CENTRAL PLACEMENT OF STORAGE SERVERS IN TREE-LIKE CONTENT DELIVERY NETWORKS

Mugurel Ionuţ Andreica, Nicolae Ţăpuş

Abstract: **In this paper we consider several problems regarding the optimal placement of storage servers in content delivery networks with tree-like topologies (paths, trees and cacti). We present several novel algorithmic solutions for locating (k+p)-centers and connected k-centers (and k-medians) in these networks, both in the continuous and the discrete cases. We comment both on the theoretical and the practical efficiency of the proposed algorithms.**

Index Terms**:** k-center, connected k-median, tree**.**

I. INTRODUCTION

The problem of efficiently placing storage servers in content delivery networks is very important due to the wide spread usage of such networks. The storage servers should be placed such that they minimize the network traffic generated in order to reach them and acquire the stored content, according to an objective metric. Weighted min-max and min-sum metrics are most widely used and, in computer science, the corresponding optimization problems consist of locating the k-center and the k-median of the graph which represents the topology of the network. In this paper we consider only networks with a tree-like topology: cacti, trees and paths. Although the tree-like structure may seem to be only a particular case, we argue that many existing networks have a hierarchical structure, e.g. with users devices at the edge and router backbones at its core. Moreover, many graph topologies can be reduced to tree topologies, by choosing a spanning tree or by decomposing the set of edges into edge disjoint spanning trees.

The *(k+p)*-center problem is the following: Given a graph with n vertices where each vertex has a weight and each edge has a length, we want to place at most $k \geq 0$ ($k+p \geq 1$) servers in the graph, such that the maximum weighted distance from a vertex to its closest server is minimized. If the servers may be placed only at the graph vertices, we are considering the *discrete* case. If they can be placed anywhere along the edges, it is the *continuous* version. Furthermore, $p \geq 0$ *fixed* servers are already placed in the network (at the graph vertices or on its edges) and they can be used by the vertices. Thus, we need to place at most k *extra* servers. Note that although the fixed servers may be located along the graph's edges, we can insert new vertices on the edges at the locations of the fixed servers, obtaining a graph with $n+p$ vertices where the fixed servers are located only at its vertices (and, thus, we will only consider this case). The *(k+p)*-median problem is defined similarly, except that we want to minimize the sum of weighted distances from all the vertices to their closest server. We also consider an extra requirement for the discrete case (of both the center and median problems): the set of (at most) k locations chosen for the servers must form a connected subgraph of the given graph. This requirement makes sense when the data stored on the servers needs to be (periodically) synchronized or when a server may redirect requests to other servers.

This paper is structured as follows. In Section II we introduce the main notations and definitions used throughout the paper. In Section III we present linear algorithms for the unweighted *(1+0)*-center problem in cactus networks, the longest path and the diameter. In Section IV we consider the general *(k+p)*-center problems in trees. In Section V we discuss a center location problem in wireless path networks. In Section VI we introduce an algorithmic framework for several optimization problems in trees, which we use in Section VII to obtain a solution to the connected *(k+0)*-median problem in trees with equal edge lengths. In Section VIII we present related work and in Section IX we conclude.

II. NOTATIONS AND DEFINITIONS

A *path network* with n nodes has the property that its vertices can be numbered $v(1), \ldots, v(n)$, such that the only existing edges are between $v(i)$ and $v(i+1)$ ($1 \leq i \leq n-1$). A *tree network* is an undirected, connected, acyclic graph. A tree may be rooted, in which case a special vertex r will be called its root. Even if the tree is unrooted, we may choose to root it at some vertex. In a rooted tree, we define *parent(r,i)* as the parent of vertex i (when the tree's root is r) and *ns(r,i)* as the number of sons of vertex i (when the tree root is r). For a leaf i, $ns(r,i)=0$ and for the root r, *parent(r,r)* is undefined. The sons of a vertex i are denoted by *s(r,i,j)* ($1 \leq j \leq ns(r,i)$). A vertex j is a *descendant* of vertex i if *(parent(r,j)=i)* or *parent(r,j)* is also a descendant of vertex i. We denote by *T(r,i)* the subtree rooted at vertex i, i.e. the part of the tree composed of vertex i and all of its descendants (and the edges connecting them), when r is the tree root.

A *cactus* is a graph in which any two cycles are edge-disjoint. The *diameter* of a graph is the length of the longest shortest path between any pair of vertices.

III. CACTUS NETWORKS

Every edge *(u,v)* has a weight *we(u,v)* and every vertex u has a weight *wv(u)*. The *length* of a path *v(1), …, v(k)* is the sum of the weights of the vertices *v(i)* ($1 \leq i \leq k$) and of the edges *(v(j),v(j+1))* ($1 \leq j \leq k-1$).

A. Longest Path

We will present a linear time algorithm for the problem of computing the longest path, i.e. the path having the largest *length*. First, we will compute the block-cut vertex tree T_{BC} [1] of the cactus graph. Such a tree can be computed in *O(n+m)* time for a graph with n vertices and m edges. Since $m=O(n)$ in a cactus, this takes linear time. Every vertex of T_{BC} is

either a biconnected component (type *BC*) or a cut vertex (type *C*) of the original graph. We assume that the root *r* is a biconnected component. The neighbors of each vertex corresponding to a biconnected component *B* are the cut vertices which belong to *B*. The sons of each vertex corresponding to a cut vertex *CV* are the biconnected components which contain *CV* (except for the component *B* which is *CV*'s parent).

For every node *x* of T_{BC}, we define *C(x)* as follows: if *x* corresponds to a cut vertex *cv*, then *C(x)=cv*; otherwise, if *x* is not the root of the tree, then *C(x)=C(parent(x))* (i.e. the cut vertex belonging to the biconnected component corresponding to *x* which is its parent in T_{BC}). If *x* is the root of the tree, then *C(x)* can be set to any vertex in the biconnected component corresponding to the tree root.

