of this linear relaxation. We are not aware of any other research that explores the realtionship between the optimum value of this LP-relaxation and the optimum integral solution. These two lower bounds independently might be significantly away from the optimum. We show however that the cost of the minimum $k$-tree is no more than three times the maximum of these lower bounds. We also show an instance for which this is tight *ie.* the cost of the minimum $k$-tree is almost three times the value of our lower bound. This implies that there are no improvements possible to the 3-approximation guarantee using only these lower bounds.

A problem similar to the minimum $k$-tree problem is that of finding *a tour that visits at least $k$ vertices and has minimum cost*; we call this the minimum $k$-tour. A bounded approximation guarantee for this problem can be obtained only under the assumption that the edge-costs satisfy triangle inequality. Our approximation algorithm for the minimum $k$-tree yields a 3-approximation algorithm for this problem. This in turn allows us to obtain a better approximation guarantee for finding *a tour that minimizes the total latency.* Once again we are given an edge weighted graph and a root $r$ and we wish to find a tour that visits all the vertices in an order such that the sum of the latencies of the vertices is minimized where the latency of a vertex is its distance from the root in the tour chosen. Blum et.al. [2] first gave an algorithm with an approximation guarantee of 144; this was subsequently improved to 21.55 [7]. In [7] it is also shown how an $\alpha$-approximation algorithm for the minimum $k$-tour problem can be used to obtain a $3.55\alpha$-approximation algorithm for the minimum latency problem. Our 3-approximation for the minimum $k$-tour therefore gives a 10.77-approximation algorithm for minimizing latency.

The paper is organized as follows. In Section 2 we discuss the two lower bounds used. We then show how these lower bounds can be used to obtain a simple 5-approximation algorithm for the minimum $k$-tree problem (Section 3). We then develop two techniques to pick the vertices in a more careful way; this yields the 3-approximation algorithm (Section 4). In Section 5 we give the tight example that shows that the gap between the value of the lower



0272-5428/96 $05.00 © 1996 IEEE

bound used and the cost of the minimum $k$-tree can be arbitrarily close to 3. Finally, in Section 6 we show how to obtain a $k$-tour of cost within thrice the optimum.

## 2. The Lower Bounds

In this paper we actually solve the rooted version of the minimum $k$-tree problem. In this version of the problem we are given a *root* vertex $r \in V$ and we wish to find the minimum $k$-tree including the root. A solution to this problem also gives a solution to the original one since we can run this algorithm for all possible choices of the root and pick the minimum $k$-tree.

We also assume that the distance of the farthest vertex from the root is a lower bound on the optimum value. Since this is not true for the original graph we work on the subgraph that includes only such vertices whose distance from the root is less than the distance of the farthest vertex (from the root) in the minimum $k$-tree. The lack of knowledge of this "farthest vertex" in the optimum solution can be gotten around by trying all possible vertices and picking the minimum solution.

We associate a variable $x_e$ with the edge $e$; $x_e = 1$ implies that the edge belongs to the $k$-tree. We also associate a variable $x_v$ with the vertex $v$ and $x_v = 1$ implies that the $k$-tree spans $v$; thus $\sum_{v \in V} x_v = k$. For every subset of vertices not containing the root, the number of edges with $x_e = 1$ that have exactly one end-point in the set should be at least as large as $x_v$ for any vertex $v$ in the set. More formally, let $\nabla(S)$ be the set of edges with exactly one end-point in $S$. Then for any set $S \subseteq \{V - r\}$ and vertex $v \in S$ we have $\sum_{e \in \nabla(S)} x_e \geq x_v$. The LP-relaxation of this integer program is obtained by replacing the 0/1 constraint on the variables $x_e, x_v$ by $0 \leq x_e, x_v \leq 1$. Further, the constraint $x_v \leq 1$ implies that no optimum solution would have $x_e > 1$ for any edge $e \in E$ and hence we can dispense with this constraint on $x_e$. This gives us the following linear program

$$\begin{array}{rcll} \text{minimize} & & \sum_{e \in E} x_e c_e & \\ \text{subject to} & & & \\ \sum_{e \in \nabla(S)} x_e & \geq & x_v & (\forall v, S : v \in S \subseteq \{V - r\}) \\ \sum_{v \in V} x_v & = & k & () \\ x_v & \leq & 1 & (\forall v \in V) \\ x_v & \geq & 0 & (\forall v \in V) \\ x_e & \geq & 0 & (\forall e \in E) \end{array}$$

The dual of this linear program has a variable $y_{v,S}$ for all $v, S : v \in S \subseteq \{V - r\}$ and a variable $p_v$ for vertex $v$ and a variable $p$.

$$\begin{array}{rcll} \text{maximize} & & p \cdot k - \sum_{v \in V} p_v & \\ \text{subject to} & & & \\ \sum_{S : v \in S} y_{v,S} + p_v & \geq & p & (\forall v \in V) \\ \sum_{S : e \in \nabla(S)} y_{v,S} & \leq & c_e & (\forall e \in E) \\ p_v & \geq & 0 & (\forall v \in V) \\ y_{v,S} & \geq & 0 & (\forall v, S : v \in S \subseteq \{V - r\}) \end{array}$$

Consider an optimum solution to the dual linear program. For vertex $v \in V$ we define the *potential* of $v$ as $\alpha_v = \sum_{S : v \in S} y_{v,S}$. Let $\alpha$ be the $k^{\text{th}}$ smallest potential.

