Undiscretized Dynamic Programming: Faster Algorithms for Facility Location and Related Problems on Trees

Rahul Shah* Martin Farach-Colton†

Abstract

In the Uncapacitated Facility Location (UFL) problem, there is a fixed cost for opening a facility, and some distance matrix d that determines the cost of distributing commodities from any facility i to any consumer j. The problem is NP-hard in general and when d consists of a distance metric in a graph [7, 12]. However, for the case where the commodity transportation costs are given by path lengths in a tree, an $O(n^2)$ dynamic programming algorithm was given by [4, 7]. We improve this dynamic programming algorithm by using the geometry of *piecewise linear functions* and fast merging of the binary search trees used to store these functions. We achieve the complexity bound of $O(n \log n)$ for the Tree Location Problem and some related problems. Our approach gives a general method for solving tree dynamic programming problems.

1 Introduction

The UFL problem [3, 4, 7] has been studied extensively in location theory. The essence of the model is a trade-off between the facility placement cost and the transportation cost. The problem is to open a subset of facilities in order to minimize the total cost (or to maximize profit) while satisfying all demands. Consider a set of clients $I = \{1, .., m\}$ and a set of sites $J = \{1, .., n\}$ where the facilities can be located. An instance of the problem is specified by integers m and n, an $m \times n$ transportation cost matrix $C = \{c_{ij}\}$ and an n-dimensional facility setup cost vector $f = \{f_j\}$, such that $f_j \geq 0$. For any set S of facilities, it is optimal to serve a client i from a facility j for which c_{ij} is minimum over all $j \in S$. Thus, given S, the cost of S is $\sum_{i \in I} \min_{j \in S} c_{ij} + \sum_{j \in S} f_j$. The problem is to find a set S so that the cost is minimized. In [4, 7], an integer linear programming formulation for this problem is given. This problem, in general, was shown to be NP-hard [7, 12] by reduction to vertex cover. But on trees, various polynomial algorithms were given by [1, 4, 7].

When defining this problem on trees (or graphs), we take the set of clients and sites to be the entire vertex set V. Let $T = (V, E)$ be a tree with vertex set V and edge set E. The cost of opening a facility at vertex j is f_j. Each edge $e \in E$ has a nonnegative length. The distance d_{ij} between any pair of vertices i and j is the length of the shortest path for graphs. We also associate a nonnegative weight w_i with each vertex i so that the matrix C becomes a $|V| \times |V|$ matrix with $c_{ij} = w_i d_{ij}$. The problem is to select a subset $S \subseteq V$ of open facilities minimizing the following objective,

$$\sum_{j \in S} f_j + \sum_{i \in V} \min_{j \in S} w_i d_{ij}$$

This model is as formulated in [1, 3]. Note that the solution set S for the problem defines the partitioning of the tree into smaller subtrees. Each subtree corresponds to the tree induced by vertices served by a particular facility. Hence, the UFL problem on trees is often taken as a special case of the tree partitioning problem (in which each possible subtree has a weight) as in [4, 7]. Both these papers give an $O(n^2)$ algorithm for the tree partitioning problem. UFL is also shown to be a special case of the generalized (where facilities have a cost and the problem is to place $\leq k$ facilities) formulation of the k-median problem in [1] where $k \geq n$. Tamir's [1] dynamic programming functions also give an $O(n^2)$-time dynamic programming algorithm for UFL.

We first show how to replace the "discrete" dynamic programming functions of [1] with continuous functions – the so-called "undiscretization" of [2]. Shah, Langerman and Lodha [2] dealt with locating filters in a multicast tree. This has dynamic programming solution analogous to the facility location problem on rooted, directed trees where facilities can only serve nodes at lower levels. They show the undiscretized representation of monotonic function as a piecewise linear function and show how to quickly add two such functions and probe them. In our case, there are two functions involved in dynamic programming and the computation of new functions is more complex than simple additions. Also, the corresponding "undiscretized" operations as

*email:sharahul@paul.rutgers.edu. Dept. of Computer Science, Rutgers University, NJ 08854

†email:martin@google.com. Google, Inc. CA 94043 and Dept. of Computer Science, Rutgers University, NJ 08854

in [2] do not maintain the consistency of these functions in our case. If we were to use these directly, it would give an $O(n^2)$-time algorithm due to additional operations to ensure consistency. Instead, we modify the dynamic programming functions of [1] so that the functions involved are either convex or concave. Further, we extend the "undiscretization" techniques and operations to convex and concave functions such that the consistency is maintained. These functions have succinct representations and can be quickly updated to construct new functions. We achieve an upper bound of $O(n \log n)$.

In section 2, we define our notation and describe the $O(n^2)$ dynamic programming algorithm for the problem given by [1]. We also show the "undiscretization" of the dynamic programming functions using the techniques of [2] and prove some necessary lemmas. In section 3 we describe the algorithm and derive its complexity. In section 4, we describe the data structures used and the operations involved. In section 5, we describe related problems on trees where our methods apply and in section 6 we give concluding remarks.