For every node *x* of T_{BC} we will compute two values: *A(x)*=the length of the longest path starting at *C(x)*, and *B(x)*=the length of the longest path passing through *C(x)* if *x* is of type *C*, or through any vertex of the biconnected component corresponding to *x*, if *x* is of type *BC*. These paths may contain only vertices contained in node *x*'s subtree of T_{BC}. The length of the longest path is *B(r)*. It is easy to compute these values for a node *x* of type *C*. We compute $A_1(x)=max\{wv(C(x)), max\{A(y)|y \text{ is a son of } x\}\}$ and set y_1 to the son *y* of *x* with the largest value *A(y)*. We also compute $A_2(x)=max\{A(y)|y \text{ is a son of } x,\ y\neq y_1\}$ (if *x* has at most one son, then $A_2(x)=wv(C(x))$). We have $A(x)=A_1(x)$ and $B(x)=A_1(x)+A_2(x)-wv(C(x))$. For a node *x* of type *BC*, we will denote by *v(x,1), …, v(x,nv(x))* the vertices of the biconnected component corresponding to *x*. In a cactus, every biconnected component is a cycle. If the component consists of only one edge *(v(x,1),v(x,2))*, we will double this edge, in order to obtain a cycle composed of two vertices and two edges (of equal weights). We will order the vertices in the order in which they appear on the cycle, starting from *v(x,1)=C(x)* and continuing in one of the two possible directions. Thus, we have *v(x,1)=C(x)*, *v(x,i)* and *v(x,i+1)* are adjacent ($1\leq i\leq nv(x)-1$), and *v(x,nv(x))* and *v(x,1)* are adjacent. We will assign a value *l(x,j)* to each vertex *v(x,j)*. If *v(x,j)* ($1\leq j\leq nv(x)$) is a cut vertex corresponding to a node *y* which is a son of *x* in T_{BC}, we set $l(x,j)=A(y)-wv(v(x,j))$; otherwise, $l(x,j)=0$. We will first compute $A_1(x)$, the length of the longest path starting at *v(x,1)* (*=C(x)*), not containing the edge *(v(x,nv(x)),v(x,1))*. We traverse the vertices in increasing order of their index *j* and compute:

$$lsum_1(x,j) = wv(v(x,j)) + \sum_{i=1}^{j-1}(\mathrm{wv}(\mathrm{v}(\mathrm{x},\mathrm{i})) + \mathrm{we}(\mathrm{v}(\mathrm{x},\mathrm{i}),\mathrm{v}(\mathrm{x},\mathrm{i}+1))) \quad (1)$$

$lsum_1(x,0)=0$, $lsum_1(x,1)=wv(v(x,1))$ and for $j>1$, $lsum_1(x,j)=lsum_1(x,j-1)+we(v(x,j-1),v(x,j))+ wv(v(x,j))$. $A_1(x)=max\{lsum_1(x,j)+l(x,j)|1\leq j\leq nv(x)\}$. We will now compute $A_2(x)$, the length of the longest path starting at *v(x,1)*, containing the edge *(v(x,nv(x)), v(x,1))*. We compute the values $lsum_2(x,j)$:

$$lsum_2(x,j) = wv(v(x,j)) + \sum_{i=j}^{nv(x)-1}(\mathrm{wv}(\mathrm{v}(\mathrm{x},\mathrm{i}+1)) + \mathrm{we}(\mathrm{v}(\mathrm{x},\mathrm{i}+1),\mathrm{v}(\mathrm{x},\mathrm{i}))) \quad (2)$$

We have $lsum_2(x,nv(x))=wv(v(x,nv(x)))$ and for $2\leq j\leq nv(x)-1$, $lsum_2(x,j)=lsum_2(x,j+1)+we(v(x,j+1), v(x,j))+wv(v(x,j))$. We set $A_2(x)$ to $max\{lsum_2(x,j)+ l(x,j)+wv(v(x,1))+we(v(x,1),v(x,nv(x))) \mid 2\leq j\leq nv(x)\}$. *A(x)* will be equal to $max\{A_1(x), A_2(x)\}$.

In order to compute *B(x)*, we consider the same two cases as before, and compute two values, $B_1(x)$=the length of the longest path passing through some vertex *v(x,j)* ($1\leq j\leq nv(x)$), not containing the edge *(v(x,1),v(x,nv(x)))*, and $B_2(x)$=the length of the longest path passing through some vertex *v(x,j)*, containing the edge *(v(x,1),v(x,nv(x)))*. $B(x)= max\{B_1(x), B_2(x)\}$. In order to compute $B_1(x)$, we will assign to each vertex *v(x,j)* the same value *l(x,j)* as before and then compute the same values $lsum_1(x,j)$. Afterwards, we will also compute the values $lmax_1(x,j)$. We have $lmax_1(x,0)=0$ and $lmax_1(x,j)= max\{lmax_1(x,j-1), l(x,j)+wv(v(x,j))-lsum_1(x,j)\}$ ($1\leq j\leq nv(x)$). Then, we compute $lmax_2(x,j)=max\{wv(v(x,j))+l(x,j), l(x,j) + lsum_1(x,j) + lmax_1(x,j-1)\}$ ($1\leq j\leq nv(x)$). $lmax_2(x,j)$ represents the length of the longest path containing a segment *v(x,i), …, v(x,j)* ($1\leq i\leq j$) of the cycle, and not containing any other part of the cycle. Thus, the path starts at some vertex *v(x,i)* ($i\leq j$) or somewhere in its subtree (if *v(x,i)* is a cut vertex), walks along the cycle from *v(x,i)* to *v(x,j)* (in increasing order of the vertex indices) and either ends at *v(x,j)* or at a vertex in *v(x,j)*'s subtree (if *v(x,j)* is a cut vertex). $B_1(x)=max\{lmax_2(x,j)|1\leq j\leq nv(x)\}$.

In order to compute $B_2(x)$, we need to compute the values $lsum_1(x,j)$ and $lsum_2(x,j)$, defined previously. Afterwards, we compute $lmax_3(x,j)$ ($lmax_3(x,1)= lsum_1(x,1)+l(x,1)$ and $lmax_3(x,j>1)=max\{lmax_3(x,j-1), lsum_1(x,j)+l(x,j)\}$) and $lmax_4(x,j)$ ($lmax_4(x,nv(x))= lsum_2(x,nv(x))+l(x,nv(x))$ and $lmax_4(x,j<nv(x)) = max\{lmax_4(x,j+1), lsum_2(x,j)+l(x,j)\}$). $B_2(x)$ is $we(v(x,1),v(x,nv(x))) + max\{lmax_3(x,j)+lmax_4(x,j+1)| 1\leq j\leq nv(x)-1\}$. $lmax_3(x,j)$ is the length of the longest path starting at *v(x,1)* and ending at a vertex *v(x,i)*, with $i\leq j$ (or in *v(x,i)*'s subtree). $lmax_4(x,j)$ is the length of the longest path starting at *v(x,nv(x))* and ending at a vertex *v(x,i)*, with $i\geq j$ (or in this vertex's subtree). Thus, the path corresponding to $B_2(x)$ is composed of the edge *(v(x,1), v(x,nv(x)))*, a segment of the cycle starting at *v(x,1)* and ending at some vertex $v(x,i_1)$ ($1\leq i_1$), a segment of the cycle starting at *v(x,nv(x))* and ending at some vertex $v(x,i_2)$ ($i_1<i_2\leq nv(x)$), plus the longest paths starting at $v(x,i_1)$ and $v(x,i_2)$ and ending in their subtrees (these paths may be void). All the values can be computed in $O(n+q)$ time, where *n* is the number of vertices of the graph and $q=O(n)$ is the number of vertices of T_{BC}.

