FRI-1.405B-1-MIP-02

SINGLE FACILITY LOCATION PROBLEMS IN K-TREES

Vladislav Haralampiev – PhD Student
Department of Computing Systems,
Sofia University "St. Kliment Ohridski"
Tel.: +359 888 88 77 87
E-mail: vladislav.haralampiev@fmi.uni-sofia.bg

Abstract: *Single facility location problems are a large class of optimization problems, concerned with the placement of a facility to optimize costs, while considering various factors. Many of these problems require at least quadratic time in general graphs. In contrast, a lot of these problems can be solved in near-linear time in trees. K-trees are a generalization of trees and offer increased representational power. In this paper, we present an algorithm for solving the 1-minisum problem in k-trees in O*(n lg(n)) time and suggest a general framework for efficiently solving single facility location problems in this class of graphs.*

Keywords: *facility location, k-trees, parameterized complexity*

INTRODUCTION

Location models have existed for a long time and are often useful in decision making. A good overview of facility location problems and specifically their obnoxious variant is (Cappanera, 1999). Single facility location problems in graphs are one of the simplest types of location problems. They occur on a regular basis when solving layout problems, like locating machine in a shop, or choosing location of a warehouse (Bidkhori & Moradi, 2009).

For many single facility location problems, solving them in general graphs requires at least quadratic time, because the corresponding algorithms need a table of pairwise distances between vertices. In contrast, many of these problems can be solved in near-linear time, when the underlying graph is a tree.

K-trees are a generalization of trees and offer increased representational power. In this paper, we show that the 1-minisum problem in k-trees can be solved in $O(n \cdot lg(n) \cdot k \cdot 2^k)$ time, where *n* is the number of vertices. Assuming *k* is a small constant, this is an efficient algorithm for the problem. The main idea of the proposed algorithm is not specific to the 1-minisum problem, and can be used as a general framework for solving other single facility location problems in k-trees. To illustrate this, we briefly describe algorithms for the anti-median, 1-maximin and 1-minimax problems. When looking at the result from the perspective of parameterized complexity, the 1-minisum problem and other single facility location problems in k-trees can be solved in time $O^*(n\ lg(n))$ (with parameter *k*).

EXPOSITION

Definitions and properties

We consider connected, undirected graphs with unweighted edges. Graph *G* with vertex set *V* and edge set *E* is denoted as *G(V, E)*. The shortest distance between two vertices *u* and *w* is denoted as *dist(u, w)*. Additionally, with each vertex $u \in V$, we associate a number *weight(u)*. The 1-minisum problem in *G* is defined as follows:

Definition 1 (1-minisum problem). *Find the vertex u from the vertex set V of the graph which minimizes* $\sum_{w \in V} weight(w) \cdot dist(u, w)$.

Intuitively, we are looking for a location of a facility, which minimizes transportation costs. Note that we are interested in the variant of 1-minisum problem, in which we are allowed to place the facility only in the vertices of the graph.

In this paper, we present an efficient algorithm for solving the 1-minisum problem in special class of graphs, called k-trees. The definitions of k-tree and tree decomposition below are taken from

(Biedl, 2005). Note that other equivalent definitions of k-tree exist in the literature, for example in (Beineke & Pippert, 1971).

Definition 2 (perfect elimination order, p.e.o.). *A perfect elimination order for a graph G with n vertices is a vertex order $u_1, u_2, \ldots, u_n$, such that $Pred(u_i)$ is a clique for all $i = 1, 2, \ldots, n$. By $Pred(u_i)$ we denote the set of vertices, which are connected to u_i and are before u_i in the vertex order.*

Definition 3 (k-tree). *Graph G is called a k-tree if G has a perfect elimination order $u_1, u_2, \ldots, u_n$ such that $u_1, u_2, \ldots, u_k$ is a clique and $|Pred(u_i)| = k$ for all $i \in \{k + 1, \ldots, n\}$.*

A special decomposition of graphs, called tree decomposition, turns out to be useful for solving facility location problems in k-trees.

Definition 4 (tree decomposition). *A tree decomposition of a graph G(V, E) is a tree T(I, F), where each node $i \in I$ has a label $X_i \subseteq V$ such that:*

- $\bigcup_{i \in I} X_i = V$.
- *For any edge $(u, w) \in E$, there exists $i \in I$ such that $u, w \in X_i$.*
- *For any $u \in V$, the nodes containing u in their label form a connected subtree of T.*

We will call nodes the vertices of the tree decomposition to distinguish them from the vertices of the original graph *G*. The following lemma shows that every k-tree has a tree decomposition with special properties.