2 Preliminaries and Dynamic Programming Functions

We shall regard the tree $T = (V, E)$ as a rooted tree with an arbitrarily chosen root node R. If the tree is a non-binary tree, it can be converted into a binary tree in linear time using a technique of [1]. This is done by splitting each vertex with $k > 2$ children into $k-1$ vertices, with edges joining them having distance zero and facility placement cost f_j for each newly introduced vertex being ∞. This at most doubles the number of vertices, hence does not affect complexity. Hence, for the rest of the paper, we shall assume that the tree is binary. Let $|V| = n$ and $|E| = n-1$. T_v denotes the subtree rooted at the vertex v. The *size* of a tree is the number of its nodes. We denote the size of T_v as s_v.

Let $G_v(x)$ be the minimum objective function value of the subproblem defined on the subtree T_v such that there is at least one facility in T_v within distance x from v. Let $F_v(x)$ be the minimum objective function value of the subproblem defined on T_v such that the nearest facility in $T - T_v$ from v is exactly at distance x from v. Note that $G_v(x)$ is a step-wise decreasing function of x (i.e. it is a piecewise linear function with the slope of each piece equal to zero.). There is a breakpoint at $G_v(x')$ if x' is the distance from v to some node $u \in T_v$, and the solution that realizes the objective function value of $G_v(x')$ has u as the facility serving v. Hence, each breakpoint in $G_v(x)$ corresponds to some unique node in T_v. $G_v(\infty)$ is the minimum value $G_v(x)$ can achieve. $G_R(\infty)$, where R is the root, is the final value of objective function we are interested in minimizing. $F_v(x)$ is a piecewise linear non-decreasing concave function of x. $F_v(\infty) = G_v(\infty)$.

For any piecewise linear function F, let the *size* of F, denoted $|F|$, be the number of breakpoints in F. Let $x_1, x_2, ..., x_k, ..., x_{s_v}$ be the distances of vertices in T_v from v in increasing order. Let l and r be left child and right child of v, separated from v by distance x_l, x_r respectively. The dynamic programming algorithm of Tamir [1] stores, at each vertex v, the values of $G_v(x)$ and $F_v(x)$ for $n-1$ distinct values of x corresponding to the distances of all other vertices in T from v. So, the storage space at each node in their algorithm is $O(n)$. They show how to compute the ($n-1$ discrete) values of $G_v(x)$ and $F_v(x)$ in $O(n)$ time at each node. Hence, their algorithm runs in $O(n^2)$ time.

By "undiscretizing" the representation of G_v and F_v we make these functions invariant of the distances of v from all vertices u which are not in the subtree T_v. In the following lemmas, we shall show that this representation of G_v and F_v takes $O(s_v)$ space. Now, if the tree T is a balanced binary tree and the computation at each vertex v is linear in the space required to store each function, we would get an $O(n \log n)$-time algorithm. However, that may not be the case, so we design a data-structure along with operations on it, that allows us to compute F_v and G_v in $O(s_l \log((s_l + s_r)/s_l))$, assuming wlog that $s_r \geq s_l$. Roughly, the computation at each node is linear in the size of its smaller subtree and logarithmic in the size of its larger subtree. This leads to an $O(n \log n)$-time algorithm over any tree. We present the following recurrence relations which show the computation of the "undiscretized" functions G_v and F_v. This are simply Tamir's [1] dynamic programming recurrences written in "undiscretized" parameter x.

if v is a leaf **then**
  $G_v(x) = f_v$ and $F_v(x) = \min\{w_v x, f_v\}$
else
  $G_v(0) = f_v + F_l(x_l) + F_r(x_r)$
  $G_v(x_k) = min\{G_v(x_{k-1}), w_v x_k + G_l(x_k - x_l) + F_r(x_k + x_r)\}$
    whenever x_k corresponds to a distance between v and a vertex in T_l
  $G_v(x_k) = min\{G_v(x_{k-1}), w_v x_k + G_r(x_k - x_r) + F_l(x_k + x_l)\}$
    whenever x_k corresponds to a distance between v and a vertex in T_r
  $G_v(x) = G_v(x_k)$ whenever $x_k < x < x_{k+1}$
  $F_v(x) = \min\{G_v(\infty), w_v x + F_l(x + x_l) + F_r(x + x_r)\}$.
end if

LEMMA 2.1. *F_v is a piecewise linear non-decreasing concave function (PLNCF) with size $|F_v| \leq s_v$.*

Proof. By induction on the height of v. For leaf v, $F_v(x) = \min\{w_v x, f_v\}$ is initially an increasing linear function with slope w_v, eventually becoming a constant function f_v. It has exactly one breakpoint at $x = f_v/w_v$. For internal node v with children l and r, $w_v x + F_l(x + x_l) + F_r(x + x_r)$ is a summation of three PLNCFs whose number of break points are not more than $0, s_l, s_r$, respectively, by induction. Since the sum of PLNCFs is a PLNCF whose number of breakpoints is at most the sum of the original two, $|w_v x + F_l(x + x_l) + F_r(x + x_r)| \leq s_l + s_r$. Taking the minimum of this function with a constant $G_v(\infty)$ will add at most one more breakpoint, and still maintain the PLNCF property. Hence, $|F_v| \leq s_v$.