**Claim 2.1** *$p$ has a value between the $k^{\text{th}}$ and the $(k+1)^{\text{th}}$ smallest potentials. The optimum value of the dual program is the sum of the $k$ smallest potentials.*

Proof: First note that if $\alpha_v \geq p$ then $p_v = 0$ else $p_v = p - \alpha_v$. Now for contradiction assume that $p$ is strictly larger than the $(k+1)^{\text{th}}$ smallest potential. Then at least $k+1$ vertices have a positive value of $p_v$. Decreasing $p$ by a small amount $\epsilon$ allows us to decrease every positive $p_v$ by $\epsilon$. This increases the objective function value by at least $\epsilon > 0$ contradicting our assumption that the solution is optimal. On the other hand if $p$ is strictly smaller than the $k^{\text{th}}$ smallest potential then for at most $k-1$ vertices $p - \alpha_v \geq 0$; for the others $p - \alpha_v < 0$. We increase $p$ by a sufficiently small amount $\epsilon$ so that the only $p_v$ values we have to modify correspond to the vertices for which $p - \alpha_v \geq 0$. Since these are at most $k-1$ vertices whose $p_v$ value has to be increased by $\epsilon$ each, the net increase in the objective function value is at least $\epsilon > 0$. Thus $p$ lies between the $k^{\text{th}}$ and the $(k+1)^{\text{th}}$ smallest potentials.

Only the vertices with the $k$ smallest potentials have a positive value for $p_v$. Since for these vertices $\alpha_v = p - p_v$, the value of the objective function is the sum of the $k$ smallest potentials. ■

The function $\alpha : V \rightarrow \mathbf{R}^+$ which gives the potentials of the vertices was defined as $\alpha_v = \sum_{S : v \in S} y_{v,S}$ where $y_{v,S}$ is an assignment of non-negative values to pairs $(v, S)$, $v \in S \subseteq \{V - r\}$ such that for every edge $e$, $\sum_{e \in \nabla(S)} y_{v,S} \leq c_e$.

**Definition 2.1** *An assignment of potentials to vertices, $\pi : V \rightarrow \mathbf{R}^+$, is* feasible *if there exists an assignment of non-negative values to pairs $(v, S)$, $v \in S \subseteq \{V - r\}$ such that for any edge $e$, $\sum_{e \in \nabla(S)} y_{v,S} \leq c_e$ and for any vertex $v$, $\pi(v) \leq \sum_{S : v \in S} y_{v,S}$.*

The dual program can now be interpreted as finding a feasible assignment of potentials to vertices such that the sum of the $k$ smallest potentials is maximized.

The condition that a potential function should satisfy to be feasible is somewhat unwieldy. Consider instead the following condition for feasibility.

**Definition 2.2** *A potential assignment is* feasible *if any tree rooted at $r$ has cost at least as large as the sum of the potentials of the vertices it spans.*

It is immediate that in any potential assignment with the above property the sum of the $k$ smallest potentials is a lower bound on the cost of the minimum $k$-tree rooted at $r$. However it is **NP**-complete to determine if a given assignment of potentials is infeasible under this new definition. On the other hand checking for the infeasibility of a potential assignment under the older definition is easy and the older definition can in fact be viewed as a "fractional" relaxation of the new definition and hence the following claim.

**Claim 2.2** *If $\pi : V \rightarrow \mathbf{R}^+$ is a feasible potential assignment under the older definition then it is also feasible under the new definition.*

Note that the converse to the above claim is not true since we can have an infeasible assignment of potentials to vertices such that any tree rooted at $r$ has cost at least as large as the sum of the potentials of the vertices it spans. Henceforth we work with the new definition of feasibility of a potential assignment and use the sum of the $k$ lowest potentials in such a feasible potential assignment as a lower bound on the optimum value.

## 3. The Basic Algorithm

The algorithm described in this section is the same as in [4] and [8]; we present it only for the sake of completeness. Each subset of vertices not containing the root, $S \subset V, r \notin S$, has a variable $y_S$ associated with it; $y_S$ is initially zero for all subsets $S$. Our assignment of values to these variables would be such that they form a packing *ie.* $\forall e \in E, \sum_{S:e \in \nabla(S)} y_S \leq c_e$. An edge is *tight* if the above inequality is satisfied as an equality for that edge.

