

# Algorithms for Efficient Filtering in Content-Based Multicast

Stefan Langerman, Sachin Lodha, and Rahul Shah

Department of Computer Science,  
Rutgers University, New Brunswick, NJ, USA  
{lfalse,lodha,sharahul}@cs.rutgers.edu

**Abstract.** Content-Based Multicast is a type of multicast where the source sends a set of different classes of information and not all the subscribers in the multicast group need all the information. Use of filtering publish-subscribe agents on the intermediate nodes was suggested [5] to filter out the unnecessary information on the multicast tree. However, filters have their own drawbacks like processing delays and infrastructure cost. Hence, it is desired to place these filters most efficiently. An  $O(n^2)$  dynamic programming algorithm was proposed to calculate the best locations for filters that would minimize overall delays in the network [6]. We propose an improvement of this algorithm which exploits the geometry of *piecewise linear functions* and fast merging of sorted lists, represented by height balanced search trees, to achieve  $O(n \log n)$  time complexity. Also, we show an improvement of this algorithm which runs in  $O(n \log h)$  time, where  $h$  is the height of the multicast tree. This problem is closely related to  $p$ -median and uncapacitated facility location over trees. Theoretically, this is an uncapacitated analogue of the  $p$ -inmedian problem on trees as defined in [9].

## 1 Introduction

There has been a surge of interest in the delivery of personalized information to users as the amount of information readily available from sources like the WWW increases. When the number of information recipients is large and there is sufficient commonality in their interests, it is worthwhile to use multicast rather than unicast to deliver information. But, if the interests of recipients are not sufficiently common, there could be huge redundancy in traditional IP multicast. As the solution to this *Content-Based Multicast* (CBM) was proposed [5, 6] where extra content filtering is performed at the interior nodes of the multicast tree so as to reduce network bandwidth usage and delivery delay. This kind of filtering is performed either at the IP level or, more likely, at the software level e.g. in applications such as publish-subscribe [2] and event-notification systems [3].

Essentially, CBM reduces network bandwidth and recipient computation at the cost of increased computation in the network. CBM at the application level is increasingly important, as the quantity and diversity of information being disseminated in information systems and networks like the Internet increases,

and users suffering from information overload desire personalized information. A form of CBM is also useful at the middleware level [3, 1] and network signaling level [4]. Previous work applies CBM to address issues in diverse areas [3, 1, 4].

[1] addresses the problem of providing an efficient matching algorithm suitable for a content based subscription system. [2] addresses the problem of matching the information being multicast with that being desired by leaves. [5] proposes mobile filtering agents to perform filtering in CBM framework. They consider four main components of these systems: subscription processing, matching, filtering and efficiently moving the filtering agents within the multicast tree.

[6] evaluates the situations in which CBM is worthwhile. It assumes that the multicast tree has been set up using appropriate methods, and concentrates on efficiently placing the filters within that multicast tree. It also gives the mathematical modeling of optimization framework. The problem considered is that of placing the filters under two criteria :

- Minimize the total bandwidth utilization in the multicast tree, with the restriction that at most  $p$  filters are allowed to be placed in the tree. This is similar to  $p$ -median problem on trees. An optimum  $O(pn^2)$  dynamic programming algorithm was described.
- Minimize total delivery delay over the network, with no restriction on number of filters, assuming that the filters introduce their own delays  $F$  and the delay on the link of multicast tree is proportional to the amount of traffic on that particular link. That means although filters have their own delays, they could effectively reduce traffic and hence delays. This problem is similar to uncapacitated facility location on trees. An optimum  $O(n^2)$  dynamic programming algorithm was described.

In this paper, we consider the second formulation, (minimizing delay) and show that the complexity of the dynamic programming algorithm can be improved. We do not concern ourselves with the construction of the multicast tree, or with processing of the subscriptions. Also, we assume that minimum required amount of traffic at each node of multicast tree is known. We consider this to be a part of subscription processing. This could be done by taking of unions of subscription list bottom up on the multicast tree, or by probabilistic estimation. Given these we focus on the question of where to place filters to minimize the total delay. We also assume that the filters do as much possible filtering as they could and do not allow any extra information traffic on subsequent links.