□

The new breakpoint added due to taking the minimum of PLNCF with a constant function is said to correspond to v. Thus, it clear from the above proof that each breakpoint in F_v corresponds to a unique vertex in T_v.

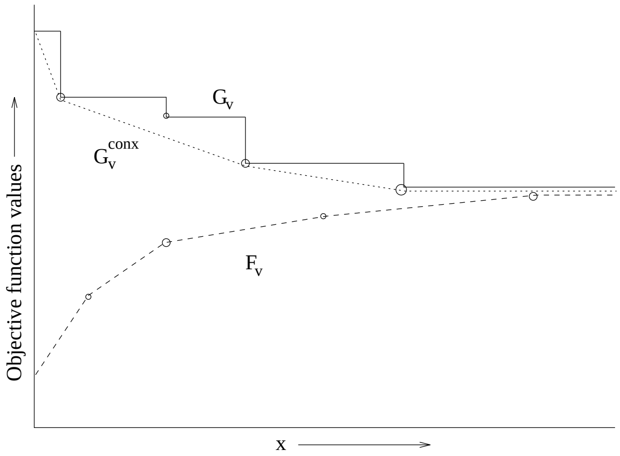
LEMMA 2.2. *G_v is a piecewise non-increasing step function(PDSF) with number of steps $|G_v| \leq s_v$.*

Proof. The only values of x where $G_v(x)$ can change value are when x equals distance of v from some vertex in T_v. So it is a piecewise step function and $|G_v(x)| \leq s_v$. Also, from the definition of G_v and the dynamic programming recurrence we get that G_v is non-increasing.

□

Define $G_v^{conx}(x)$ to be the convex hull function of $G_v(x)$. That is, G_v^{conx} is a convex function such that $\forall x G_v^{conx}(x) \leq G_v(x)$ and any convex function $H(x)$ such that $\forall x H(x) \leq G_v(x)$ satisfies $\forall x H(x) \leq G_v^{conx}(x)$.

Note that G_v^{conx} is a piecewise linear non-increasing convex function (PLDXF). The set of breakpoints of G_v^{conx} is a subset of breakpoints of G_v. Thus, the number of breakpoints $|G_v^{conx}| \leq |G_v| \leq s_v$. Also, it is clear from the proof of the previous lemma that each breakpoint in G_v corresponds to some vertex in T_v. From the dynamic programming recurrence relations it is clear that each breakpoint in G_v comes either from breakpoints of G_l or G_r or the vertex v itself for $G_v(0)$. The figure illustrates these three functions.



LEMMA 2.3. *For any breakpoint at y in G_l (or G_r) that is not in G_l^{conx} (or G_r^{conx}), there will not be a corresponding breakpoint at $y + x_l$ in G_v^{conx}.*

Proof. G_v would possibly have the breakpoint $y + x_l$ corresponding to breakpoint y in G_l with $G_v(y + x_l) = w_v(y + x_l) + F_r(y + x_l + x_r) + G_l(y)$. Since y does not belong to the breakpoints of G_l^{conx}, there are two breakpoints t, u in G_l such that $t < y < u$ and $G_l(y) > \frac{(u-y)G_l(t)+(y-t)G_l(u)}{u-t}$. That means that the point $(y, G_l(y))$, lies above the line formed by points $(t, G_l(t))$ and $(u, G_l(u))$. Since $H(x) = w_v(x + x_l) + F_r(x + x_l + x_r)$ is a concave function of x, $H(y) \geq \frac{(u-y)H(t)+(y-t)H(u)}{u-t}$. Hence, summing up, $G_v(y + x_l) > \frac{((u+x_l)-(y+x_l))G_v(t)+((y+x_l)-(t+x_l))G_v(u)}{(u+x_l)-(t+x_l)}$. So $y+x_l$ is not a breakpoint in G_v^{conx}.