B. Discrete 1-Centers and Diameter

In order to compute the center vertices and the diameter of a cactus, we will use the same block-cut vertex tree T_{BC}. For each node *x* of T_{BC}, we will compute *A(x)*=the longest shortest path starting at *C(x)*, which may contain only vertices located in node *x*'s subtree. For each vertex *i* of the cactus, we will compute $l_1(x)$, $l_2(x)$ and $l_3(x)$, the lengths of the longest, 2^{nd} longest and 3^{rd} longest shortest paths in the graph, with the condition that these *3* paths *are*

computed at different nodes of T_{BC}. Initially, the values $l_1(i)$, $l_2(i)$ and $l_3(i)$ are set to *0*. We also maintain the nodes $x_p(i)$ of T_{BC} where the corresponding $l_p(i)$ value was computed (initially, $x_p(i)=0$, $p=1,\ldots,3$) During the algorithm described below, we will frequently identify a candidate value *val* (computed at a node x of T_{BC}) for $l_1(i)$, $l_2(i)$ and $l_3(i)$. Every time we do this, if we have $x_p(i)=x$ (for some $p=1,\ldots,3$), we replace $l_p(i)$ by $max\{l_p(i), val\}$ and then re-sort the values $l_1(i), \ldots, l_3(i)$. If $x\neq x_p(i)$ (for all $p=1,\ldots,3$), we will compute val_1, val_2 and val_3, the 1^{st}, 2^{nd} and 3^{rd} maximum values in the (multi)set $\{val, l_1(i), l_2(i), l_3(i)\}$. The new values of $l_1(i)$, $l_2(i)$ and $l_3(i)$ will be val_1, val_2 and val_3 (in this order); we then also set the values $x_p(i)$ accordingly ($p=1,\ldots,3$).

For a type *C* node x of T_{BC}, we have $A(x)=max\{wv(C(x)), max\{A(y)|y \text{ is a son of } x\}\}$. For a type *BC* node x of T_{BC}, we consider an ordering of the vertices in the corresponding cycle (biconnected component): $v(x,1)=C(x)$, $v(x,2), \ldots, v(x,nv(x))$. We will compute $WC(x)$, the sum of all the edge and vertex weights in the cycle. Afterwards, we will double this list of vertices, by attaching a copy of the list at the end of the list. Thus, we will have $nv'(x)=2\cdot nv(x)$ elements in the list, with $v(x,j)=v(x,j-nv(x))$ ($nv(x)<j\leq nv'(x)$). We will compute the same $lsum_1(x,j)$ and $l(x,j)$ values as in the longest path case (for all the $nv'(x)$ vertices). We will traverse all the vertices in order, from *1* to $nv'(x)$, and maintain a sorted double-ended queue (deque) *DQ*. All the pairs (p,val) in *DQ* will be maintained in increasing order. Whenever we want to add a value (q,val) at the end of *DQ*, we will repeatedly remove the element (p,val') at the end of *DQ*, as long as $val'\geq val$; only after this will we insert (q,val) at the end of *DQ*. For each vertex $v(x,1\leq j\leq nv'(x))$, we will first add $(j,l(x,j)+wv(v(x,j))-lsum_1(x,j))$ at the end of *DQ*. Then, we will repeatedly remove the element (p,val) at the front of *DQ*, as long as $(p<j)$ and $(lsum_1(x,j)-lsum_1(x,p)-wv(v(x,j)) > (WC(x)-wv(v(x,j))-wv(v(x,p)))/2)$. After this, a candidate value for $l_1(v(x,j))$, $l_2(v(x,j))$ and $l_3(v(x,j))$ will be computed (at node x) as follows: (1) let (p,val) be the first element in *DQ*; (2) the candidate value will be $val+lsum_1(x,j)$. At the end of this stage, we will consider a different ordering v': $v'(x,1)=v(x,1)$ and $v'(x,2\leq j\leq nv'(x))=v(x,nv'(x)-j+2)$ (this is the opposite ordering on the cycle, starting from $v(x,1)$). We will run the algorithm described in this paragraph again, considering the new ordering (we start from (re)computing the $lsum_1(x,j)$ and $l(x,j)$ values, and then we traverse the vertices in the new order). After processing the node x (of type BC), we set $A(x)=max\{l_p(C(x))|1\leq p\leq 3\}$.

After running the algorithm for the entire tree T_{BC}, we need to traverse the tree again, top-down this time (i.e. we consider a node x before all of its sons). If x is a non-root node of type *BC*, we need to consider the following candidate values for $l_1(v(x,i))$, $l_2(v(x,i))$ and $l_3(v(x,i))$ (considering the same ordering of the vertices, starting from $v(x,1)=C(x)$): for a vertex $v(x,j>1)$, a candidate value (computed at node x) would be $dist(v(x,1), v(x,j))+max\{l_p(v(x,1)))|1\leq p\leq 3, x_p(v(x,1))\neq x\}$ (the last term is the length of the longest shortest path that starts at $C(x)$ and does not pass through another vertex of x); $dist(v(x,1), v(x,j))=min\{lsum_1(x,j), WC(x)-lsum_1(x,j)+wv(v(x,1))+wv(v(x,j))\}$ (i.e. the length of the shortest path between $v(x,1)$ and $v(x,j)$ on the cycle). The radius of the cactus graph is $r=min\{l_1(i)|1\leq i\leq n\}$ and the diameter is $max\{l_1(i)+l_2(i)|1\leq i\leq n\}$. The center vertices of the graph are those vertices i with $l_1(i)=r$. The complexity is $O(n)$.