At any step in the algorithm the vertices are partitioned into a collection of *active* and *inactive components* each of which has a non-negative *potential* associated with it. A component is active if and only if it has a positive potential and does not contain the root. To begin with, every vertex besides the root is assigned a potential $p > 0$. Thus at the first step all vertices form active components. We raise the variables corresponding to the active components uniformly, simultaneously decreasing their potentials till either

1. an edge $e = (u, v)$ goes tight[1]. The two components containing vertices $u$ and $v$ are merged into one. This new component is assigned a potential equal to the sum of the residual potentials of the merged components. If this new component contains the root it is made inactive.
2. the potential of a component reduces to zero. The component is made inactive.

The process halts when all components are inactive.

The following observations follow immediately from the above description.

1. The component containing the root is always inactive and this is the only inactive component with a non-zero potential; all other inactive components have zero potential.
2. At any point of the algorithm the tight edges form a forest the trees of which define the components at that stage.

**Delete Phase.** Let $T_p$ be the tree of tight edges spanning the final component containing the root; the subscript $p$ denotes the fact that this component and the tree spanning it were obtained when every vertex was assigned an initial potential $p$. Viewing $T_p$ as a tree rooted at $r$ we associate with every edge $e \in T_p$ the subtree of $T_p$ rooted at the endpoint of $e$ farther from the root. We now delete from $T_p$ all those edges whose associated subtree formed an inactive component at some point in the algorithm. Note that when an edge $e$ is deleted the subtrees associated with the edges on the path from $e$ to the root get modified. Let $\hat{T}_p \subseteq T_p$ be the residual tree *ie.* the part of $T_p$ containing the root.

Let $\pi_p$ denote the assignment which gives every vertex besides the root a potential $p$. $\pi_p$ is not necessarily feasible and the following lemma gives a sufficient condition for the feasibility of $\pi_p$.

**Lemma 3.1** *An assignment $\pi$, of potentials to the vertices is feasible if the root component has zero potential when the above algorithm is run with every vertex having an initial potential given by $\pi$.*

Proof: Omitted ■

We now show how to reduce the potentials on the vertices to obtain a feasible potential assignment $\hat{\pi}_p$ while ensuring that the behavior of the above algorithm when the initial assignment of potentials is given by $\hat{\pi}_p$ is exactly the same as when all vertices have an initial potential $p$.

Note that the only vertices whose potential can be decreased are those for which every component containing the vertex has non-zero residual potential. We decrease the potential of all these vertices uniformly till some component, which earlier had a non-zero potential, has a zero potential; the potential of the vertices in this component cannot be decreased further. We continue in this manner till there is no vertex left whose potential can be decreased further; $\hat{\pi}_p$ denotes this final potential assignment.

From our procedure for decreasing potentials it follows that the behavior of the algorithm on $\hat{\pi}_p$ is the same as that

[1] If many edges go tight simultaneously then we consider them in lexicographic order.

on $\pi_p$. Further, the maximum potential of a vertex in the assignment $\hat{\pi}_p$ is $p$ and the vertices not in $\hat{T}_p$ have this potential.

**Lemma 3.2** *The potential assignment $\hat{\pi}_p$ is feasible.*

Proof: We show that the potential on $T_p$ is zero; this by Lemma 3.1 implies that $\hat{\pi}_p$ is feasible. For contradiction assume that $T_p$ had positive potential. Then one of the two components which merged to form $T_p$ also has positive potential. Repeating the argument on this component we finally obtain a vertex such that every component that this vertex belongs to has positive potential. Therefore the potential of this vertex can be reduced, contradicting our assumption on $\hat{\pi}_p$. ■

**Theorem 3.3**

$$cost(\hat{T}_p) \leq 2 \cdot \sum_{v \in \hat{T}_p} \hat{\pi}_p(v)$$

Running the above two procedures gives us a feasible potential assignment, $\hat{\pi}_p$, and a tree $\hat{T}_p$ such that $\text{cost}(\hat{T}_p) \leq 2 \cdot \sum_{v \in \hat{T}_p} \hat{\pi}_p(v)$. Further, the potential of any vertex under the assignment $\hat{\pi}_p$ is no more than $p$.

## 4. A 5-approximation Algorithm

Let $q$ be the largest value for which $\left|\hat{T}_q\right| \leq k$; thus $\left|\hat{T}_q\right| = k_1 \leq k < \left|\hat{T}_{q+\epsilon}\right| = k_2$. The tree $\hat{T}_q$ spans only $k_1$ vertices and hence we need to pick an additional $k - k_1$ vertices. Let $X = \hat{T}_{q+\epsilon} - \hat{T}_q$ be the set of vertices that are not in $\hat{T}_q$ but are there in $\hat{T}_{q+\epsilon}$. Clearly $|X| \geq k_2 - k_1$ and we shall pick the additional vertices from this set. This we do as follows. We first short-circuit the tree $\hat{T}_{q+\epsilon}$ into a cycle that has cost at most $2\text{cost}(\hat{T}_{q+\epsilon})$ and includes all the vertices of $X$. We then pick the least cost segment of $k - k_1$ vertices from the cycle; the cost of this segment is no more than $(k - k_1)/(k_2 - k_1)$ times the cost of the cycle. This yields us two connected components that span $k$ vertices in all and these can be joined by picking the cheapest edge between them. Also note that the tree $\hat{T}_{q+\epsilon}$ is a solution since it spans $k_2 > k$ vertices.