□

LEMMA 2.4. *At each vertex v, computing G_v^{conx} instead of G_v is sufficient to carry on the recursion and G_R^{conx} is sufficient for computing the minimum objective function for T.*

Proof. For any vertex v, $G_v^{conx}(\infty) = G_v(\infty)$ and since $G_R(\infty)$ is the final value we are interested in, it is sufficient to compute G_R^{conx}. Now we only need to show how to compute G_v^{conx}, given $G_l^{conx}, G_r^{conx}, F_l$ and F_r where l and r are left and right children of some node v. If v is a leaf, then $G_v^{conx}(x) = G_v(x) = f_v$. Given the previous lemma, we compute G_v^{conx} by taking $H(x)$ as the convex hull function of $min\{w_v x + G_l^{conx}(x - x_l) + F_r(x + x_r), w_v x + G_r^{conx}(x - x_r) + F_l(x + x_l)\}$ and then making it non-increasing by taking a break point t where $H(x)$ achieves minimum and defining $G_v^{conx}(x) = H(x)$ for all $x \leq t$ and $G_v^{conx}(x) = H(t)$ for all $x > t$.

□

UFL$(v)\{$
  if v is a leaf **then**
    $G_v^{conx} = createG(f_v)$;
    $F_v = createF(w_v, f_v)$;
  else
    $(G_l^{conx}, F_l) = UFL(l)$;
    $(G_r^{conx}, F_r) = UFL(r)$;
    $g^0 = f_v + probeF(F_l, x_l) + probeF(F_r, x_r)$;
    $G^1 = add\_dissolveFinG(G_l^{conx}, F_r, x_l, x_r)$;
    $G^2 = add\_probeFforG(G_r^{conx}, F_l, x_r, x_l)$;
    $G^2 = add\_point(G^2, (0, g^0))$;
    $G^3 = min\_mergeG(G^1, G^2)$;
    $G_v^{conx} = add\_lineG(G^3, w_v)$;
    $g^{inf} = probeG(G_v^{conx}, \infty)$;
    $F_v = add\_merge(F_l, F_r, x_l, x_r)$;
    $F_v = add\_line\_pruneF(F_v, w_v, g^{inf})$;
  end if
  return (G_v^{conx}, F_v);
$\}$

Table 1: Undiscretized Algorithm

3 Algorithm and Analysis

We are now ready to present the algorithm to compute the functions defined in section 2. We describe UFL(v) which is a recursive procedure that returns (G_v^{conx}, F_v). Recall that l and r are the left and right children of v at distances x_l and x_r respectively. Wlog, we assume $s_l \geq s_r$.

3.1 Algorithm. The algorithm presented in table 1 above is nothing but a translation of the dynamic programming recurrences shown in section 2. The procedures in the algorithm mainly perform four functions: create new PLFs with unit size, make a unit update in the existing PLF, evaluate a PLF at some point or add two PLFs. Depending on the relative sizes and the types of PLFs, these operations need to be carried out differently.

3.2 Data Structure Operations. Here we describe each of the operations used in the algorithm above and give their running times. The corresponding data structure operations are described in section 4.

$createG(c)$: returns a constant PLDXF with value identically c for all x. Running time $O(1)$.

$createF(d, c)$: returns a PLNCF with exactly one breakpoint at $x = c/d$. The slope of the first line segment from 0 to c/d is d and the slope of the line segment from c/d to ∞ is 0. The running time is $O(1)$.

$probeG(G, t)$: takes the PLDXF G and a value t as parameters and returns the y value of the breakpoint in G just less than t. Running time $O(\log |G|)$.

$probeF(F, t)$: takes the PLNCF F and a value t as parameters and returns the value $F(t)$. Running time $O(\log |F|)$.

$add\_line\_pruneF(F, d, c)$: adds a linear function with slope d to PLNCF F and then finds the point of intersection t of PLNCF F with constant c, and makes $F(x) = c$ for all $x \geq t$. Running time $O(\log |F|)$ plus time for deleting all breakpoints $u > t$ in F.

$add\_lineG(G, d)$: adds a linear function with slope d to PLDXF G. Then prunes the function as required to restore non-increasing behavior. Running time $O(\log |G|)$.

$add\_point(G, (t, u))$: inserts a new breakpoint t with function value u into a PLDXF G and then restores convexity by deleting points in neighborhood of t if necessary. Running time $O(\log |G|)$ plus time for deletions.

$add\_merge(F_1, F_2, t_1, t_2)$: adds two PLNCFs F_1, F_2 shifted back by values t_1, t_2 respectively. Running time $O(|F_2| \log \frac{|F_1|+|F_2|}{|F_2|})$.

$min\_merge(G_1, G_2)$: lists the breakpoints of PLDXF G_2 and inserts them along with their function values into PLDXF G_1 sequentially in increasing order, restoring convexity after each insertion by deleting few points if necessary. Returns G_2. Running time $O(|G_2| + |G_2| \log \frac{|G_1|+|G_2|}{|G_2|})$ plus the time for deletions.

$add\_probeFforG(G, F, t_g, t_f)$: lists all breakpoints in PLDXF G and shifts them forward (add) by $t_g + t_f$. Sequentially probes PLNCF F at these values and adds the return value to the function value at breakpoints in G. Shifts them backwards (delete) by t_f. Now, with these points in sorted order, takes the convex hull and generates a new PLDXF. Running time $O(|G| + |G| \log \frac{|F|+|G|}{|G|})$.