In section 2 , we show the formulation of the problem. We discuss the preliminaries and assumptions. We go over the dynamic programming algorithm described in [6] and give the intuition which motivates faster algorithms. In sections 3 and 4, we describe two algorithms, the first being an improvement of the dynamic programming and the second being an improvement of the first. We also include the analysis of their running time . In section 5, we describe the piecewise linear functions data structure and the required operations on it along with the complexity bounds. In section 6, we consider further extensions and related problems.

## 2 Preliminaries

### 2.1 Notations

A filter placement on a rooted multicast tree  $M = (V, E)$  with vertex set  $V$  and edge set  $E \subset V \times V$  is a set  $S \subset V$  where filters are placed at all vertices in  $S$  and on no vertex in  $V - S$ . Let  $|V| = N$ , and so  $|E| = N - 1$ . We denote the root of  $M$  by  $r$ .  $Tree(v)$  denotes the subtree rooted at vertex  $v \in V$ . For example,  $Tree(r) = M$ . Let us denote the height of tree  $M$  by  $H$ .

For simplicity of writing, we will use some functional notations. We denote size of  $Tree(v)$ , that is the number of vertices in  $Tree(v)$ , by  $n(v)$ . Thus,  $|Tree(r)| = n(r) = N$ .  $c(v)$  denotes the number of children of vertex  $v \in V$ , while  $s(v)$  denotes the number of leaves in  $Tree(v)$ . For example,  $c(v) = 0$  and  $s(v) = 1$  if vertex  $v$  is a leaf in  $M$ .

$f(v)$  is the total size of information requested in  $Tree(v)$ . For a leaf  $v \in V$ ,  $f(v)$  denotes the size of the information requested from that user. In other words,  $f(v)$  is also the amount information that node  $v$  gets from its parent, if the parent has a filter. We assume that  $f(v)$  for each  $v$  is known.

### 2.2 Assumptions

We make the following assumptions in our model.

- The delay on a link is proportional to the length of the message transmitted across the link, ignoring propagation delay. Thus if  $m$  is the length of the message going across a link (or an edge), then the delay on that link is  $mL$  units, where the link delay per unit of data is  $L$ , a constant.
- The delay introduced by an active filter is a constant. We denote it by  $F$ . It is a (typically) a big constant.<sup>1</sup>
- Each internal vertex of  $M$  waits and collects all incoming information before forwarding it to its children. But this time is much smaller than the delay rate over the link.

### 2.3 Recurrence and Dynamic Programming

Our objective is to minimize the average delay from the instant the source multicasts the information to the instant that a leaf receives it. Since number of leaves is a constant for a given multicast tree  $M$ , we can think of minimizing the total delay, where total is made over all leaves in  $M$ .

Let  $A(v)$  stand for the lowest ancestor of  $v$  whose parent has a filter. For example,  $A(v) = v$  if parent of  $v$  has a filter.

Now consider a CBM with a required flow  $f$  known for each vertex in the tree. For a vertex  $v$ , let  $D(v, p)$  denote the minimum total delay in  $Tree(v)$ ,

---

<sup>1</sup> It is not necessary to assume that  $L$  and  $F$  are same constants for each link and each filter locations, they could be different constants for different links and location as in general formulation of  $p$ -median problem[7]. This will not change the algorithm.

assuming  $A(v) = p$ . Let  $v_1, v_2, \dots, v_{c(v)}$  be children of vertex  $v$ . Let  $C_v = s(v)F + L \sum_{i=1}^{c(v)} f(v_i)s(v_i)$  and  $E_v = Ls(v)$ .  $C_v$  and  $E_v$  are constants and can be computed for each vertex  $v$  in  $O(N)$  time by bottoms up calculation.

Then the minimum total delay can be expressed by the following recurrence relation

<pre> <b>if</b> <math>v</math> is a leaf <b>then</b>     <math>D(v, p) = 0</math> for all <math>p</math> <b>else</b>     <math>D(v, p) = \min\{</math>         <math>C_v + \sum_{i=1}^{c(v)} D(v_i, v_i)</math>, if <math>v</math> has a filter         <math>f(p).E_v + \sum_{i=1}^{c(v)} D(v_i, p)</math>, otherwise     <math>\}</math> <b>end if</b> </pre>
---

The optimal placement can be found using dynamic programming as noted in [6]. But a naive implementation of it would take time  $O(NH)$ .