IV. TREE (K+P)-CENTERS

We consider a tree with n vertices, in which every vertex u has a weight $wv(u)$ and every edge (u,v) has a length $l(u,v)$. The p fixed servers are placed at some of the tree vertices. The weighted distance from a vertex u to a server v in the tree is $wd(u,v)=wv(u)\cdot dist(u,v)$, where $dist(u,v)$ is the sum of the lengths of the edges on the unique path between u and v. We will binary search the minimum weighted distance *WD*, such that the weighted distance from every vertex u to the closest server is at most *WD*. For a candidate weighted distance WD_{cand}, we will perform the following feasibility test. For each vertex i, we compute $dmax(i)=WD_{cand}/wv(i)$. We root the tree at an arbitrary vertex r and traverse the tree bottom-up (from the leaves towards the root). We will maintain a counter *nf*, initialized to *0*. For each vertex i we will compute the values: $dmin(i)$=the (non-weighted) distance to the closest server in $T(r,i)$ and $smin(i)$ the maximum distance away from i at which a new server needs to be placed. We initialize $dmin(i)=+\infty$ and $smin(i)=dmax(i)$. If we have a fixed server placed at vertex i, we set $dmin(i)=0$ and $smin(i)=+\infty$; if not, we traverse all the sons $s(r,i,j)$ of vertex i and, for each son, we perform the following updates: $dmin(i)=min\{dmin(i), l(i,s(r,i,j))+dmin(s(r,i,j))\}$ and $smin(i)=min\{smin(i), smin(s(r,i,j))-l(i,s(r,i,j))\}$. If $(dmin(i)\leq smin(i))$ then we set $smin(i)=+\infty$. If $(i\neq r)$ and $(smin(i)<l(parent(r,i),i))$ then: (1) in the discrete case, we place a server at vertex i and set $dmin(i)=0$; (2) in the continuous case, we place a server on the edge $(i, parent(r,i))$, at distance $smin(i)$ from i and, after this, we set $dmin(i)=-smin(i)$. In both situations ((1) and (2)), we increment *nf* by *1* afterwards, and then set $smin(i)=+\infty$. If $(i=r)$ and $(smin(i)<dmin(i))$ then we place a new server at vertex r and increase *nf* by *1*. If, at the end, we have $nf\leq k$, then WD_{cand} is feasible; otherwise, it is not. The time complexity of the algorithm is $O(n\cdot log(WMAX))$ (*WMAX* is $max\{wv(i)\}$ multiplied by the length of the longest path).

For the connected k-center problem in trees, we could only find a solution for the unweighted case (i.e. all vertex weights are equal to *1*). We will start with the case where all the edges have unit length. In this case, we can repeatedly remove the layers of leaves from the tree, until we remain with only one or two vertices (the tree center(s)). We consider the layers as $l(1), \ldots, l(nl)$, sorted from the center(s)

towards the leaves of the tree ($l(1)$ contains the tree center(s)); each layer $l(i)$ has $nlv(i)$ vertices. We will find the largest index j, such that $nlv(1)+\ldots+nlv(j)\leq k$ and ($j=nl$ or $nlv(1)+\ldots+nlv(j+1)>k$) by linear or binary search. We will place the k servers at all the vertices in $l(1), \ldots, l(j)$. When the edges have different lengths, we compute for each vertex i the value $dmax(i)$=the largest distance from vertex i to one of the tree leaves. Then, we sort the vertices such that $dmax(v(1))\geq\ldots\geq dmax(v(i))$. For both the discrete and continuous cases, we will place the k servers at $v(1), \ldots, v(k)$. It is easy to prove that they form a connected subset. The algorithm has linear complexity in the unit edge length case and $O(n\cdot log(n))$ for the case with different edge lengths.

V. CENTERS ON WIRELESS PATH NETWORKS

In this section we consider a wireless path network modeled as a set of n points located on the real line. Each point i is a node of the network, is located at coordinate x_i and has a weight w_i (e.g. its expected service demand or its expected amount of requested data). We are interested in placing (at most) k identical servers, each of them easily covering all the nodes at a distance equal to $L/2$ around them. In geometric terms, each server is the midpoint of an interval of fixed length $L\geq 0$, which covers all the points contained in the interval. The weighted distance from a point x_j to an interval $[a,b]$ is: 0, if $a\leq x_j\leq b$; $w_j\cdot(a-x_j)$, if $x_j<a$; $w_j\cdot(x_j-b)$, if $x_j>b$. If the weighted distance is non-zero, the corresponding vertex needs to increase its power consumption in order to reach the closest server. We want to place the k servers (intervals) in such a way that the maximum weighted distance from a point to the closest interval is minimized. Furthermore, p fixed servers are already placed. We will binary search the weighted distance WD with a specified accuracy. The feasibility test consists of computing for each point i an interval $[lx_i,rx_i]$ [4], where the left endpoint of one of the k intervals can be placed. We will first verify if any of these intervals contains one of the left endpoints of the p fixed servers. We will ignore all such intervals in the decision algorithm, because the points associated with these intervals are "satisfied" by the p fixed servers (i.e. are already within weighted distance WD from one of these servers). In order to exclude the points which are satisfied by the p fixed servers, we sort the intervals assigned to each point and the left endpoints of the intervals of the fixed servers and maintain a set S of open point intervals (an interval is opened when its left endpoint is encountered and closed when we encounter its right endpoint). When we reach the left endpoint of a fixed server, we mark all the open point intervals at that moment (and remove them from S). In the end, we sort the unmarked intervals and we compute the minimum number of points np such that every interval is pierced by at least one point. If $np>k$, then a larger candidate weighted distance is tested; otherwise, we test a smaller one.

We will now present some improvements for $k=1$ and $p=0$. In *Solution 1*, the feasibility test becomes linear (no sorting is required) and consists of comparing the smallest right endpoint sre of an interval $[lx_i,rx_i]$ against the largest left endpoint lle of an interval $[lx_j,rx_j]$. If $sre\geq lle$, then the candidate weighted distance is feasible; otherwise, it is not. In *Solution 2* we define a function $dmax(q)$=the maximum weighted distance from one of the n points to the interval $[q,q+L]$. This function is *unimodal*, i.e. it descends up to $q=q_0$, where $dmax(q_0)$ is the minimum value of $dmax(q)$ and then it ascends again. As it is well-known, we can find the minimum value (and x-coordinate) of a unimodal function by using binary search on its "derivative". To be more precise, we binary search the optimal value q_0 in the interval $[a,b]$, where a is the x-coordinate of the leftmost point and b is $max\{a,c-L\}$, with c being the x-coordinate of the rightmost point. The feasibility test for a value q consists of computing $dif(q)=dmax(q+\varepsilon)-dmax(q)$, where $\varepsilon>0$ is a very small constant. If $dif(q)\geq 0$, then $q\geq q_0$; otherwise, $q<q_0$. The feasibility test runs in linear time (it computes in $O(1)$ time the weighted distance from each point to the interval and selects the maximum distance).

Although the solutions presented above are more efficient than the standard algorithm for interval k-centers [4] (for $k=1$ and $p=0$), we can improve the complexity to $O(n)$, if the points are sorted. We will first introduce a linear algorithm for computing the upper-envelope of right-oriented half-lines.

