UNDISCRETIZED DYNAMIC PROGRAMMING AND ORDINAL EMBEDDINGS

BY RAHUL SHAH

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Martin Farach-Colton

and approved by

____

____

____

____

New Brunswick, New Jersey

May, 2002

ABSTRACT OF THE DISSERTATION

Undiscretized Dynamic Programming and Ordinal Embeddings

by Rahul Shah

Dissertation Director: Martin Farach-Colton

Many optimization problems which are known to be NP-hard on graphs are polynomially solvable on trees using dynamic programming. Dynamic programming typically involves recursive functions stored as tables. Each entry of the table corresponds to the optimal subproblem solution. In many applications the complexity of brute-force dynamic programming can be improved especially when the functions involved are sparse, convex or concave. Algorithms for speeding up dynamic programming on path-like one dimensional structures are known. Here, we "undiscretize" the functions i.e. represent them as functions of continuous variable instead of storing the functions as the tables of discrete values. We use efficient data structures to store such functions and show how to quickly carry out operations involving these functions. We improve the complexity bounds for many tree dynamic programming problems, typically, from $O(n^2)$ to $O(n \log n)$. These include problems like facility location, covering, economic lot sizing and multicast filtering.

Given a set of pairwise distances on a set of n points, constructing an edge-weighted tree whose leaves are these n points such that the tree distances would mimic the original distances under some criteria is a fundamental problem in computational biology. This problem can also be seen as hierarchical clustering or as an embedding into additive (tree) metric spaces. In ordinal embeddings, the distance preservation criterion is to preserve the total or partial order of the pairwise distances. We show that the problem of finding a weighted tree, if it exists, which would preserve the total order on pairwise distances is NP-hard. A partial order on pairwise distances between points which orders all distances that share an end point is equivalent to an order where the distances in each triangle are ordered. This order, called *triangle order*, has been studied in biological settings. We also show the NP-hardness of the problem of finding the weighted tree which would preserve a triangle order. This answers a long standing open problem considered extensively by computational biologists.

Acknowledgements

This work reflects a collection of my attempts to rise to a point where I can see beyond the fortifications of current knowledge and make new frontiers all my own. Whether this is yet high enough, I do not know. But I stand on many shoulders to achieve this height, and to each I owe an immeasurable debt of gratitude.

Above all (or beneath all!) stand my family. I thank them for their love and consistent motivation, and the unbending faith they have in me. Particularly, my beloved wife Avni. This thesis is as much hers as it is mine. She has supported me in every way possible. Listening to and creating my presentations, writing my resumes, proofreading my papers, writing prototypes for my documents and collaborating on some of my projects are only a few of the unending forms of life support she has provided.

Special thanks to my childhood friends Jitu and Sunil. They made sure that my transition to life in US was as smooth as possible, and taught me almost everything I know about it. They have been an unfailing resource for everything from telephone deals to tax returns. My life here would not have been as complete and enriching without them.

Thanks also to Sachin for being a benevolent senior and friend. He taught me the ABCs of the computer science department at Rutgers - from what courses to take to how to go about the qualifiers. I thank him for valuable and memorable discussions on research and philosophy, and for writing a wonderful draft of our award winning paper. My special thanks to Stefan Langerman whose contributions to, and faith in, the undiscretization paradigm were really critical to the completion of this thesis. This thesis would surely not have been anywhere near its completion (in fact, may not have existed) without his sharp and brilliant ideas. His inspirations motivated several critical proofs contained herein.

Thanks to Stefan (again) and John Iacono, for their initiative with the Rutgers Algorithms Group. Thanks also to Navin Goyal, for useful discussions and for voluntarily taking on part of my teaching responsibilities.

Thanks to Val (Mrs. Valentine Rolfe) for being a caring well-wisher. She made the department seem like a home away from home.

My deepest thanks to Dr. Ondria Wasem, Dr. Farooq Anjum and Dr. Ravi Jain from Telcordia Technologies for supervising various summer internships and helping to shape parts of this work.

Thanks are due to Prof. Michael Fredman, Prof. Leonid Khachian, Prof. Bahman Kalntari, Prof. Joszef Beck and Prof. Mike Saks for the background-building courses they taught. Thanks to Prof. Fredman and Prof. Saul Levy for being on my qualifiers committee. Thanks to Prof. Fredman for interesting discussion regarding the choice of data structures for "undiscretization".

Thanks are due to Prof. Arie Tamir, Prof. Alexander Ageev and Dr. Stephen Alstrup for insightful and instructive comments and discussions regarding my work.