$add\_dissolveFinG(G, F, t_g, t_f)$: inserts the linear segments in PLNCF F sequentially in PLDXF G, adding the linear value to breakpoints in G. It also checks and restores convexity around each breakpoint of F. Running time $O(|F| + |F| \log \frac{|G|+|F|}{|F|})$ plus time for deletions.

3.3 Analysis. Here, we show that our algorithm solves the UFL problem on a tree in $O(n \log n)$. The time required by functions *createF* and *createG* is constant per leaf. Hence the total time for these operation over the entire algorithm is $O(n)$. The time required for each of $probeG, probeF, add\_line\_pruneF, add\_lineG$ and *add_point* operations is bounded above by $O(\log n)$ and each operation is carried out at most once at each vertex v. Hence, the time taken by these operation over the entire algorithm is bounded above by $O(n \log n)$. In operations involving deletions, the time taken is $O(\log n)$ per deletion. Once the breakpoint is deleted it never re-enters the data structure. Hence the total number of deletions is bounded above by $2n$ (for F and G) and the total cost of deletion is $O(n \log n)$ over the entire algorithm. What remains to be shown is that the total cost of "merge" operations $min\_merge, add\_merge, add\_probeFforG$ and $add\_dissolveFinG$ is bounded by $O(n \log n)$.

THEOREM 3.1. *The total time required to compute all the "merge" operations in T_v (in UFL(v)) is $O(s_v \log s_v)$.*

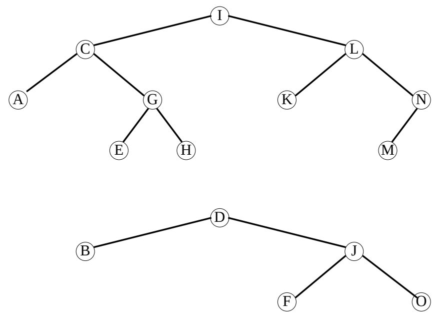
Proof. By induction on height of v. If v is a leaf then in $UFL(v)$, there are no "merge" operations, so the claim is true. Note that for any $x \geq y > 2$, $O(y + y\log((x + y)/y))$ is asymptotically same as $O(y\log((x + y)/y))$. So, for any internal node v with left child l and right child r, with $s_l \geq s_r$ by induction we get that the total cost of "merge" operations is $O(s_l \log s_l) + O(s_r \log s_r) + O(s_r \log((s_l + s_r)/s_r))$. This is bounded above by $O(s_v \log s_v)$.

□

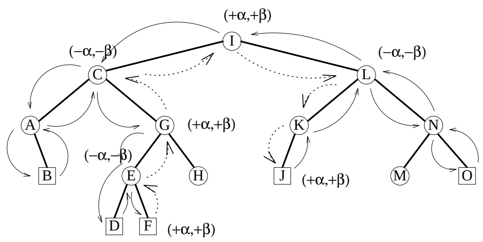
4 Data Structures

Here, we describe the data-structures used to store the functions F_v and G_v^{conx} which are PLNCF and PLDXF respectively. The main data-structure is a height balanced binary search tree. We shall use AVL trees which can be merged fast using Brown and Tarjan's fast merging algorithm [8].

4.1 Fast Merging Algorithm. Brown and Tarjan [8] described the algorithm to merge two binary search trees which represent ordered lists. They use AVL trees which are height balanced. If T_1 and T_2 are AVL trees representing sorted lists of m and n elements respectively, with $m \geq n$, they insert the elements from T_2 into T_1 in sorted order to obtain a new AVL tree with $m + n$ elements. Rather than doing each insertion independently of the others by starting each search from the root, the search for the insertion of a new element is started from the position of previously inserted element, climbing up to the first ancestor(LCA) having the next element to search in its subtree, and continue searching down the tree from there. Brown and Tarjan show this can be done in $O(n \log((m + n)/n))$. It is easy to show that the upper bound of the length of the walk performed during the insertions of n sorted elements is indeed $O(n \log((m + n)/n))$. This is done by considering the distance traveled in two parts, one that is within the top $\log n$ levels of AVL tree and the other which is within the bottom $\log((m + n)/n)$ levels of the tree. For $m \geq n \geq 2$ both of these are bounded above by $O(n \log((m + n)/n))$. They additionally show how to maintain the height balance during these operations. Also n values, given in sorted order, can be accessed (searched) in the tree containing m nodes in $O(n \log(m/n))$ time by the same algorithm.