A. Upper Envelope of Right-Oriented Half-Lines

Let's consider a set of n half lines with their origins at $(x_{i,0}, y_{i,0})$ and with equations of the form $y_i(x)=y_{i,0}+w_i\cdot(x-x_{i,0})$ $(1\leq i\leq n)$, defined only for x in $[x_{i,0},\infty]$ (w_i is the slope of the half line). The upper envelope of these half-lines consists of a set of line segments (except for the last part, which is a half line). Each line segment (x_a,y_a)-(x_b,y_b) is part of a half-line i such that the values $y_i(x)$ $(x_a\leq x\leq x_b)$ are larger than the values of all the other half lines. We will present a linear time algorithm for computing the upper envelope of these half-lines, if they are given in sorted order of their origins (ascending after $x_{i,0}$ and descending after $y_{i,0}$, for equal values of $x_{i,0}$), i.e. for $i<j$ we have $(x_{i,0}<x_{j,0})$ or $((x_{i,0}=x_{j,0})$ and $(y_{i,0}>y_{j,0}))$. We will assume that no two half-lines i and j have the same origin; if they do, then we will keep the one with the larger slope. Moreover, we need to have the following condition: the origins of every half line i $(x_{i,0}, y_{i,0})$ must be located below every half line $j<i$. With this condition, we present the following algorithm: We will maintain a stack, containing pairs $(hl, xfirst)$, where hl is the index of a half-line and $xfirst$ is the smallest x value where the y-value of the half line hl is the largest among the values of all the other half lines. The pairs will be sorted in increasing order of both hl and $xfirst$. We will insert the half-lines in the stack in increasing order of their index. If the current half-line i has a slope which is smaller

than the slope of the half-line *hl* at the top of the stack, then we will discard the half-line *i*, because it will not be part of the upper envelope. Otherwise, we will compute the x-value *xcross* where *i surpasses hl*. If *xcross* is smaller than the value *xfirst* of *hl*, then we remove *hl* from the top of the stack and repeat the procedure; otherwise, we insert the pair *(i, xcross)* at the top of the stack. The pseudocode is given below:

UpperEnvelopePositiveHalfPlane():

stack={(1,$x_{1,0}$)}
for *i=2* **to** *n* **do** // i=2, 3, …, n
 (hl, xfirst)=stack.top()
 if *($w_i \le w_{hl}$)* **then continue** // jump to the next iteration
 popped=true
 while *((stack.size()>0)* **and** *(popped=true))* **do**
  (hl, xfirst)=stack.top()
  xcross=the crossing point of half-lines i and hl
  if *(xcross≤xfirst)* **then {** *stack.pop(); popped=true* **}**
  else *popped=false*
 if *(stack.size()>0)* **then** *stack.push((i, xcross))*
 else *stack.push((i, $x_{i,0}$))*

At the end of the algorithm, the pairs on the stack define the upper envelope of the *n* half-lines. A pair *(hl(b), xfirst(b))*, located on top of a pair *(hl(a), xfirst(a))* defines a line segment *(xfirst(a), $y_{hl(a)}$(xfirst(a))) – (xfirst(b), $y_{hl(a)}$(xfirst(b)))* of the upper envelope. The topmost pair *(hl(top), xfirst(top))* defines the last part of the upper envelope, given by the half-line *hl(top)*, starting from *x=xfirst(top)*. The time complexity of the algorithm is *O(n)*, because each half line is pushed on and/or popped from the stack at most once. Some particular situations where this algorithm can be used are when all the lines have their origins on the *OX* or the *OY* axis.

B. Interval 1-Center on a Path Network

Before proceeding to the algorithm, we should notice that if $L \ge x_n - x_1$, then all the points can be covered by one interval of length *L* and, thus, the weighted distance is *0*. Otherwise, we assign to each point *i* a right-oriented half-line with slope $wl(i)=w_i$, starting at $(x_i,0)$. We will compute the upper-envelope of these half-lines, restricted to the interval $[x_1,x_n]$. In order to achieve this, we can use the linear time algorithm presented before, because the half-lines assigned to the *n* points satisfy the conditions required by that algorithm. Thus, the upper-envelope of these half-lines consists of *lp* points *a(1), a(2), ..., a(lp)* and *lp-1* indices *f(1), f(2), ..., f(lp-1)*, where: $a(1)=x_1$; $a(lp)=x_n$; any two intervals *(a(i),a(i+1))* are disjoint; the union of the intervals *[a(i),a(i+1)]* is $[x_1,x_n]$; the half-line associated to point *f(i)* has the largest *y-value* among all the other half-lines on the interval *[a(i),a(i+1)]*. With this upper envelope, we can compute in *O(1)* time the largest distance *dleft* from a point located to the left of an interval *[q,q+L]*, if we know the interval *[a(j),a(j+1)]* which contains *q* ($dleft=wl(f(j))\cdot(q-x_{f(j)})$). Afterwards, we assign to each point *i* a left-oriented half-line with slope $wr(i)=w_i$, starting at $(x_i,0)$. By running the same linear algorithm for computing the upper envelope and considering the half-lines from right to left, we obtain the upper-envelope of this second set of half-lines. More exactly, we obtain *rp* points, *b(1), …, b(rp)*, such that $b(1)=x_1$, $b(rp)=x_n$, every two intervals *[b(i),b(i+1)]* are disjoint and their union is $[x_1,x_n]$. We also obtain the *rp-1* indices *g(1), …, g(rp-1)*, meaning that the half-line assigned to point *g(i)* has the largest y-value among all the other half-lines on the interval *[b(i),b(i+1)]*. *lp* and *rp* are of the order *O(n)*.

We will now traverse in *O(n)* time the intervals *[a(i),a(i+1)]* with the left endpoint *q* of a length *L* interval, while the right endpoint *q+L* traverses the *[b(j),b(j+1)]* intervals. We start with *q=a(1)* and find the interval *[b(v),b(v+1)]* containing *q+L*. For each position of *q* and considering that *q* is located in *[a(u),a(u+1)]* ($q<a(u+1)$) and *q+L* is located in *[b(v),b(v+1)]* ($q+L<b(v+1)$), we define $d(q)=min\{a(u+1)-q,\ b(v+1)-(q+L)\}$. We need to compute the minimum value of the maximum weighted distance if the left endpoint of the interval belongs to *[q,q+d(q)]*. Let's consider $yleft_u(p)$=the value of the right-oriented half-line assigned to point *f(u)*, at the coordinate *p* and $yright_v(p)$=the value of the left-oriented half-line assigned to point *g(v)*, at the coordinate *p+L*. Thus, we want to compute $min\{max\{yleft_u(p),yright_v(p)\} |\ q \le p \le q+d(q)\}$. The candidate values of *p* which could minimize the maximum weighted distance are *p=q*, *p=q+d(q)* and *p=peq*, where, if it exists, *peq* is the coordinate such that $yleft_u(peq)=yright_v(peq)$. Since $yleft_u$ and $yright_v$ are linear functions, we can compute *peq* in *O(1)* time by interpreting $yleft_u$ and $yright_v$ as straight lines instead as half-lines (*peq* is the x-coordinate of the point of intersection of two lines) and then verifying if *peq* belongs to *[q,q+d(q)]*. After this, we increment *q* by *d(q)*. If *q* becomes equal to *a(u+1)*, we increment *u* by *1*; if *q+L* becomes equal to *b(v+1)*, we increment *v* by *1*. We stop when *v>rp*.

The following table shows the running times of the algorithm described above and *Solution 1* (described previously). In *Solution 1*, the answer was searched for with a precision of *4* decimal digits. The range of the coordinates was $[0,10^8]$.