Lemma 1. *Every k-tree G(V, E) has a tree decomposition T(I, F) with size linear in the size of G and satisfying the properties:*

- *For every $i \in I$, $|X_i| \leq k + 1$.*
- *For every $i \in I$ and for every two distinct vertices $u, w \in X_i$, the edge $(u, w) \in E$.*

Proof. Let $u_1, u_2, \ldots, u_n$ be a perfect elimination order of *G* from Definition 3 of k-tree. We will build the required tree decomposition for *G* by induction on *i*. In the tree decomposition we build, for each $i > k$ there will be a node containing $\{u_i\} \cup Pred(u_i)$.

The base case is when $i = k + 1$. We add all of $u_1, u_2, \ldots, u_{k+1}$ into one node of the tree decomposition. Note that these vertices form a clique, so all requirements are satisfied.

Now assume $i > k + 1$ and we have the required tree decomposition for vertices $u_1, \ldots, u_{i-1}$. We create a new node containing $\{u_i\} \cup Pred(u_i)$. These vertices again form a clique. Let u_j be the last predecessor of u_i (closest to the left in the perfect elimination order). We attach the newly created node, containing $\{u_i\} \cup Pred(u_i)$, to the node containing $\{u_j\} \cup Pred(u_j)$ (such a node exists from the induction). From the properties of p.e.o. $Pred(u_i) \subseteq \{u_j\} \cup Pred(u_j)$, so the nodes containing a vertex from $Pred(u_i)$ continue to form a connected subtree in the tree decomposition. Vertex u_i occurs so far only in the newly created node of the tree decomposition, so connectivity is also satisfied. This shows that all required properties still hold after processing u_i. ■

Corollary 1. *Every k-tree G has a tree decomposition T with size linear in the size of G such that it satisfies the properties from Lemma 1 and additionally the degree of every node in T is at most three.*

Proof. Let *T'* be the tree decomposition of *G* from Lemma 1. For every node $u \in T'$ with degree $d > 3$, create $d - 2$ (including *u*) copies of the node, attached in a chain, and attach the original neighbours of *u* to them in such a way that the degree of every node is at most 3. The resulting tree decomposition satisfies all the required properties and its size is linear in the size of *T'*.

■

Further in the paper we assume that we have a tree decomposition of the input graph, satisfying the properties from Corollary 1. This is not a restriction because, given a k-tree, we can construct the perfect elimination order from Definition 3 in linear time by successively removing k-degree vertices (every k-tree has at least one such vertex) and their adjacent edges. Once we have the p.e.o., we can follow the procedures from the proofs above to produce the required tree decomposition.

Centroid decomposition

The algorithm, proposed in this paper, uses centroid decomposition. This is a well-known framework for solving problems on trees, based on the following classic result by Jordan (Jordan, 1869):

Theorem 1. *For every tree with n vertices, there exists a vertex, whose removal partitions the tree into components, each with at most n / 2 vertices.*

Note that finding such a vertex for a tree can be easily done with depth-first search. We can then remove this vertex, recursively solve the given problem for the created connected components, and merge the results. Since the size of the components at least halves, the depth of the recursion is *O(lg(n))*. If the processing at each level works in linear time, the whole recursion will work in time *O(n lg(n))*.

For solving facility location problems in k-trees, we will use centroid decomposition over the tree decomposition of the k-tree. In Figure 1 the general structure of the proposed algorithm is shown in pseudocode.

```
1: function Solve(T(I, F))
2:     if T is small then
3:         Directly solve the problem on T
4:         Return
5:     vr ← centroid of T
6:     components ← components of T after removing vr
7:     for each Component ∈ components
8:         Solve(Component ∪ vr)
9:     Merge results from recursive calls
```

Fig. 1. Centroid decomposition over tree decomposition algorithm

Note that there are two placeholders in the proposed scheme – how to directly solve the problem when *T* is small and how to merge the results of the recursive calls. We will talk about these in the next section. Also, the proposed scheme differs a bit from the usual centroid decomposition, because the recursive calls process the subtrees with one additional node added – the centroid node *vr*. This clearly doesn't change the time complexity. In our case, the placeholders will spend $O(n \cdot k \cdot 2^k)$ time for each level of the recursion, so the whole algorithm will work in $O(n \cdot lg(n) \cdot k \cdot 2^k)$ time.