I also thank Prof. Muthukrishnan, Prof. Vasek Chvatal and Prof. Sampath Kannan for serving on my committee, extensively reviewing my work for quality and completeness, and keeping me in agony of suspense for no more than 30 seconds before declaring my defense successful. Any further delay would have put me in danger of coronary failure.

Last but not the least, thanks to my advisor Prof. Martin Farach-Colton for introducing me to deeply interesting problems, for helping me progress in my research every way possible and for giving me enough independence that I could take my own research path. Thanks also for his interesting lessons on life, the universe and everything.

Thank you all, for helping me climb this far, and for adding to my strength and confidence as I take on the road ahead.

Dedication

To my pappa

whom I owe my scientific attitude and aptitude

To my mummy

for her unconditional love and sacrifices

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

We consider two sets of problems in this thesis. One is Ordinal Clustering where we consider the problem of constructing a tree representing some order on pairwise distances. The other is a set of facility location related problems on trees where we propose faster algorithms. We shall organize the material presented in three parts:

- **Ordinal Embeddings into Trees:** Given a total or triangle order on the set of pairwise distances on n points, we wish to construct a weighted tree whose leaves represent these n points and the pairwise pathlengths follow the given order. We show that this problem is NP-hard.

- **Efficient Filtering in Content-Based Multicast:** Filters on the intermediate routers of the content based multicast tree can help reduce congestion and delivery delays. Optimal dynamic programming algorithms are developed to choose the best nodes in the tree where the filters could be placed. We develop a technique called "undiscretized dynamic programming" This technique speeds up the dynamic programming algorithm for minimizing delays on multicast tree. We improve an $O(nh)$ algorithm to $O(n \log h)$ where n is the number of nodes in the multicast tree and h is the height.

- **Facility Location on Trees and Related Problems:** We apply the "undiscretization" technique to dynamic programming algorithms for facility location and related problems on trees. We typically improve $O(n^2)$ algorithms to $O(n \log n)$.

We provide the detailed introduction to above topics in the remainder of the chapter.

1.1 Ordinal Embeddings into Trees

Clustering data based on pairwise distances is a fundamental problem. Weighted trees can be used to represent hierarchical clusters. Thus, constructing weighted trees that "fit" a distance matrix is a type of clustering. This metric based problem has been extensively studied, for example, in evolutionary biology and computational linguistics [15, 21, 8, 12, 36]. The general problem is to find an edge weighted tree which approximates the distance matrix under some criteria. These kind of trees are also known as *phylogenetic trees* or simply *phylogenies*. A particular instance of this problem has been considered in the algorithmic computational biology community : Given a (total or partial) order on the pairwise distances between points, give an edge weighted tree on those points so that the pairwise pathlengths between these points in the tree satisfies this order. This can also be seen as the ordinal version of the Hierarchical Multidimensional Scaling (HMDS) problem [31]. If the tree is unweighted then pairwise pathlengths would be the number of edges on the path. The problem of obtaining such an unweighted tree was considered by Kannan and Warnow [23] and Kearney, Hayward and Meijer [26].

In general, we are given an $n \times n$ distance matrix M and asked to find tree T on leaves $1, 2, .., n$ such that path distance $d_T(i, j)$ in the tree closely approximates the matrix M. When $d_T(i, j) = M(i, j)$ the matrix is said to be *additive* and efficient algorithms exist for constructing tree from additive distance data in $O(n^2)$ time [12, 21, 36]. But when the matrix is not additive then various optimization criteria were proposed and many of them are shown to be NP-hard [13, 26, 15].

However, for many applications, the actual numeric data is quite unreliable (see [15]). The distance data obtained through experiments could have kinds of errors like *homoplasy* and *superimposed changes* [26]. Also there are experiments which give only relative information about pairwise distances [23]. Hence, Kannan and Warnow [23] took an approach which assumes confidence only in relative information and the input is in the form of partial or total order on the pairwise distances. [26] and [23] consider the problem of constructing an edge-weighted tree which preserves the order among the distances. The advantage of using orders is that they are less vulnerable to errors due to superimposed changes. Superimposed changes do not affect the relative ordering of pairwise distances.