Table 1. Running times: O(n·log(WMAX))-vs-O(n)

N	L	O(n·log(WMAX))	O(n)
1000000	17	3,1 sec	0,78 sec
1000000	900000	3,2 sec	0,78 sec
900000	0	3,2 sec	0,73 sec
999999	100000	3,4 sec	0,81 sec
100000	234567	0,4 sec	0,07 sec

VI. ALGORITHMIC FRAMEWORK FOR SOME OPTIMIZATION PROBLEMS IN TREES

In this section we introduce an algorithmic framework which we will use in the following section for developing (new) solutions for the connected k-center and the (restricted) connected k-median in a tree. The framework is applicable in the following situation. Let's assume that we want to compute a subset of vertices of a tree, subject to certain restrictions, such that the value of a global property is

optimized. Let's also assume that we know how to compute the optimal subset, with the restriction that the subset contains a given vertex i (algorithm $A(i)$). This computation is performed bottom-up in $Q(n)$ time, by rooting the tree at the vertex $\underline{i}$. In order to compute the optimal subset, an easy solution would be to root the tree at every vertex i, run the algorithm for each vertex, and choose the best solution obtained. This would take $O(n \cdot Q(n))$ time. For some problems, we can do better and maintain the $Q(n)$ time complexity, by using a top-down approach. Let's assume that $V(r,i)$ is a tuple of values computed for each vertex i of the tree, during the execution of the algorithm $A(r)$ (with vertex r as the root). $V(r,i)$ is computed based on the values $V(r,s(r,i,j))$ of the sons of vertex i. For the root r, we can derive the optimal solution based on the values $V(r,r)$, but we cannot do this for the other vertices $i \neq r$, because the values $V(r,i)$ are only based on the vertices in $T(r,i)$. Thus, we are interested in computing $V(i,i)$ for each vertex i. The proposed algorithmic framework consists of two stages. In the first stage, we choose an arbitrary vertex r and run the algorithm $A(r)$. This way, we compute the values $V(r,i)$ for every vertex i. The 2^{nd} stage consists of the following recursive algorithm, called with the vertex r as its (first) argument.

TopDownTraversal(i, r):
if $(i \neq r)$ **then** {
Compute$(parent(r,i), i, r)$; **Compute**(i, i, r) }
for $j=1$ **to** $ns(r,i)$ **do TopDownTraversal**$(s(r,i,j), r)$
Compute(i, root, r):
if $(i \neq root)$ **then** $V(root,i)$=**UpdateRemove**$(i, root)$
else $V(i,i)$=**UpdateAdd**$(i, parent(r,i), r)$

As can be noticed, we only need to define the functions *UpdateRemove* and *UpdateAdd*. *UpdateRemove* computes the value $V(root,i)$ based on the same values which determine $V(i,i)$ ($V(r,s(r,i,j))$ and $V(i,parent(r,i))$), from which we must disconsider the son $s(r,i,j)=root$. *UpdateAdd* computes the value $V(i,i)$ based on the values $V(r,s(r,i,j))$ of vertex i's sons (which determine $V(r,i)$), at which we add an "extra son", the vertex $parent(r,i)$, which contributes with $V(i,parent(r,i))$. The most straight-forward implementation is to reconsider all the current sons of a vertex (which may include its former parent and may exclude one of its former sons) and compute the required values the same way they are computed in the algorithm A. Let's assume that algorithm A takes $O(ns(r,i)^c)$ time to compute the values for a vertex i (thus, $Q(n)=O(n^c)$). If we use the straight-forward implementation, the extra time complexity for each edge $(parent(r,i),i)$ would be $O(ns(r,parent(r,i))^c)$. Thus, the overall time complexity would be $O(n^{c+1})=O(n \cdot Q(n))$. In case the degree of every vertex is bounded by a constant D, then the time complexity remains $O(Q(n))$ and the straight-forward implementation is acceptable. Even if the degree is not bounded, implementing things this way may still produce large improvements over the obvious solution, in terms of running time. However, if we consider the star with one central vertex and $n-1$ leaves, we can see that we obtain no improvement over the obvious solution of considering every vertex as a possible root of the tree. We will consider two problems in this section, in order to show how the algorithmic framework can be used.

A. Minimum Frequency Conversion Costs in WSNs

In [7], a slightly less general form of the following problem was considered. We are given a tree, representing a wireless sensor network, in which we need to identify a source vertex s. This vertex needs to receive a message on a frequency f and then broadcast the message to all the other vertices of the tree. We will consider the tree rooted at s. Each vertex i can receive the data from its parent on a frequency fin with a power cost $c_{recv}(i,fin)$ and can send the data further to all of its sons on the same frequency $fout$, with a power cost of $c_{conv}(i,fin,fout)$. The set of frequencies is $\{1,\ldots,k\}$. We want to find the source vertex s and the distribution strategy, such that the total power cost (of all the vertices) is minimum. Once a vertex r is chosen as the root, we can compute the following values: $C_{min}(r,u,fin,fout)$=the minimum total cost of broadcasting the data in $T(r,u)$, if vertex u receives the data on the frequency fin and sends it further on the frequency $fout$, and $C_{best}(r,u,fin)$:

$$C_{best}(r,u,fin) = c_{recv}(u,fin) + \min_{1 \le fout \le k} \{C_{\min}(r,u,fin,fout)\} \quad (1)$$

$$C_{\min}(r,u,fin,fout) = c_{conv}(u,fin,fout) + \sum_{j=1}^{ns(r,u)} C_{best}(r,s(r,u,j),fout)\} \quad (2)$$

For a fixed root r, the answer is $C_{best}(r,r,f)$. The problem was solved in [1] using an approach similar to the generic framework we presented. However, when lifting a vertex i as the new root of the tree, it recomputed the values by traversing all the neighbors of vertex i and $parent(r,i)$. This takes $O(n \cdot k^2)$ time only if each vertex has at most a constant number of neighbors (this was not explicitly stated in [7]). We assume that the $V(r,i)$ values are a tuple composed of all the $C_{min}(r,i,*,*)$ values, followed by the $C_{best}(r,i,*)$ values. In order to obtain the $O(n \cdot k^2)$ time complexity irrespective of the number of neighbors a vertex has, the *UpdateRemove* function needs to return a tuple containing: $C_{min}(root,i,fin,fout) = C_{min}(i,i,fin,fout) - C_{best}(r,root,fout)$ ($1 \le fin,fout \le k$) and the updated values, $C_{best}(root,i,*)$ (these are recomputed from scratch, after the values $C_{min}(root,i,*,*)$ are obtained).