Algorithm

We will calculate, for every vertex *u* of the input graph *G*, the value $answer(u) = \sum_{w \in V} weight(w) \cdot dist(u, w)$. Then, the answer to the 1-minisum problem is $\min_{u \in V} answer(u)$. In the previous section we outlined the general structure of the proposed algorithm. The function *Solve(T(I, F))* will calculate the answers for the subgraph of *G* corresponding to *T*, taking into account only vertices from this subgraph. More formally, if *G'* with vertex set *V'* is the subgraph of *G*, corresponding to *T*, then *Solve(T(I, F))* will calculate, for each vertex $u \in V'$,

$$answer(u) = \sum_{w \in V'} weight(w) \cdot dist(u, w)$$

Calling the *Solve* function with the whole tree decomposition of the input graph *G* will compute the answer values needed to solve the 1-minisum problem. In the algorithm from Figure 1 there are two placeholders – how to directly solve the problem for small graphs and how to merge the results of the recursive calls.

For the first placeholder, we assume the subgraph is small, if its tree decomposition *T* has less than four nodes. We can use breadth-first search to directly evaluate the answers, restricted to this subgraph. Since *T* has less than four nodes, the corresponding subgraph has *O(k)* vertices, which gives us $O(k^3)$ time complexity, a constant, when *k* is fixed.

For the second placeholder (line 9 in Figure 1), we need to be able to efficiently merge the answers from the recursive calls to *Solve*. Next it will be described how to do the merging in time $O(m \cdot 2^k)$, where *m* here is the number of edges in the subgraph, corresponding to *T*.

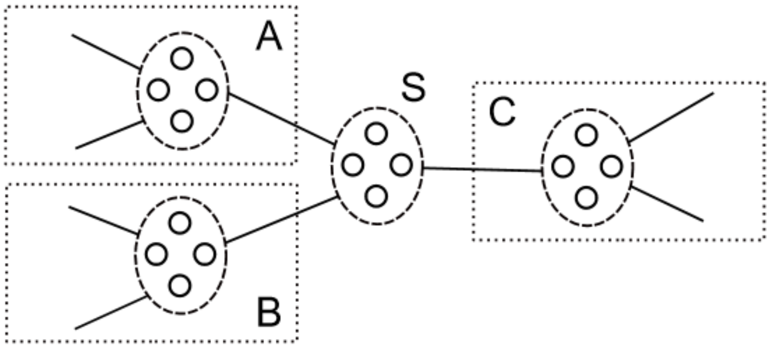

Fig. 2. Merging answers from recursive calls

In Figure 2 *S* is the centroid vertex of *T* and after removing it, three components are formed – *A*, *B* and *C*. From Corollary 1 we know that at most three components are formed, and the case when there are less than three components is analogous. For each $u \in S$, we will calculate *answer(u)* from scratch, using breadth-first search. This takes $O(k \cdot m)$ time.

Pick two vertices $u \in A$ and $w \in B \cup C$. From the properties of tree decomposition we know that all paths from *u* to *w* pass through a vertex from *S*. This means $dist(u, w) = \min_{i \in 0,\dots,k}(dist(u, s_i) + dist(s_i, w))$, where s_0 , s_1 , ..., s_k are the vertices, corresponding to *S*. Let *q* be equal to $\min_{i \in 0,\dots,k} dist(u, s_i)$ and lets call *h* the vector $[dist(u, s_0) - q, \dots, dist(u, s_k) - q]$. With this notation $dist(u, w) = q + \min_{i \in 0,\dots,k}(h_i + dist(s_i, w))$. A crucial observation is that the vector *h* consists of only zeroes and ones. This is because, from Lemma 1, the vertices in *S* form a clique, so the distance from any two of them to any vertex of the graph differs by at most one. Note that the vector *h* depends on the choice of $u \in A$, but the previous statement means, in total, there are at most 2^{k+1} possibilities for *h*.

For each vertex $u \in A$, the recursive calls to *Solve* have already computed *answer(u)*, restricted to $A \cup S$. What is left is to add to *answer(u)* the

$$\sum_{w \in B \cup C} weight(w) \cdot dist(u, w) = \tag{1}$$

$$q \cdot \sum_{w \in B \cup C} weight(w) + \sum_{w \in B \cup C} (weight(w) \cdot \min_{i \in 0,\dots,k} (h_i + dist(s_i, w)))$$

When entering the *Solve* function, we can easily precompute $\sum_{w \in B \cup C} weight(w)$ in linear time in the size of *T*. Also, we can precompute $\sum_{w \in B \cup C}(weight(w) \cdot \min_{i \in 0,\dots,k}(h_i + dist(s_i, w)))$ for each possible vector *h*, using breadth-first search. There are only $O(2^k)$ possibilities for the vector *h*, so the whole precomputation will work in time $O(m \cdot 2^k)$. Once we have these precomputed values, we can compute (1) in constant time for each vertex in *A* (in a breadth-first traversal of the vertices of *A*, starting from the vertices in *S*, so that we directly have the *q* values).