We have thus obtained two different solutions; the cost of the first solution can be bounded by

$$\text{cost}(\hat{T}_q) + \frac{k - k_1}{k_2 - k_1} \cdot 2 \cdot \text{cost}(\hat{T}_{q+\epsilon}) + \text{OPT}$$

and that of the second by $\text{cost}(\hat{T}_{q+\epsilon})$. The sum of the $k$ lowest potentials in the two feasible potential assignments $\hat{\pi}_q$ and $\hat{\pi}_{q+\epsilon}$ provide lower bounds on the cost of the minimum $k$-tree and hence

$$\begin{aligned} \text{OPT} &\geq \sum_{v \in \hat{T}_q} \hat{\pi}_q(v) + q \cdot (k - k_1) \\ \text{OPT} &\geq \sum_{v \in \hat{T}_{q+\epsilon}} \hat{\pi}_{q+\epsilon}(v) - (q + \epsilon) \cdot (k_2 - k_1) \end{aligned}$$

Since $\text{cost}(\hat{T}_q) \leq 2\sum_{v \in \hat{T}_q} \hat{\pi}_q(v)$ and $\text{cost}(\hat{T}_{q+\epsilon}) \leq 2\sum_{v \in \hat{T}_{q+\epsilon}} \hat{\pi}_{q+\epsilon}(v)$ we have

$$\begin{aligned} \text{cost}(\hat{T}_q) &\leq 2(\text{OPT} - (k - k_1) \cdot q) \\ \text{cost}(\hat{T}_{q+\epsilon}) &\leq 2(\text{OPT} + (k_2 - k) \cdot (q + \epsilon)) \end{aligned}$$

It is easily checked that one of the two solutions has cost no more than $5 \cdot \text{OPT}$.

## 5. A 3-approximation Algorithm

One of the shortcomings of the 5-approximation algorithm is that the $(k - k_1)$ vertices of $X$ are picked at a very high cost. We did not consider the distribution of the vertices of $X$ in the tree $\hat{T}_{q+\epsilon}$; such an argument could perhaps give us a better way for picking the additional subset of vertices. We now present an algorithm that achieves an approximation guarantee of 3. The algorithm proceeds by first finding two trees $\hat{T}_-$ and $\hat{T}_+$ such that $\left|\hat{T}_-\right| < k$ and $\left|\hat{T}_+\right| \geq k$. Additionaly these trees have a very similar structure. We then give a procedure for picking a subset of the vertices that are in $\hat{T}_+$ but not in $\hat{T}_-$ so that together with the vertices of $\hat{T}_-$ this forms a set of $k$ vertices.

### 5.1. Finding similar trees

*Event points* are values of the initial potential at which the tree returned by the first phase of the algorithm changes. Let $p+, p-$, be potentials that are infinitesimally larger and smaller than $p$ respectively. Thus $p$ is an event point if $T_{p-} \neq T_{p+}$. Let $p_1, p_2, \ldots p_i, \ldots$ be the event points. Note that the tree returned by our algorithm changes only at the event points, between event points when the potential varies, the trees remain the same *ie.*, $T_{p_{i-1}+} = T_{p_i-}$. However the tree $\hat{T}_p$ which is obtained from $T_p$ by deleting subtrees that correspond to inactive components might change at potential values that are between event points, *ie.*, $\hat{T}_{p_{i-1}+}$ might be different from $\hat{T}_{p_i-}$.

Let $i$ be such that $\left|\hat{T}_{p_{i-1}+}\right| < k$ and $\left|\hat{T}_{p_i+}\right| \geq k$. Then either $\left|\hat{T}_{p_{i-1}+}\right| < k$ and $\left|\hat{T}_{p_i-}\right| \geq k$ or $\left|\hat{T}_{p_i-}\right| < k$ and $\left|\hat{T}_{p_i+}\right| \geq k$. In the former case define $q$ to be that value

of potential between $p_{i-1}$ and $p_i$ such that $\hat{T}_{q-} < k$ and $\hat{T}_{q+} \geq k$ while in the latter case define $q$ to be the potential $p_i$. It follows from our choice of the potential $q$ that $\hat{T}_{q-} < k$ and $\hat{T}_{q+} \geq k$. The following claim relates the structure of trees $T_{q-}$ and $T_{q+}$.