We will “undiscretize” the above recurrence relation and write it as a function of a real number  $p$ , which is now the incoming information flow into  $v$ . To make it clear that this function is specific for a vertex  $v$ , we denote it as  $D_v$ . Now our recurrence relations takes a new look

<pre> <b>if</b> <math>v</math> is a leaf <b>then</b>     <math>D_v(p) = 0</math> for all <math>p</math> <b>else</b>     <math>D_v(p) = \min\{</math>         <math>C_v + \sum_{i=1}^{c(v)} D_{v_i}(f(v_i))</math>, if <math>v</math> has a filter         <math>p.E_v + \sum_{i=1}^{c(v)} D_{v_i}(p)</math>, otherwise     <math>\}</math> <b>end if</b> </pre>
---

Notice that we can still compute  $D(v, p)$  by plugging the discrete value  $f(p)$  in  $D_v$ . Intuitively,  $D_v$  is a function of real value  $p$  which is incoming flow to the  $Tree(v)$ . It is a piecewise linear non-decreasing function. Each break point in the function indicates a change in the arrangement of filters in the subtree  $Tree(v)$ . This change occurs in order to reduce the rate of increase of  $D_v$  (slope) for higher values of  $p$ . The slope of each segment is lesser than the previous, and the slope of final segment (infinite ray) is zero because this would correspond to filter at  $v$ . Once, a filter is placed at  $v$ , the value of variable  $p$  no longer matters. Therefore,  $D_v$  is a piecewise linear non-decreasing concave function. We will use  $|D_v|$  notation to denote the number of break-points (or number of linear pieces) in  $D_v$ .

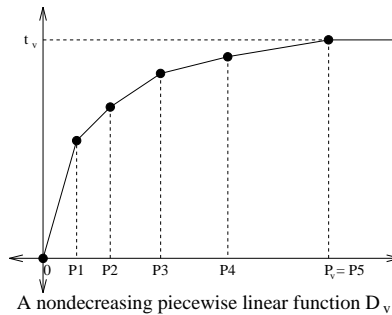
The big advantage of above formulation is that it allows us to store  $D_v$  as a height balanced binary search tree which in turn allows efficient probing and merging, so that we can implement above recurrence and find optimal filter placements in quicker time.

Before we proceed with actual description of algorithms and data-structures, we present two simple lemmas which prove useful properties of  $D_v$  as claimed above.

**Lemma 1.**  $D_v$  is a piecewise linear non-decreasing function and  $\exists p_v, t_v$  such that  $D_v(p) = t_v$  for all  $p \geq p_v$ .

**Proof:** By induction on height<sup>2</sup> of  $v$ . Claim is trivially true for a leaf  $v$ .  $D_v = 0$ . So  $p_v = 0$  and  $t_v = 0$ . Let's assume the claim is true for all  $c(v)$  children of  $v$ . Also let  $L_v(p) = p.E_v$ .  $L_v$  is a line passing through origin. Hence, it is a piecewise linear non-decreasing function.

Let  $W_v(p) = C_v + \sum_{i=1}^{c(v)} D_{v_i}(f(v_i))$ .  $W_v$  is a constant and hence a piecewise linear non-decreasing function. Let  $F_v(p) = L_v(p) + \sum_{i=1}^{c(v)} D_{v_i}(p)$ . Hence,  $F_v$  is a piecewise linear non-decreasing function.  $D_v(p) = \min\{W_v(p), F_v(p)\}$ . Therefore,  $D_v$  is a piecewise linear non-decreasing function because minimum preserves piecewise linear non-decreasing property.  $t_v = W_v(p)$  and  $p_v$  is the value of  $p$  where  $W_v$  and  $F_v$  intersect.  $\square$



**Lemma 2.**  $|D_v| \leq n(v)$ .

**Proof:** By induction on height of  $v$ . Claim is trivially true for  $v$  if its a leaf, that is its height is 0. If claim were true for each of the  $c(v)$  children of  $v$ , then each of  $D_{v_i}$  is a piecewise linear function made up of at most  $n(v_i)$  different linear pieces.  $D_v$  is a minimum of sum total of  $D_{v_i}$  and a constant. It can have at most one more extra piece added to it. So the number of linear pieces in it cannot be more than  $1 + \sum_{i=1}^{c(v)} n(v_i)$ . But that is precisely  $n(v)$ .  $\square$

It is apparent from the above proof that each break-point in  $D_v$  is introduced by some node in  $Tree(v)$  as a result of “min” operation.