The *UpdateAdd* function returns a tuple containing: $C_{min}(i,i,fin,fout)=C_{min}(r,i,fin,fout)+C_{best}(i, parent(r,i),fout)$ ($1 \le fin,fout \le k$) and $C_{best}(i,i,fin)= c_{recv}(i,fin)+min\{C_{min}(i, i, fin, fout) | 1 \le fout \le k\}$. Thus, in $O(n \cdot k^2)$ time we can compute $C_{best}(i,i,f)$ for every vertex i of the tree and not just for one root.

B. First-Fit Online Tree Coloring

In [5,6], the first-fit online tree coloring was analyzed. A well-known heuristic for coloring a graph with a minimum number of colors is to consider the graph vertices in some order $v(1), \ldots, v(n)$ and color vertex $v(i)$ with the smallest color which was not used for coloring a neighbor $v(j)$ of

v(i) (with *j<i*). The maximum number of colors that can be used by this heuristic for the worst-case ordering of the vertices of a given tree was computed in [5,6]. In [6], the time complexity was *O(n·log(log(n)))*, using a technique similar to the one presented in this section. However, like in the case of the previous problem, the time complexity is valid only if each vertex has at most a constant number of neighbors. For the general case, we present an *O(n·log(n))* solution. In [5] it was proved that the maximum number of colors used by the heuristic is *C=O(log(n))* for any tree with *n* vertices. We will root the tree at a vertex *r* and, for each vertex *i*, we will compute *Cmax(r,i)*=the largest color that can be assigned to vertex *i* if we restrict our attention to *T(r,i)* only, and *Cnt(r,i,col)*=the number of sons *s(r,i,j)* of vertex *i* with *Cmax(r,s(r,i,j))=col*. For a leaf vertex *i*, *Cmax(r,i)=1* and *Cnt(r,i,col)=0* (*0≤col≤ C*). For a non-leaf vertex *i*, we initialize *Cnt(r,i,*)* to *0* and then traverse all of its sons. When considering a son *s(r,i,j)*, we increment *Cnt(r,i,Cmax(r,s(r,i,j)))* by *1*. Then, we can easily compute *Cmax(r,i)* based on the values *Cnt(r,i,*)*. We initialize *Cmax(r,i)* to *1* and then traverse the colors *col* in order, from *1* to *C*. If *col≥Cmax(r,i)*, we increment *Cmax(r,i)* by *min{col-Cmax(r,i)+1, Cnt(r,i,col)}*. This algorithm obviously takes *O(n·log(n))* time. In order to compute *Cmax(i,i)* for every vertex *i*, we will use the algorithmic framework we introduced. The *UpdateRemove* function returns a tuple containing: *Cnt(root,i,col)=(if (Cmax(r,root)=col) then Cnt(i,i,col)-1 else Cnt(i,i,col))* (*1≤col≤C*), and *Cmax(root,i)*, recomputed as described above, based on the values *Cnt(root,i,*)*. The *UpdateAdd* function returns: *Cnt(i,i,col)=(if (Cmax(i,parent(r,i))=col) then Cnt(r,i,col)+1 else Cnt(r,i,col))* (*1≤col≤C*), and *Cmax(i,i)*, recomputed as described above, using the values *Cnt(i,i,*)*.

VII. CONNECTED K-CENTER AND K-MEDIAN

Let's assume that we have a tree, rooted at a vertex *r*. Each tree edge *(parent(r,i),i)* has a weight *w(r, parent(r,i), i)*, such that *w(r,parent(r,i),i))>w(r, i, p)* (for any *i* and *p≠parent(r,i)*); that is, the weight of the edges *(parent(r,i), i)* are larger than the weights of the edges in *T(r,i)*. We want to find a connected subset *S* composed of *k* tree vertices, such that *r∈ S* and the sum (maximum) of the weights of the edges not connecting two vertices *u* and *v* from *S* is minimized. A connected subset composed of *k* vertices implies that there are *k-1* edges whose weights are not added to the sum (maximum). The best solution would be obtained if we could select the *k-1* edges with the largest weights. A problem that could occur is that the subset of the endpoints of the *k-1* largest edges is not connected. However, as it was proven in [8], it is impossible for this subset not to be connected. Let's assume that the subset is not connected. That means that there is an edge *(parent(r,u),u)* which is not part of the set of *k-1* edges, but some edges in *T(r,u)* were selected. This is impossible, because *w(r,parent(r,u), u)* is larger than the weights of the edges in *T(r,u)*. Thus, if we selected an edge in *T(r,u)*, we must have also selected the edge *(parent(r,u), u)*.

In order to select the *k-1* largest edges, we insert all the *n-1* tree edges into a max-heap *HT*, from which we extract the desired edges. Thus, in *O(n·log(n))* time, we can solve this problem. Let's assume that the problem is extended as follows. If we choose a different vertex *r'* as the root of the tree, the edges of the weights do not stay the same. However, they change according to the following restrictions. When considering as the new tree root a vertex *r'* which is a son of *r*, only the weight of the edge *(r',r)* changes (it does not necessarily increase, but its new weight is larger than all the weights of the edges located in *r*'s subtree). We will use the algorithmic framework introduced in the previous section. Let's assume that we have a min-heap *HS* which maintains all the edges in the solution for the vertex *r* as the root (the other edges are in *HT*). In the *UpdateRemove* function, if *(r,r')* is in *HS*, we remove it from *HS* and insert it in *HT*; otherwise, we remove the edge with the minimum weight from *HS* and insert it in *HT*. In the *UpdateAdd* function, we add the edge *(r',r)* to *HS*, considering its new weight (removing it from *HT*). For the min-sum case, we will also maintain the sum *esum* of the edges which are outside *HS*. Initially, *esum*=the sum of all the edges in *HT*. Whenever we remove (insert) an edge from (into) *HT*, we decrease (increase) *esum* by the (current) weight of that edge. At any time, the maximum weight of an edge in *HT* (for the min-max case) or the value of *esum* are the cost corresponding to the current root. Thus, in *O(n·log(n))* time, we can compute the optimal solutions for every vertex *i* as the tree root (the alternative would have been to root the tree independently at every vertex and recompute the solution from scratch every time – this would have taken $O(n^2 \cdot log(n))$ time). We will now show how the (unweighted) connected k-center (also discussed in Section IV) and a restricted version of the weighted connected k-median problem can be solved using the generic solution presented above, in *O(n·log(n))* time.