**Claim 5.1** *Let $S$ be a set of vertices such that $y_S > 0$ when we run our algorithm with an initial potential $q$. Then the vertices of $S$ occur contiguously in the trees $T_{q-}$ and $T_{q+}$.*

Proof: Since the initial potentials $q-$ and $q+$ are only infinitesimally different from $q$ any set which has a positive potential when the algorithm is run with an initial potential $q$ would continue to have a positive potential when the initial potentials are $q-$ or $q+$. Hence the vertices of this set would occur contiguously in the trees $T_{q-}$ and $T_{q+}$. ■

We now show a sequence of steps by which tree $T_{q+}$ can be obtained from $T_{q-}$. Let $S$ be a set of vertices such that $y_S > 0$ when we run our algorithm with an initial potential $q$. Since the vertices of $S$ occur contiguously in trees $T_{q-}$ and $T_{q+}$ we can replace the edges induced by $S$ in $T_{q-}$ with the edges that this set induces in $T_{q+}$ without increasing the cost of the tree. We perform these modifications on the active sets $S$ in the order that they were created. This gives us a sequence of trees beginning with $T_{q-}$ and ending with $T_{q+}$. Since $\left|\hat{T}_{q-}\right| < k$ and $\left|\hat{T}_{q+}\right| \geq k$ there exist two trees $T_-$ and $T_+$ in the sequence such that $\left|\hat{T}_-\right| < k$ and $\left|\hat{T}_+\right| \geq k$. Furthermore by our procedure for swapping edges it follows that there exists an active set $S$ formed from components $S_1, S_2, \ldots$ such that the trees $T_-, T_+$ differ only in the edges between these components. We can further narrow the difference between the trees $T_-$ and $T_+$ by replacing the edges that run between these components in the tree $T_-$ with the edges in $T_+$ one at a time. We remove an edge of $T_-$ and add that edge of $T_+$ that coonects the two components formed. Redefine $T_-, T_+$ to be the trees differing in an edge and satisfying $\left|\hat{T}_-\right| < k, \left|\hat{T}_+\right| \geq k$. Note that although trees $T_-, T_+$ differ in only one edge, the trees $\hat{T}_-$ and $\hat{T}_+$ could be quite different.

Let $e_- \in T_-, e_+ \in T_+$ be the pair of edges that trees $T_-, T_+$ differ in and let $S_1, S_2$ be the two components of $S$ between which these edges run with $S_1$ being the component closer to the root.

## 5.2. Picking suitable vertices

Let $k_1, k_2$ be the number of vertices in trees $\hat{T}_-, \hat{T}_+$ respectively. Since $k_2 \geq k > k_1$, tree $\hat{T}_+$ has some vertices that do not belong to the tree $\hat{T}_-$; we call these vertices *new vertices*. The remaining vertices in $\hat{T}_+$ (which also belong to the tree $\hat{T}_-$) are the *old vertices*. Let $s$ be the number of new vertices; clearly, $s \geq k_2 - k_1$. Since the new vertices do not belong to $\hat{T}_-$ they have a potential $q$.

**Lemma 5.1** *In the tree $\hat{T}_+$ all new vertices occur contiguously while the old vertices form at most two contiguous sets.*

Proof: Omitted ■

Let $X_1, X_2$ be the two contiguous sets of old vertices; the set $X_1$ contains the root while $X_2$ is possibly empty. All new vertices belong to the set $S$. We shall pick a set $Y_1 \cup Y_2$ of $k-k_2+s$ new vertices which together with the $k_2-s$ old vertices, $X_1 \cup X_2$, in $\hat{T}_+$ forms the set, $Z = X_1 \cup X_2 \cup Y_1 \cup Y_2$, of $k$ vertices. The new vertices we pick are such that $Y_1$ occurs contiguously with $X_1$ and $Y_2$ with $X_2$ in the tree $\hat{T}_+$. Thus the edges of $\hat{T}_+$ induce two trees on the vertices of $Z$; let $T_1$ be the tree induced over $X_1 \cup Y_1$ and $T_2$ the tree induced over $X_2 \cup Y_2$. We shall later argue that the total cost of these two trees is at most twice the sum of the potentials of the vertices in $Z$ (Lemma 5.3). Since all new vertices have potential $q$ (the maximum potential) and the old vertices have potential at most $q$, the vertices in $Z$ are the $k$ vertices with the lowest potentials. Hence the sum of their potentials is a lower bound on the optimum and so the total cost of trees $T_1$ and $T_2$ is at most twice the optimum. By picking the least weight edge between the sets $X_1 \cup Y_1$ and $X_2 \cup Y_2$ we get a tree with $k$ vertices and of cost no more than thrice the optimum.

The vertices of $S_1$ (resp. $S_2$) are contiguous with the vertices of $X_1$ (resp. $X_2$). Only if there are not enough new vertices in $S_1$ do we pick some from $S_2$. Thus either it is the case that $Y_2 = \phi$ or $Y_1$ is the set of all new vertices in $S_1$. We henceforth assume that we are in the latter case since the other case is similar.