### 3 Algorithm-1

We are now ready to present our first algorithm, *Algorithm-1*. Let  $I$  be the total amount of the incoming information at  $r$ . The function  $A(r)$  returns the piecewise linear function  $D_r$  at root  $r$ .  $D_v$  is stored as a balanced binary search tree whose size is equal to the number of break-points in  $D_v$ .

<sup>2</sup> Height of  $v = 1 + \max\{\text{Height of } u \mid u \text{ is a child of } v\}$ . Height of a leaf is 0.

### 3.1 Algorithm

<pre> <b>Algorithm-1</b> {   A(<i>r</i>);   M-DFS(<i>r</i>, <i>I</i>); } M-DFS (<i>v</i>, <i>p</i>) {   if <i>c</i>(<i>v</i>) == 0 then     return;   end if   if <i>p</i> &gt; <i>p</i><sub><i>v</i></sub> then     place filter at <i>v</i>;     for <i>i</i> = 1 to <i>c</i>(<i>v</i>) do       M-DFS(<i>v</i><sub><i>i</i></sub>, <i>f</i>(<i>v</i><sub><i>i</i></sub>));     end for   else     for <i>i</i> = 1 to <i>c</i>(<i>v</i>) do       M-DFS(<i>v</i><sub><i>i</i></sub>, <i>p</i>);     end for   end if   return; } </pre>	<pre> A(<i>v</i>) {   if <i>c</i>(<i>v</i>) == 0 then     <i>p</i><sub><i>v</i></sub> = +∞;     return create(0,0);   else     for <i>i</i> = 1 to <i>c</i>(<i>v</i>) do       <i>q</i><sub><i>i</i></sub> = A(<i>v</i><sub><i>i</i></sub>);     end for     <i>t</i><sub><i>v</i></sub> = <i>C</i><sub><i>v</i></sub>;     for <i>i</i> = 1 to <i>c</i>(<i>v</i>) do       <i>t</i><sub><i>v</i></sub> = <i>t</i><sub><i>v</i></sub> + probe(<i>q</i><sub><i>i</i></sub>, <i>f</i>(<i>v</i><sub><i>i</i></sub>));     end for     <i>z</i> = create(<i>E</i><sub><i>v</i></sub>, 0);     for <i>i</i> = 1 to <i>c</i>(<i>v</i>) do       <i>z</i> = add_merge(<i>z</i>, <i>q</i><sub><i>i</i></sub>);     end for     <i>p</i><sub><i>v</i></sub> = truncate(<i>z</i>, <i>t</i><sub><i>v</i></sub>);     return <i>z</i>;   end if } </pre>
--	--

### 3.2 Data Structure Operations

The data structure supports the following operations:

*create*(*a*, *b*): Returns a new function with equation  $y = ax + b$  in time  $O(1)$ .

*probe*(*q*, *t*): returns  $q(t)$  in  $O(\log |q|)$  time.

*add\_merge*(*q*<sub>1</sub>, *q*<sub>2</sub>): Returns a piecewise linear function which is the sum of *q*<sub>1</sub> and *q*<sub>2</sub>. Assuming without loss of generality  $|q_1| \geq |q_2| \geq 2$ , the running time is  $O(|q_2| \log(\frac{|q_1|+|q_2|}{|q_2|}))$ . *q*<sub>1</sub> and *q*<sub>2</sub> are destroyed during this operation and the new function has size  $|q_1| + |q_2|$ .

*truncate*(*q*, *t*): This assumes that some  $z$  s.t.  $q(z) = t$  exists. Modifies *q* to a function *q'* which is equal to  $q(x)$  for  $x \leq z$ , and  $t$  for  $x > z$ . This destroys *q*. It returns *z*. *q'* has at most one more breakpoint than *q*. All the breakpoints in *q* after  $z$  are deleted (except at  $+\infty$ ). The running time is  $O(\log |q|)$  for search plus time  $O(\log |q|)$  per each deletion.