A. Unweighted Connected K-Center in Trees

We assume that every edge *(u,v)* of the tree has a length (e.g. delay, latency) *l(u,v)*. We want to find a connected subset *S* composed of *k* vertices, such that the maximum distance from a vertex outside of *S* to the nearest vertex in *S* is minimized. The distance from a vertex *u* to a vertex *v* is *d(u,v)*, the sum of the lengths of the edges on the (unique) path between *u* and *v*. The vertices in *S* need to form a connected subtree. We will root the tree at an arbitrary vertex *r* and we will want to compute a connected subset *S* of *k* vertices which contains *r*. For each vertex *i*, we compute *lmax(r,i)*=the length of the longest path starting at *i* and ending at a vertex in *T(r,i)*, and *lmax2(r,i)*=the length of the 2nd longest path starting at *i* and ending in *T(r,i)*. We assign a weight *we(r,parent(r,u),u)* to each edge *(parent(r,u),u)*:

we(r,parent(r,u),u)=l(parent(r,u),u)+lmax(r,u). It is obvious that the weight of the edge *(parent(r,u),u)* is larger than all the weights of the edge in *T(r,u)*. Furthermore, if we choose a connected subset of *k* vertices containing *r*, the maximum distance from a vertex outside of *S* to its closest vertex in *S* will be equal to the largest weight of an edge which does not connect two vertices in *S*. Thus, it is optimal to choose the *k-1* edges having the largest weights. When changing the root of the tree (a son *r'* of the previous root *r* is lifted as the new root), only the weight of the edge *(r',r)* changes. We will have *lmax(r',r)= lmax(r,r)* if the path corresponding to this value did not pass through the vertex *r'*; otherwise, *lmax(r',r)=lmax2(r,r)*. For *r'*, *lmax(r',r')* and *lmax2(r', r')* are the smallest two values in the (multi)set *{lmax(r,r'), lmax2(r,r'), l(r',r)+lmax(r',r)}*. The new weight of the edge *(r,r')* will be *l(r,r')+lmax(r',r)*. We can now easily use the algorithm presented previously. If *k=1*, then we don't need to use the heaps, as only the values *lmax(i,i)* are of interest, obtaining an *O(n)* solution for the unweighted tree center problem.

B. Connected K-Median in Trees

Each vertex *u* is enhanced with a weight *wv(u)* (e.g. expected amount of data transferred from the servers). The k-median problem asks for a connected subset *S* composed of *k* vertices, such that the total sum of weighted distances from the vertices outside *S* to the corresponding closest vertex in *S* is minimized. The weighted distance from a vertex *u* to a vertex *v* is *dw(u,v)=wv(u)·d(u,v)*, where *d(u,v)* is the sum of the lengths of the edges on the path between *u* and *v*.

For each vertex *i*, we will compute *wvt(r,i)*=the sum of the weights of the vertices in its subtree; *wvt(r,i)=wv(i)+wvt(r,s(r,i,1))+ … +wvt(r,s(r,i,ns(r,i)))* (if *i* is a leaf, *wvt(r,i)=wv(i)*). We will assign to each edge *(parent(r,i),i)* a weight *we(r,parent(r,i),i)= wvt(r,i)·l(parent(r,i),i)*. In the general case, this weight function does not imply the property required by the algorithm described previously (that *we(r,parent(r,i),i)* is larger than the weight of any edge in *T(r,i)*). We will restrict our attention to situations in which this property holds, e.g. when all the edge lengths are equal. The minimum cost criterion (for a subset containing the vertex *r*) implies that the sum of the weights of the edges not connecting two vertices in *S* must be minimized. Thus, we again want to choose the *k-1* edges with largest weights, which will form a connected subset of vertices. When lifting a vertex *r'* as the new root (above the previous root *r*), we have: *wvt(r',r)=wvt(r,r)-wvt(r,r')* and *wvt(r',r')=wvt(r,r)*. The new weight of the edge *(r',r)* is *wvt(r',r)·l(r',r)*. All the other edge weights and *wvt(*)* values stay the same. Thus, we can use the presented algorithm.

VIII. RELATED WORK

Center, diameter and longest path problems have been considered in [1,2,3], where efficient algorithms were proposed for several weighted and unweighted center problems on cacti. Our algorithms, however, are derived from the dynamic programming solutions on trees and may be more easily extended to other problems which have a dynamic programming solution on trees. In [8], a linear time algorithm for the unweighted connected k-center problem in trees was given. A framework for centrality problems on trees (which resembles the framework we proposed to some extent) was introduced in [9]. Center problems on path networks were considered in [10].

IX. CONCLUSIONS

In this paper we presented several novel algorithms for some center and median location problems in cacti, trees and paths. Some of the considered problems are new (connected k-median), while others have been considered previously, but the algorithms we proposed are either better or almost equally good, but easier to implement. We have also introduced an algorithmic framework for several optimization problems in trees (not just centers and medians), similar to that presented in [9]. The center and median problems we discussed are applicable to the optimal placement of storage servers in content delivery networks with tree-like topologies.

REFERENCES

[1]. K. Das and M. Pas, An Optimal Algorithm to Find Maximum and Minimum Height Spanning Trees on Cactus Graphs, Advanced Modeling and Optimization **10** (1), 2008, pp. 121-134.

[2]. Y.-F. Lan, Y.-L. Wang, and H. Suzuki, A Linear-Time Algorithm for Solving the Center Problem on Weighted Cactus Graphs, Information Processing Letters **71**, 1999, pp. 205-212.

[3]. B. Ben-Moshe, B. Bhattacharya, Q. Shi, and A. Tamir, Efficient Algorithms for Center Problems in Cactus Graphs, Theor. Comp. Sci. **378**, 2007, pp. 237-252.

[4]. M. I. Andreica, E.-D. Tirsa, C. T. Andreica, R. Andreica, and M. A. Ungureanu. Optimal Geometric Partitions, Covers and K-Centers, Proc. 9th WSEAS Conf. on Mathematics and Computers in Business and Economics, Bucharest, Romania, 2008, pp. 173–178.

[5]. S. M. Hedetniemi, S. T. Hedetniemi, and T. Beyer. A Linear Algorithm for the Grundy (Coloring) Number of a Tree, Congr. Numerantium **36**, 1982, pp. 351-362.

[6]. M. I. Andreica. Algorithmic Techniques for Several Optimization Problems Regarding Distributed Systems with Tree Topologies, ROMAI Journal, 2008. in press.

[7]. M. I. Andreica and N. Tapus. Offline Algorithmic Techniques for Several Content Delivery Problems in Some Restricted Types of Distributed Systems, Proc. 2nd Intl. Workshop on High Perf. Grid Middleware (HiPerGrid), Bucharest, Romania, 2008, pp. 65-72.

[8]. W. C.-K. Yen and C.-T. Chen, The Connected p-Center Problem with Extension, Proc. Joint Conf. on Information Sciences, Kaohsiung, Taiwan, 2006.

[9]. A. Rosenthal and J. A. Pino, A Generalized Algorithm for Centrality Problems on Trees, J. of the ACM, **36** (2), 1989, pp. 349-361

[10]. N. Halman, A Linear Time Algorithm for the Weighted Lexicographic Rectilinear 1-center Problem in the Plane, Inf. Proc. Lett. **86** (3), 2003, pp. 121-128.