Let $T$ be the subtree of $\hat{T}_+$ induced over the vertices of $X_2 \cup S_2$ and $W$ be the set $S_2$. We are therefore trying to pick some number of new vertices from the set $W$; we assume that we have already picked all the old vertices in $T$. At any step in the following algorithm we have a tree $T$ and a contiguous subset $W$ of its vertices; $W$ can be viewed as a *window* of $T$. Only those new vertices that are in this window have not been picked yet; all other vertices of $T$ have been picked already. Our aim is to pick a certain number of new vertices from the window. One invariant that we shall maintain is that *all the picked vertices occur contiguously in $T$*. At each step we shall narrow the window $W$ till we reach a stage when the number of new vertices in $W$ is exactly the number we need.

By running our algorithm on the original graph we obtained certain active and inactive components at each iteration and also assigned dual variables to sets. At a particular step of the following procedure, we consider the restriction of these components to the vertices of $T$; these restrictions define the components in this step. The dual variable on a set, $S \subset T$, is the sum of the dual variables on the sets that

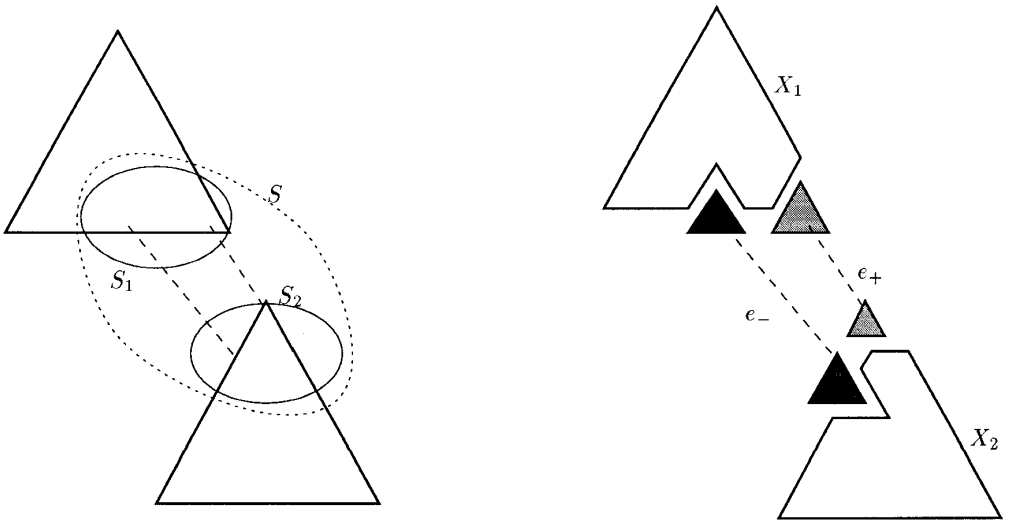


**Figure 1. The left tree is the tree $T_-/T_+$. On the right, the white part together with the part shaded gray (resp. black) is the tree $\hat{T}_+$ (resp. $\hat{T}_-$).**

when restricted to the vertices of $T$ yield the set $S$. We shall always maintain that *the window, $W$, is a component.* Note that this is true for the window defined for the first step.

**Definition 5.1** *A component $C$ pays for itself if* $\sum_{S \subseteq C} y_S \leq \sum_{v \in C} \hat{\pi}_q(v)$.

The third invariant we maintain is: *the components that do not pay for themselves are supersets of the window $W$.* Note that at the first step all components pay for themselves.

Consider the components at an iteration; the edges of $T$ form a tree over these components. A component is an *inactive leaf* if in some iteration it is inactive and a leaf of this tree. The final invariant we maintain is that *no inactive leaf is disjoint from $W$ or from any other inactive leaf.* At the first step let $v$ be the vertex of $T$ at which the edge $e_+$ is incident. Then every inactive leaf contains $v$ or else we would have deleted this component in obtaining $\hat{T}_+$ form $T_+$. Further since $v \in W$ the invariant holds. We are now ready to describe the actual algorithm for input $(W, T)$.

We first consider the case when $T$ has an inactive leaf, say $C$, that is a proper subset of $W$. If $W - C$ has enough new vertices then we move to the next step with arguments $(W - C, T - C)$ Else we move to the next step with arguments $(C, T)$.

If we are not in the above case then no proper subset of $W$ is an inactive leaf. Consider the two components forming $W$. If $W$ has only new vertices then only one of these components can be contiguous with vertices in $T - W$ (this follows from the invariant that the picked vertices are contiguous in tree $T$). We label this component $C_1$ and the other component $C_2$. The other case arises when $W$ has both old and new vertices. From the contiguity of old and new vertices it follows that either one of the components forming $W$ has only old vertices or the other component has only new vertices. We label the first component $C_1$ and the other $C_2$. Having suitably labeled the components we check to see if $C_1$ has enough new vertices. If so we go on to the next step with arguments $(C_1, T - C_2)$. If $C_1$ does not have enough new vertices then we move to the next step with arguments $(C_2, T)$.

**Lemma 5.2** *The above algorithm maintains the four invariants mentioned earlier.*

Proof: We consider the four invariants below.