### 3.3 Analysis of Algorithm-1

Algorithm-1 first recursively builds up the piecewise linear function  $D_r$ , bottom up, by calling *A*(*r*). It uses  $p_v$  values stored for each *v* in the tree and runs simple linear time Depth-First-Search algorithm to decide filter placement at each vertex of the tree.

We will now show that the total running time of algorithm *A*, and therefore Algorithm-1, is  $O(N \log N)$ . There are three main operations which constitute

the running time: *probe*, *truncate* and *add\_merge*. Over the entire algorithm, we do  $N$  probes each costing time  $\leq \log N$  because each probe is nothing but a search in a binary search tree. *truncate* involves  $N$  search operations and  $\leq N$  deletions (because each break-point is deleted only once and there is only one break-point each node in the multicast tree can introduce) each costing  $\leq \log N$  time. Therefore, *truncate* and *probe* cost  $O(N \log N)$  time over the entire algorithm. We still need to show that total time taken by all the merge operations is  $O(N \log N)$ . The following lemma proves this.

**Lemma 3.** *Total cost of merge in calculation of  $D_v$  is at most  $n(v) \log n(v)$ .*

**Proof :** We proceed by induction on height of  $v$ . The claim is trivially true for all leaf nodes since there are no merge operations to be done. At any internal node  $v$ , to obtain  $D_v$  we merge  $D_{v_1}, D_{v_2}, \dots, D_{v_{c(v)}}$  sequentially. Let  $s_i = \sum_{j=1}^i n(v_j)$ . Now, again by induction (new induction on the number of children of  $v$ ) assume that the time to obtain the merged function of first  $i$   $D_{v_j}$ 's is  $s_i \log s_i$ . The base case when  $i = 1$  is true by induction (previous induction). Then, assuming without loss of generality  $s_i \geq n(v_{i+1})$ , the total time to obtain merged function of first  $i+1$   $D_{v_j}$ 's is at most  $s_i \log s_i + n(v_{i+1}) \log n(v_{i+1}) + n(v_{i+1}) \log ((s_i + n(v_{i+1}))/n(v_{i+1}))$  which is at most  $s_{i+1} \log s_{i+1}$ . Therefore time taken to merge all the children at  $v$  and obtain  $D_v$  is at most  $n(v) \log n(v)$ .  $\square$

## 4 Algorithm-2

### 4.1 Motivation

We observe that lemma 2 suggests a bound of  $n(v)$  on the number of different linear pieces in  $D_v$ . On the other hand, we need to probe and evaluate  $D_v$  at at most  $H$  different values (that is the number of ancestors  $v$  can have !). This suggests that we can gain more if we “convert” our functions which grow “bigger” and have more than  $H$  breakpoints and reduce them to at most  $H$  breakpoints.

For example, consider the case of multicast tree  $M$  being a balanced binary tree. Let  $Y$  be the set of nodes at depth  $\log \log N$ . For each  $v \in Y$  the subtree size  $n(v)$  is roughly  $\log N$ .  $|Y|$  is roughly  $N/\log N$  and computing  $D_v$  at each such  $v$  takes  $\log N \log \log N$  time. This makes it  $N \log \log N$  over all  $v \in Y$ . Now, we convert  $D_v$  into array form as in dynamic programming and resume the previous dynamic programming algorithm in [6]. This dynamic programming calculations occur at roughly  $N/\log N$  nodes each taking  $\log N$  ( $H = \log N$ ) time. Hence we achieve an enhancement in total running time, taking it down to  $N \log \log N$  which is essentially  $N \log H$ . However, to achieve this in general case, we still have to stick to the binary search tree representation in the second phase. The advantage is that the size of the binary search tree never grows more than  $H$ .

### 4.2 Data Structure Operations

Before we proceed with this new algorithm, we consider “converted” functions, since some of the functions would be evaluated only for a small number of values. A “converted” function  $q_X(x)$  for  $q(x)$  with respect to (sorted) set  $X =$

$x_1, x_2, \dots, x_k$  is a piecewise linear function such that  $q(x_i) = q_X(x_i)$  for  $x_i \in X$ , and piecewise linear in between those values. We define the following operations:

*convert*( $q, X$ ): Returns the “converted” function  $q_X$  in  $O(k \log(|q|/k))$ . Assumes  $X$  is sorted.