Sorted lists represented as height-balanced trees



Merging by sequential insertions (square nodes have been inserted)

4.2 Representation of PLNCF. For storing the PLNCF F we will maintain the breakpoints sorted by their x coordinate in an AVL tree. Along with the x coordinate of the breakpoint each node also contains two numbers a and b such that the linear segment in PLNCF to the left of this breakpoint has the equation $y = Ax + B$ where A (resp. B) is the sum of all the a (resp. b) values on the path from the node to the root of the tree. Along with this, we also store a number x^{off} which records the offset of the x values within the tree. The actual x coordinate of a breakpoint is its x coordinate stored in the data structure node minus

x^{off}. The function value $F(x)$ is given by $y = Ax' + B$ where $x' = x + x^{off}$ and A, B represent the equation of the line passing through x' in the data structure. Note that given the breakpoints and equations of the line segments joining them in sorted order, we can construct the data structure for F in linear time of size i.e. $O(|F|)$. Similarly, given the data structure representing F we can list all the breakpoints and equations of lines in $O(|F|)$. Given this, we describe how the operations in 3.2 are carried out.

createF(d, c): Create an AVL tree with singleton node, with $x = c/d, a = d, b = 0$. Set $x^{off} = 0$.

probeF(F, t): Let $t' = t + x^{off}$. Search for t' in the data structure and reach the node at coordinate u in the data structure such that $u \geq t'$ and there is no breakpoint s such that $u > s \geq t'$ in the data structure. Let A and B be the sums of the a and b values from root to u. These values can be computed along the search path. Return $At' + B$. If there is no such value u then access (search) the rightmost breakpoint and return its y value with obtained by A, B, x values at that breakpoint.

add_line_pruneF(F, d, c): Shift the equation of the line. The slope remains the same, but the y-intercept instead of zero is now $-dx^{off}$. So add the tuple $(d, -dx^{off})$ to tuple (a, b) at the root. The slope of the rightmost (infinite) line segment, assumed to be 0 by the data structure, is no longer zero, but d. To make it consistent and correct, prune the function at $y = c$. For this, search the breakpoint with smallest x (leftmost) value starting from root with its y value bigger than c. This search can be carried out in the same way as an AVL search because y monotonically increases with x. Then, set the x value of this breakpoint to $(c - B)/A$ where A, B are sum of a, b values from root to this breakpoint. Now delete all the breakpoints from the data structure with $x > (c - B)/A$.

add_merge(F_1, F_2, t_1, t_2): Assume $|F_1| \geq |F_2|$. In F_1 set $x_1^{off} = x_1^{off} + t_1$. Delete all breakpoints in F_1 with $x < x_1^{off}$. Similarly, in F_2 set $x_2^{off} = x_2^{off} + t_2$ and delete breakpoints similarly. Before adding F_1 and F_2, we need to align their offsets. Since $|F_2| \leq |F_1|$, list all the breakpoints of data structure for F_2 and list all the equations of the line segments in increasing order. We subtract $(x_2^{off} - x_1^{off})$ from each x coordinate and for each line segment $Ax + B$, we add $A(x_2^{off} - x_1^{off})$ to B. With this transformation the offset of F_2 is same as that of F_1. Now, use Brown and Tarjan's Fast Merging Algorithm to insert the breakpoints of F_2 in F_1. When a breakpoint u is inserted, we have to add the equation of line segment $y = \alpha x + \beta$ on the left of u to all points in the data structure between u and the previously inserted point s. This can be done by updating the a, b values along the walk from s to u performed during the merge algorithm. To do this, add tuple (α, β) to (a, b) values at the $LCA(s, u)$. Then, on the path from $LCA(s, u)$ to s, whenever we choose a right child after a (non empty) series of left children, subtract (α, β) from the node where the decision is made and add the (α, β) in the vice-versa case. On the path from $LCA(s, u)$ to u, do the same thing reversing the sense of left and right. For completeness sake, we state that the values of (a, b) at the nodes can be preserved during rotation and double-rotation operations involved in AVL insertions and deletions. The offset of the new PLNCF is same as that of F_1.

4.3 Representation of PLDXF. Here, again we maintain the breakpoints of PLDXF G in the AVL tree. Also, the tuple (a, b) is stored along with x value. However, unlike PLNCF, it doesn't represent the equation of line-segment to the left. In fact, in this case, it is only used to obtain the y value (same as $G(x)$) at a particular breakpoint. The value is calculated as $y = Ax + B$ where A, B are same as in the previous subsection. x^{off} is defined similarly, except that it records the addition required to the x values in the data structure to reflect the correct x values. PLDXF G can be listed and constructed from the list in linear time, as in the case of PLNCF.

createG(c): Create an AVL tree with a singleton node. Set $x = 0, a = 0, b = c$. Set $x^{off} = 0$.

probeG(G, t): $t' = t - x^{off}$. Search for t' in the AVL tree and reach the breakpoint $u \leq t'$ such that there is no breakpoint s with $u < s \leq t'$. Return the y value at u calculated as $Au + B$ where A, B are sums of a, b values along the path from root to the breakpoint at u.