1. From the way we labelled the components $C_1, C_2$ and the order of picking them it follows that the picked vertices are contiguous in the tree $T$. In the other case recall that we first pick vertices from the set $W - C$ and then from $C$. Since $C$ is an inactive leaf of the tree $T$, the set $W - C$ is contiguous with $T - W$ and hence the picked set of vertices is always contiguous in $T$. However, it could be the case that $T - W$ is empty and $C$ contains all old vertices of the tree $T$. In such a setting picking vertices from $W - C$ would not give us a contiguous set of picked vertices. But all old vertices of $T$ cannot belong to an inactive leaf of $T$ because then they would never have been part of $\hat{T}_-$ and hence such a situation would never arise.
2. When the window is narrowed to $W - C$ the tree is also redefined to be $T - C$ so that $W - C$ is the restric-

tion of the component $W$ to this tree. In all the other cases the new window is a component in the previous step and hence also a component in the next step.

3. Suppose that the components that do not pay for themselves at this step are restrictions of the components that did not pay for themselves at the previous step. Since the new window is only a subset of the previous window the invariant continues to hold. The only case when we might be creating a component that does not pay for itself is when we redefine the tree to $T - C_2$ and the window to $C_1$. In this setting the component $C_1$ may not be paying for itself anymore. But this component is also the new window and hence the invariant is still maintained.

4. When we modify the tree some components might become inactive leaves in the new tree. This happens when we modify the tree to $T - C$ or to $T - C_2$. Let $v$ be the vertex in $W - C$ (resp. $C_1$) at which the unique edge from $C$ (resp. $C_2$) is incident. Then any newly created inactive leaf has to include this vertex. Further $v$ is contained in the window $W - C$ (resp. $C_1$) and hence none of these inactive leaves are disjoint from the window or from each other. Similarly one can argue that these newly created inactive leaves are not disjoint from the previously existing inactive leaves.

   The other setting when this invariant might be violated is when we redefine the window. Since the new window is only a subset of the previous one, inactive leaves that were supersets of the previous window continue to be so for the current window as well. We now consider the possibility that an inactive leaf which was a subset of the previous window is disjoint from the current window. This cannot happen when we define the window to be $C$ because then $C$ and this inactive leaf would be disjoint violating the invariant that no two inactive leaves are disjoint. Neither can this happen when we define the window to be $C_2$ because then a subset of $C_1$ would be an inactive leaf violating our assumption that no subset of $W$ was an inactive leaf.

■

**Lemma 5.3** *The total cost of trees $T_1$ and $T_2$ is no more than twice the sum of the potentials of the vertices in $Z$.*

Proof: The proof of this lemma is similar to that of Theorem 3.3 but now we need to argue separately for the iterations before and after $S_1, S_2$ merged to form the component $S$.

We first argue about the iterations before component $S$ was formed. Consider the final window $W$ and the tree $T$ in the above procedure. From the invariants that we maintained it follows that at any iteration of the basic algorithm the only active component that could not pay for its growth was the one containing $W$. At any iteration the tree $T_1$ had at most two inactive leaves – the root component and the component containing the end-point of $e_+$. On the other hand tree $T_2 = T$ had at most one inactive leaf at any iteration; this follows from our invariant that no two inactive leaves are disjoint. Furthermore, since this inactive leaf is not disjoint from $W$, we could not have both an active component that cannot pay for its growth and an inactive leaf of $T_2$ in the same iteration. Thus in an iteration, $T_1 \cup T_2$ either has at most three inactive leaves or it has at most two inactive leaves and an active component that cannot pay for its growth. This, together with the fact that in these iterations the edges of $T_1 \cup T_2$ induce a forest of two trees on the components allows us to argue that the contribution of each iteration to cost$(T_1 \cup T_2)$ is at most twice the total decrease in potentials of the components in that iteration.

In the iterations after $S$ is formed the edges of $T_1 \cup T_2$ induce only one tree over the components and this tree has atmost one inactive leaf – the root component. We however need to argue that when $S$ is formed its potential when we restrict ourselves to vertices of $Z$ is the same as its original potential obtained by considering all vertices of the graph. This is so, because $Z$ includes all the vertices of $\hat{T}_-$. ■

## 6. A Tight example

The graph in Figure 2 has a root $r$ and $p$ clusters of $k/2-1$ vertices each. Two vertices in the same cluster are zero distance apart while the distance between the clusters and the root is as shown in the figure. Assuming that $d \geq 1$, the minimum $k$-tree would pick vertices from three clusters and hence would have cost $d + 2$. The optimum fractional solution on the other hand would assign each vertex other than the root a value $(k-1)/(p(k/2-1))$ and the root a value 1. The edges on the circumference would then have a value $(k-1)/(p(k-2))$ and one of the radial edges would have a value $(k-1)/(p(k/2-1))$. Thus the cost of the fractional solution is $(k-1)/(k-2)+d(k-1)/(p(k/2-1))$ which for very large $p$ is roughly $(k-1)/(k-2)$. Note that this is independent of $d$ so that for large $d$ the gap between the optimum fractional and integral solutions could be unbounded. However, since $d$ is also a lower bound on the cost of the minimum $k$-tree, the lower bound we have for this instance is really $\max\{d, (k-1)/(k-2)\}$. Setting $d$ to $(k-1)/(k-2)$ then gives us a gap of $1 + 2(1 - 1/(k-1))$ between the lower bound and the cost of the minimum $k$-tree.