*add\_dissolve*( $q_X, g$ ): Adds function  $g$  to the converted function  $q_X$ , and returns the resulting function dissolved w.r.t. set  $X$ . Running time :  $O(|g| \log |q_X|)$

*add\_collide*( $q_{X_1}, g_{X_2}$ ): Adds two converted functions  $q_{X_1}$  and  $g_{X_2}$ . Creates new converted function only on  $X_1 \cap X_2$ .

*truncate\_converted*( $f_X, t$ ): Almost the same as *truncate*. Does not cause any deletions. It uses some “mark” to keep track of invalid values in data structure. Running time:  $O(\log |f_X|)$ .

### 4.3 Description of Algorithm

Using this, we modify the implementation of Algorithm-1. We continue building  $D_v$ ’s bottom up as in algorithm  $A(v)$ . Suppose we are at some  $v$  whose all  $c(v)$  children, namely  $v_1, v_2, \dots, v_{c(v)}$ , have less than  $H$  breakpoints in their respective data-structures. We now start building  $D_v$  by merging  $D_{v_1}, D_{v_2}, \dots$  one by one. Suppose after merging  $D_{v_1}$  through  $D_{v_i}$  for first  $i$  children, we find that the number of breakpoints for function  $q$  constructed so far exceeds  $H$  (and it’s trivially less than  $2H$ ), then we call function *convert*( $f, X$ ). Here  $X$  is the sorted list of values  $f(p)$  of  $v$ ’s ancestors. Note that  $X$  is very easy to maintain due to recursive top-down calls in  $A(v)$  and monotonicity of  $f(p)$  values along a path. Then, for the remaining children of  $v$ , we use *add\_dissolve*( $q, D_{v_j}$ ), where  $j \in \{i+1, \dots, c(v)\}$ . Once, the function is “converted”, it always remains “converted”. For *add* operation involving one “converted” function  $q_1$  and one “unconverted” function  $q_2$  we use *add\_dissolve*( $q_1, q_2$ ) which runs in  $O(q_2 \log H)$  and for two “converted” functions  $q_1, q_2$  we use *add\_collide*( $q_1, q_2$ ) which runs in  $O(H)$  since the size of the bigger data structure is now restricted by  $H$ .

Thus, we don’t store more than required information about  $D_v$ , while maintaining enough of it to calculate the required parts of  $D_u$ , where  $u$  is  $v$ ’s parent.

### 4.4 Analysis of Algorithm-2

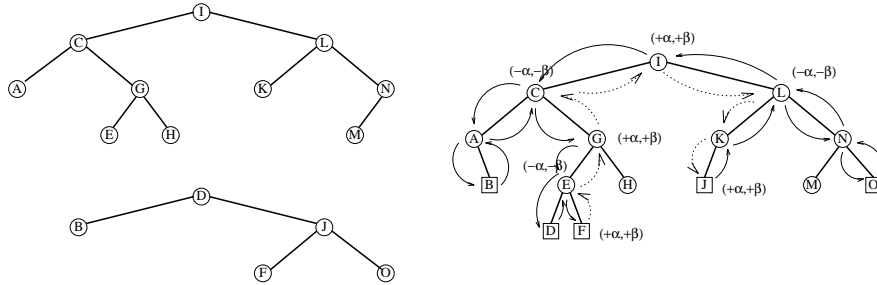
Let  $Y$  be the set of all nodes  $v \in V$  such that we needed to use the *convert* for finding  $D_v$ . Use of *convert* function implies  $n(v) = |Tree(v)| \geq H$  in the light of lemma 2.  $|Y| \leq \frac{N}{H}$  because for any two  $u, v \in Y$ ,  $Tree(u) \cap Tree(v) = \phi$ .

Let  $W$  be union of  $Y$  and all ancestors of nodes in  $Y$ . Subgraph of  $M$  on set  $W$  forms an upper subtree of  $M$ . Let  $U$  be the set of children of nodes in  $Y$ .



node to the root of the tree. A dummy breakpoint at  $x = +\infty$  will be included in the tree to encode the rightmost linear piece of the function.