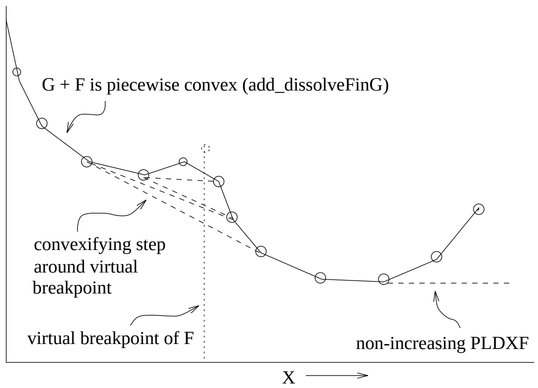
add_line(G, d): Take the y intercept of the line as $+dx^{off}$. Add (d, dx^{off}) to the tuple (a, b) at the root. Now, to ensure non-increasing character, we delete points from behind (right to left) till we reach a point u, the point to the left of which has higher y value. Then, we do not delete u and halt.

add_point$(G, (t, u))$: Insert a breakpoint at $t - x^{off}$. Calculate the inherited y value u' at this point. Set $a = 0, b = u - u'$. Now, having inserted this point we need to maintain convexity and non-increasing property. Check left and right neighbors of this point in sorted order. If this point lies above the line formed by joining these neighbors then delete the newly inserted point and return. If not then from this newly inserted point we go rightwards and delete all the points which have y values higher than u. Now, we traverse leftwards in the AVL tree and check the points in decreasing order of x coordinates. We keep track of slopes of segments formed by adjacent pairs of points. In the case of convex functions, the slope (which is negative always) should decrease as we move leftwards. If we find that the slope increased then we delete the breakpoint to the right of that segment. And we continue, till we find the decreasing slope. Then we stop. If the inserted point is a leftmost point then we do the similar convexifying step towards the right. In this procedure, there are only a constant number of more accesses than the number of deletions. We charge the cost of access of the deleted point to the deletion operation. So, the time taken by this procedure is same as the time taken for access, which is $O(\log|G|)$.

$min\_merge(G_1, G_2)$: Assume $|G_1| \geq |G_2|$. Assume offset x_2^{off} of $|G_2|$ is 0. List all the points in $|G_2|$ in increasing order of x with their x and y values. Subtract offset x_1^{off} of G_1 from all x values. Now using Brown and Tarjan's algorithm, insert these points into the AVL tree representing G_1 along with their y values as in *add_point* and also perform the convexifying step around each insertion. The offset of the new PLDXF is the same as that of G_1. Again, we access only a constant number of extra undeleted points per insertion. Also, these accesses are in the neighborhood of newly inserted points. Charging the cost of accessing deleted points to deletion, it can be shown that the total cost is $O(|G_2| + |G_2|\log((|G_1| + |G_2|)/|G_2|))$ plus the cost of deletion.

$add\_probeFforG(G, F, t_g, t_f)$: We first list all the breakpoints of G in increasing order with their x and y values. We then add t_g to each x value. For each breakpoint x in G, we check the values of $probeF(F, x + t_f)$ and add them to their corresponding y values in G. Now, we keep only those points in G which form a convex function. Since the points are already sorted, the convex hull can be computed in linear time. For sequential probes in F we again use Brown and Tarjan's algorithm.

$add\_dissolveFinG(G, F, t_g, t_f)$: We list all the breakpoints in F along with the equations of segments and transform them accordingly as in *add_merge* considering the values x_g^{off}, x_f^{off}, which are x offsets of G, F respectively, and t_g, t_f. The offset of the new data structure will be same as that of G. Now we virtually insert the breakpoints of F and actually insert linear segments of F into G. By this we mean that we do update the (a, b) values along the Brown and Tarjan's Merging walk performed during the algorithm but do not actually insert points. However, we remember the locations of each virtually inserted breakpoint of F in G. There could be a possible region of concavity around this virtual breakpoint. Again we apply a convexifying step around these virtual breakpoints to make G convex and non-increasing. The figure illustrates convexifying step involved.



5 Related Problems

Several generalizations of the UFL problem on trees have been proposed. The tree partitioning problem by Cornuejols, Nemhauser and Wosley [7] was shown to be a generalization of the Economic Lot Sizing (ELS) problem as well as of UFL. Shaw [4] gives the tree partitioning generalization of UFL, Facility Constrained Covering (FCC) problem, Customer Constrained Covering (CCC) problem and Generic Customer Covering (GCC) problem. They give $O(n^2)$ algorithms for solving all these problems on trees. As noted earlier the tree partitioning problem differs from the UFL problem on trees in the sense that the transportation cost can be arbitrary and not linear, in particular with tree distances. Since the problem size involved in Tree Partitioning is $O(n^2)$, our algorithm can hardly hope to beat $O(n^2)$. The same is true in the case of GCC, where each customer has a specified subtree in which a facility is needed in order to cover that customer. However, our technique applies well to FCC, CCC and ELS, giving a time complexity of $O(n \log n)$ for each. Also, related is the problem of placing filters in a multicast tree for which an $O(n \log h)$ algorithm was given by Shah, Langerman and Lodha [2] which motivates our work. We briefly outline these problems and the "undiscretized" functions which can be used to solve them.