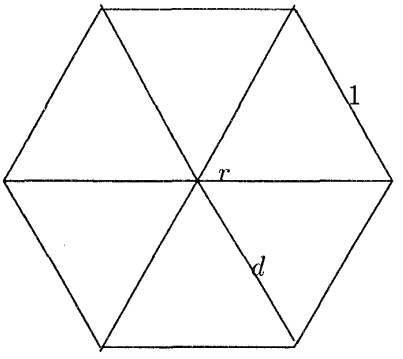


**Figure 2. The example that establishes a gap of $1+2(1-1/(k-1))$ between the lower bound and the optimum.**

## 7. On the minimum tour visiting $k$-vertices

Given an undirected graph $G = (V, E)$ with edge costs $c : E \rightarrow \mathbf{R}^+$ that satisfy traingle inequality, we now consider the probelm of finding the minimum cost simple cycle of that includes some $k$ vertices, we call such a cycle a $k$-tour. As in the case of the spanning tree we consider the rooted version of this problem where we are only interested in cycles that include a specified root vertex $r$. We show how the techniques of the previous sections can be used to obtain a 3-approximation for this problem.

Using the same argument as for the $k$-tree, we can assume that the farthest vertex from the root is included in the minimum $k$-tour. This implies that twice the distance of the farthest vertex from the root is a lower bound on the optimum value. The feasible potential assignments, $\hat{\pi}_p$, that we constructed satisfied the property that for any rooted tree $T$, the cost of $T$ was at least as large as the sum of the potentials of the vertices in $T$. This allowed us to argue that the sum of the $k$ lowest potentials in any feasible assignment was a lower bound on the cost of the minimum $k$-tree. Interestingly our feasible potential assignments also astisfy the property that for any tour $C$, the cost of $C$ is at least twice the sum of the potentials of the vertices in $C$. Therefore, twice the sum of the $k$ lowest potentials in any feasible assignment is a lower bound on the cost of the minimum $k$-tour.

Recall that the $k$-tree returned by our algorithm has cost bounded by twice the sum of the $k$ lowest potentials in a feasible potential assignment plus the distance of the farthest vertex from the root. This implies that this $k$-tree has cost at most 3/2 times the cost of the minimum $k$-tour. By duplicating the edges of the $k$-tree and shortcircuiting an eulerian walk of this "doubled tree" we obtain a $k$-tour of cost at most twice the cost of the $k$-tree and hence within thrice that of the minimum $k$-tour.

## References


[1] B. Awerbuch, Y. Azar, A. Blum, and S. Vempala. Improved approximation guarantees for minimum weight $k$-trees and prize-collecting salesmen. In *Proceedings, ACM Symposium on Theory of Computing*, pages 277–283, 1995.

[2] A. Blum, P. Chalasani, D. Coppersmith, W. Pulleyblank, P. Raghavan, and M. Sudan. The minimum latency problem. In *Proceedings, ACM Symposium on Theory of Computing*, pages 163–171, 1994.

[3] A. Blum, P. Chalasani, and S. Vempala. A constant-factor approximation for the $k$-mst problem in the plane. In *Proceedings, ACM Symposium on Theory of Computing*, pages 294–302, 1995.

[4] A. Blum, R. Ravi, and S. Vempala. A constant factor approximation for the $k$-mst problem. In *Proceedings, ACM Symposium on Theory of Computing*, 1996.

[5] D. Eppstein. Faster geometric $k$-point mst approximation. Technical Report 13, University of California, Irvine, CA, 1995.

[6] N. Garg and D. Hochbaum. An O(log $k$) approximation algorithm for the $k$ minimum spanning tree problem in the plane. In *Proceedings, ACM Symposium on Theory of Computing*, 1994.

[7] M. Goemans and J. Kleinberg. An improved approximation ratio for the minimum latency problem. In *Proceedings, ACM-SIAM Symposium on Discrete Algorithms*, pages 152–158, 1996.

[8] M. Goemans and D. Williamson. A general approximation technique for constrained forest problems. *SIAM J. Comput.*, 24:296–317, 1995.

[9] J. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: a simple new method for the geometric $k$-mst problem. In *Proceedings, ACM-SIAM Symposium on Discrete Algorithms*, pages 402–408, 1996.

[10] S. Rajagopalan and V. V. Vazirani. Logarithmic approximation of minimum weight $k$ trees. Unpublished Manuscript, 1995.

[11] R. Ravi, R. Sundaram, M. Marathe, D. Rosenkrantz, and S. Ravi. Spanning trees short and small. In *Proceedings, ACM-SIAM Symposium on Discrete Algorithms*, 1993.