Each node will also contain a mark for handling truncated parts of a function. A node is invalid (i.e. its  $a$  and  $b$  values are not correct) if itself or a node on the path to the root is marked. The linear function at the  $x$  value of an invalid node will be the same as the function of the first valid node that appears after it in the tree inorder. The node at  $x = +\infty$  will always be valid. Every time we visit a marked node during any of the operations, we unmark it, correct its  $a$  and  $b$  values and mark its two children. This ensures that the only invalid nodes are the ones the algorithm doesn't see. This marking scheme will be necessary to implement *truncate\_converted* which is "truncate" on "converted" functions, since we cannot delete the nodes in that case.



Sorted lists represented as height-balanced trees    Merging by sequential insertions (square nodes have been inserted)

The data structure will use the AVL tree merging algorithm of Brown and Tarjan [10] to implement *add\_merge*, *convert* and *add\_dissolve*. Given two AVL trees  $T_1$  with  $n$  elements and  $T_2$  with  $m$  elements,  $m \leq n$ , we will search/insert the elements of  $T_2$  into  $T_1$  in sorted order, but instead of performing each search from the root of the tree, we start from the last searched node, climb up to the first parent (LCA) having the next element to search in its subtree, and continue searching down the tree from there. Brown and Tarjan show that the total number of operations performed during this procedure is  $O(m \log((n+m)/m))$ . This method can also be used to search or visit  $m$  sorted values in  $T_1$  within  $O(m \log(n/m))$  time.

In order to add two functions, while merging the corresponding AVL trees using Brown and Tarjan's method, and we will need to update the  $a$  and  $b$  values of the nodes. First, when we insert a new node in the tree, we find its "inherited"  $A$  and  $B$  values and adjust its  $a$  and  $b$  values accordingly. Then we consider the effect of adding the linear piece  $y = \alpha x + \beta$  to its right in the previous data structure where it came from. This can be done along the same walk in the tree. While walking in the tree from an element  $u$  to the next element to be inserted  $v$ , we will need to add the piecewise linear function joining them, say  $\alpha x + \beta$  to all the nodes between  $u$  and  $v$ . To do that, add  $\alpha$  and  $\beta$  to the  $a$  and  $b$  values of the least common ancestor (LCA) of  $u$  and  $v$ . Now, the function values for all the nodes between  $u$  and  $v$  have been increased correctly, but some nodes outside of that range might have been increased as well. To correct that,

we walk down from the LCA to  $u$ . This is a series of right child and left child choices, the first being left. In this series, whenever we choose right child after some (non-empty) sequence of left child choices, we subtract tuple  $(\alpha, \beta)$  at that node. Similarly, whenever we choose left child after (non-empty) sequence of right child choices, we add tuple  $(\alpha, \beta)$  to the node where choice is made. Also, similarly (vice-versa)  $a$  and  $b$  values can be adjusted along the path LCA to  $v$ . Thus, updates are only required along the Brown and Tarjan's search path. To complete the argument, it can be verified that the validity of the  $a$  and  $b$  values of the nodes can be preserved during a rotations and double rotations in the tree for the AVL insertion. The figure illustrates the insert path along with updates due to linear segment  $\alpha x + \beta$  between inserted points  $F$  and  $J$ .

We now outline the workings of the different operations:

*create(a, b)*: Create a new AVL tree with 1 node at  $x = +\infty$ , and set its  $a$  and  $b$  values.

*add\_merge(f<sub>1</sub>, f<sub>2</sub>)*: Use the method of Brown and Tarjan as described above.

*truncate(f, t)*: Find  $z$  such that  $f(z) = t$  by performing a search in the tree. As we go down the tree, we maintain  $A$  and  $B$ , the sum of the  $a$  and  $b$  values of all the ancestors of the current node. This way, we can compute the value of  $f(x)$  for the  $x$  value of each of the nodes visited. Since the function  $f$  is non-decreasing, the tree is also a binary search tree for the  $f(x)$  values. Once the search is done, we find the linear segment for which  $Az + B = t$ , and thus find  $z$ . We then insert a new breakpoint in the tree at  $x = z$ , and delete all the break-points in  $f$  which come after  $z$  (except the one at  $+\infty$ ) one-by-one, using usual AVL-tree deletion. Add the line segment  $(0, t)$  between  $z$  and  $+\infty$ .