5.1 Facility Constrained Covering Problem.

This problem was first studied by Kolen [3]. In this problem, there exists a radius s_j for each facility j which has a set-up cost of f_j. A customer i can be served by a facility j only if the distance d_{ij} between them is at most s_j. If a customer i is not served by any facility, then a penalty cost of q_i is incurred. Here, for each v we define $G_v(x)$ as the optimal subproblem value in subtree T_v assuming that there is at least one facility in T_v whose radius of influence covers at least distance x beyond v in $T - T_v$. And we define $F_v(x)$ as the optimal subproblem value in subtree T_v assuming that the distanced covered in T_v by the *most influential* facility in $T - T_v$ is exactly x. Here G_v is a stepwise increasing function and F_v is a stepwise decreasing function of x, with $G_v(0) = F_v(0)$. Similar recurrences hold and the data structure using the fast merging of BSTs can be used to give an $O(n \log n)$ algorithm. The data structure operations are much simpler here since slopes and convexity issues need not be handled.

5.2 Customer Constrained Covering Problem.

This problem is also due to Kolen [3] and it differs from the FCC in that instead of a radius for facility, there is a radius of attraction r_i for each customer c_i. Here, we define G_v and F_v in exactly the same way as in the UFL problem in this paper. In this case, G_v is a stepwise decreasing function and F_v is a stepwise increasing function with $G_v(\infty) = F_v(\infty)$. As in FCC, we get an $O(n \log n)$ algorithm.

5.3 Economic Lot Sizing Problem [7].

There is demand d_i in period i, $i = 1, .., n$. The fixed cost of producing in period j is f_j and the variable cost is p_j. The variable storage and backorder costs are c_j^+ and c_j^-. This problem can be seen as UFL on a path, with the distance function on each edge being c_j^+ in one direction and c_j^- in the other. $G_v(x)$ and $F_v(x)$ can again be similarly defined with the minor modification that x for G_v means upward distance while x for F_v means downward distance. The variable costs p_j's are seen as the distances of leaves attached to the path.

6 Remarks and Future Work

Another generalization of UFL was given by Tamir [1] which has UFL as a particular case of the general model for k-median problem. Again, the dynamic programming functions here can be undiscretized but this involves two parameters, and an effective data structure for handling this is not known. [1] gives an $O(kn^2)$ algorithm for the k-median problem. The number of facilities opened in UFL can be controlled by varying the facility costs. In this sense, faster algorithms for UFL can in effect lead to faster algorithms for k-median on trees.

References

[1] A. Tamir, "An $O(pn^2)$ algorithm for the p-median and related problems on tree graphs", *Operations Research Letters, 19:59-94*, 1996.

[2] R. Shah, S. Langerman and S. Lodha,"Algorithms for efficient filtering in content-based multicast", *To appear in 9th annual European Symposium on Algorithms (ESA)* , 2001.

[3] A. Kolen, "Solving covering problems and the uncapacitated plant location problem on trees", *European Journal of Operations Research*, 12, 266-278, 1983.

[4] D. Shaw,"A unified limited column generation approach for facility location problems on trees", *Annuals of Operations Research*, 87, 363-382, 1999.

[5] R. Hassin and A. Tamir, "Improved complexity bounds for location problems on the real line", *Operations Research Letters*, 10, 395-402, 1991.

[6] A. Tamir and T. Lowe, "The generalized p-forest problem on a tree network", *Networks* 22, 217-230, 1992.

[7] G. Cornuejols, G.L. Nemhauser and L.A. Wosley, "The uncapacitated facility location problem", in P.B. Mirchandani and R.L. Francis(eds), *Discrete Location Theory*, Wiley, New York, 1990, pp. 119-171.

[8] M. Brown and R. Tarjan, "A fast merging algorithm",*Journal of ACM, 26(2), pp 211-225*, Apr 79.

[9] A. Aho, J. Hopcroft and J. Ullman, "The design and analysis of computer algorithms", *Addison-Wesley, Reading, Mass*, 1974

[10] G. Adel'son-Vel'skii and Y. Landis, "An algorithm for the organization of information", *Dokl. Akad. Nauk SSSR 146*, 263-266, (in Russian) English translation in *Soviet Math. Dokl.*, 3-1962, pp1259-1262.

[11] C. Crane, "Linear lists and priority queues as balanced binary trees", *PhD Thesis, Stanford University*, 1972.

[12] M. Garey and D. Johnson, "Computers and intractability: A guide to the theory of NP-completeness", Freeman, San Francisco, California, 1979.