In the previous paragraph we outlined how to update the answer values for the vertices in the component *A*. Clearly, we can apply the same procedure to the other components. The heaviest operation is the precomputation of sums, and it works in time $O(m \cdot 2^k)$. So, merging the answers of the recursive calls in *Solve* can be done in $O(m \cdot 2^k)$. As a corollary from this and the previous paragraph, we get the following theorem:

Theorem 2. *In a k-tree we can solve the 1-minisum problem in time $O(n \cdot lg(n) \cdot k \cdot 2^k)$.*

Proof. The *Solve* procedure we outlined will be used to compute *answer(u)* for each vertex *u* of the graph. The answer to the 1-minisum problem is then $\min_{u \in G} answer(u)$

The *Solve* procedure from the previous section has two placeholders and their implementation was described in this section. Since solving small instances takes $O(k^3)$ time and merging takes $O(m \cdot 2^k)$ time, the work on each level is $O(|E| \cdot 2^k)$. From the reasoning in the previous section it follows that the time complexity of the complete *Solve* procedure is $O(|E| \cdot lg(n) \cdot 2^k)$. In k-trees with n vertices the number of edges $|E| = O(k \cdot n)$. By substituting this into the expression for the time complexity, we get $O(n \cdot lg(n) \cdot k \cdot 2^k)$. ■

Other single facility location problems

We can informally say that the property of k-trees, which is exploited in the proposed algorithm, is that the number of "ways" a shortest path can "pass" through a centroid node, is small. This allows the precomputation we do to be efficient. The proposed algorithmic scheme can be used whenever we need to infer some property in a k-tree, based on shortest paths between pairs of vertices.

As concrete examples we will mention three other single facility location problems. The obnoxious variant of the 1-minisum problem is called anti-median problem. We are looking for a vertex u which maximizes $\sum_{w \in V} weight(w) \cdot dist(u, w)$. The algorithm from the previous section computes this sum for each vertex u of the graph, so for solving the anti-median problem we just need to take the maximum of the answer values, instead of the minimum.

In the 1-minimax problem we seek for a vertex u which minimizes the $\max_{w \in V} weight(w) \cdot dist(u, w)$, the maximum weighted distance to other vertices. This problem is similar to the 1-minisum problem. We will again compute, for each vertex $u \in V$, the maximum weighted distance from this vertex, and take the minimum of these. Merging the results of the recursive calls is more complicated, but can be done in $O(n \cdot lg(n))$ time for each level of the recursion with similar precomputation for vectors h and using a data structure called "convex hull trick" (PEGWiki, 2016). This makes the total complexity of solving the 1-minimax problem $O^*(n \cdot lg^2(n))$, with parameter k.

Finally, the obnoxious variant of 1-minimax problem is called 1-maximin problem and can be solved with similar technique.

CONCLUSION

We described a $O(n \cdot lg(n) \cdot k \cdot 2^k)$ algorithm for solving the 1-minisum problem in k-trees, a generalization of trees. The main idea of the algorithm is not specific to the 1-minisum problem and can be used to develop efficient algorithms for other single facility location problems in k-trees. As examples, we briefly described algorithms for solving the anti-median, 1-minimax and 1-maximin problems. For future work it is left to investigate if the described algorithmic framework can give us efficient algorithms for other problems in k-trees.

REFERENCES

Beineke, L., & Pippert, R. (1971). Properties and characterizations of k-trees. *Mathematika*, 18(1), 141-151.

Bidkhori, M., & Moradi, E. (2009). Single Facility Location Problem. In: *Facility Location. Contributions to Management Science.* Physica, Heidelberg.

Biedl, T. (2005). *Graph-theoretic algorithms. Lecture notes of a graduate course.* University of Waterloo.

Cappanera, P. (1999). A Survey on Obnoxious Facility Location Problems. *University of Pisa, technical report.*

Jordan, C. (1869). Sur les assemblages de lignes. *J. Reine Angew. Math.* 70, 185-190.

PEGWiki. (2016). *Convex hull trick.* Available at:
http://web.archive.org/web/http://wcipeg.com/wiki/Convex_hull_trick (Accessed 20 August 2019).