*probe(f, t)*: Search in the tree the successor for  $t$  in the  $x$  values. Compute the  $A$  and  $B$  sums on the path, returns  $f(t) = At + B$ .

*convert(f, X)*: Use the method of Brown and Tarjan to find the successors of all  $x_i \in X$  in  $O(k \log(n/k))$ . Evaluate  $f(x_i)$  at each of those values, and construct a new AVL tree for a piecewise linear function with  $x_i$  values as breakpoints, and joining each adjacent breakpoints with an ad-hoc linear function.

*add\_dissolve(f<sub>X</sub>, g)*: Just like in *add\_merge*, but we do not insert the break-points, we just update the  $a$  and  $b$  values of the existing breakpoints.

*add\_collide(f<sub>X<sub>1</sub></sub>, g<sub>X<sub>2</sub></sub>)*: Find the values of  $f$  and  $g$  on  $X_1 \cap X_2$  and construct a new AVL tree as in *convert*.

*truncate\_converted(f<sub>X</sub>, t)*: As in *truncate*, we find  $z$ . But in this case we do not insert the new break-point. Also we do not delete the break-points in  $f_X$  after  $z$ . We invalidate the  $(a, b)$  values of the remaining points by marking the right child of the nodes traversed from the left and also we adjust  $(a, b)$  value at these nodes so that the linear function reflects the line  $y = t$ , while walking up from the position of  $z$  to the root. Once at the root, we walk down to  $+\infty$ , validating the  $a$  and  $b$  values on that path. We then set the  $a$  and  $b$  values at  $+\infty$  such that  $A = 0$  and  $B = t$ . It returns  $z$ .

## 6 Related Problems and Future Work

Future work in this area is to design an algorithm based on similar methods for the first model of [6]. Here, the dynamic programming algorithm runs in  $O(pn^2)$  time, since the objective function at each node involves two parameters. This model, where only  $p$  filters are allowed, is a variant of  $p$ -median on tree where all the links are directed away from the root. The problem is called  $p$ -inmedians in [9]. When  $p \geq n$  it reduces to our problem. The dynamic programming optimal algorithm known for the uncapacitated facility location (which is  $p$ -median with  $p \geq n$ ) over trees takes  $O(n^2)$  time. Interesting future work is to find faster algorithms for uncapacitated facility location [9] and  $p$ -median [7]. That could also give us faster algorithms for  $p$ -forest [8] and tree partitioning problems [9].

**Acknowledgments:** We would like to thank John Iacono and Michael Fredman for useful suggestions regarding choice of data structures and Farooq Anjum and Ravi Jain for providing useful background on the problem.

## References

- [1] Aguilera et al, "Matching Events in a Content-based Subscription System", <http://www.research.ibm.com/gryphon>
- [2] Banavar et al, "An efficient multicast protocol for content-based publish-subscribe systems", *Technical report*, IBM 1998.
- [3] Carzaniga et al, "Design of Scalable Event Notification Service: Interface and Architecture", *Tech Report CU-CS-863-98, University of Colorado, Dept. of Computer Science*, 1998.
- [4] Kasera et al, "Scalable Fair Reliable Multicast Using Active Services", *IEEE Network Magazine*, Jan/Feb 2000.
- [5] F. Anjum and R. Jain, "Generalized Multicast Using Mobile Filtering Agents", *Internal Report, Telcordia Tech, Morristown*, Mar 00.
- [6] F. Anjum, R. Jain, S. Rajagopalan and R. Shah, "Mobile Filters for Efficient Dissemination of Personalized Information Using Content-Based Multicast", *submitted*, 2001.
- [7] A. Tamir, "An  $O(pn^2)$  algorithm for the  $p$ -median and related problems on tree graphs", *Operations Research Letters*, 19:59-94, 1996.
- [8] A. Tamir and T. Lowe, "The generalized  $p$ -forest problem on a tree network", *Networks* 22, 217-230, 1992.
- [9] 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.
- [10] M. Brown and R. Tarjan, "A Fast Merging Algorithm", *Journal of ACM*, 26(2), pp 211-225, Apr 79.
- [11] 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.
- [12] C. Crane, "Linear lists and priority queues as balanced binary trees", *PhD Thesis, Stanford University*, 1972.