The first explicit study of ordinal methods for inferring phylogenetic trees was undertaken by Kannan and Warnow [23]. They worked with orders generated by experiments on pairwise distances of triplets of points. They consider a particular partial order on pairwise distances in which every triangle is totally ordered. This order is called *triangle order*. They gave an $O(n^3)$ algorithm for constructing an unweighted tree, if it exists, from a triangle order on pairwise distances. In the unweighted case, each edge of the tree is assumed to have unit weight. And also it is assumed that there are no vertices with degree two. Kearney, Hayward and Meijer [26] extended this work. They gave an $O(n^2 \log^2 n)$ algorithm for constructing an unweighted tree, if it exists, from a total order. The weighted case was posed as the major open problem in both the papers. The unweighted case is a special case of weighted case. Although we have polynomial time algorithms to find the unweighted tree, for many orders the unweighted tree representing that order may not exist at all even if the weighted tree exists. A polynomial time algorithm was conjectured in these papers for the weighted case. The algorithm was based on so called *mid-path tree conjecture* which stated that if the weighted tree representing triangle or a total order existed then it could be represented with some weight function on a special tree called *midpath tree* corresponding to that order. Contrary to the conjecture, we show in this paper that the problem of constructing weighted trees from total as well as triangle orders is NP-hard. We call these problems *Total Ordinal Clustering* and *Triangle Ordinal Clustering*, respectively. The reduction uses the generalization of a counter-example to the midpath tree conjecture given by Shah and Farach-Colton [31]. This NP-hardness is an interesting result since the unweighted case here is polynomial time solvable. Also, these are the first NP-hardness results in this area where all the known NP-hard problems come only from incomplete distance matrices or incompletely specified orders.

1.2 Efficient Filtering in Content-Based Multicast

In this part, we consider the problem of optimally locating filters on a multicast tree. Since this is a tree problem, $O(n^2)$ dynamic programming solution was known. We introduce the technique called undiscretized dynamic programming and pose this problem as a simple and straight forward application of this technique. We improve the running time of the dynamic programming algorithm. We first provide some background on the problem and the technology involved.

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 a huge redundancy in traditional IP multicast. As the solution to this *Content-Based Multicast* (CBM) was proposed [5, 30] 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 [6] and event-notification systems [9].

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 [9, 3] and network signaling level [24]. Previous work applies CBM to address issues in diverse areas [9, 3, 24, 37].

[3] addresses the problem of providing an efficient matching algorithm suitable for a content based subscription system. [6] 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.

The benefits of CBM depend critically upon how well filters are placed at the interior nodes of the multicast tree. [30] 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. They assume that subscriptions are appropriately processed and minimum required information for each subtree of each node is known. 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.

Here, we consider the second formulation, (minimizing delay) and show that the complexity of the dynamic programming algorithm can be improved. The traditional dynamic programming computes the optimal objective function for the subproblem at each subtree of the rooted multicast tree recursively in the bottom-up fashion. At each node, the incoming traffic at that node is taken as a parameter. There could be as many as n different values of incoming traffic. Hence, at each node the dynamic programming table could have n entries. This makes total table size of n^2.

We "undiscretize" the traditional dynamic programming function so that we don't have to maintain a separate entry for each possible incoming traffic value at a particular node. In that sense, we represent the dynamic programming function as a continuous function of incoming traffic value. We show that this continuous function is piecewise linear. We use AVL-tree based data-structure to represent this continuous function. We bound the size of this data-structure and show how to create subsequent generations of the data structure as the dynamic programming progresses towards the top. We use Brown and Tarjan's fast merging algorithm to add two of such functions and show how to prune and probe this data structure to create the subsequent functions.

Thus, using a succinct representation of dynamic programming functions and a quicker way of updating this representation, we achieve the complexity bound of $O(n \log n)$ for this algorithm.

We further show an improvement of this algorithm. This is a two-phased algorithm which works in between undiscretized and traditional dynamic programming. We improve the running time to $O(n \log h)$ where h is the height of the multicast tree.

1.3 Facility Location on Tree and Related Problems

In this part, we further extend, the "undiscretization" technique to dynamic programming functions involving more complex operations. We also exploit concavity and convexity of the functions involved. The problems considered involve the uncapacitated facility location on trees, covering problems on trees and Economic lot sizing problems. Again, we improve the complexity of traditional dynamic programming algorithms.

The UFL problem [27, 32, 10] 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 [32, 10], an integer linear programming formulation for this problem is given. This problem, in general, was shown to be NP-hard [10, 18] by reduction to vertex cover. But on trees, various polynomial algorithms are known.

The first algorithms for tree location problem were independently given by Gimadi [19] and Kolen [27]. Kolen reduced UFL problem to covering problem and showed when the cost matrix has a standard greedy form, UFL can be solved in $O(n^3)$. Cost matrix for trees does satisfy this standard greedy criteria. Gimadi gave $O(n^2)$ algorithm for UFL when the cost matrix is connected with respected to a tree. Kolen's algorithm was later improved and dynamic programming algorithms were given by [10, 34]. Shaw [32] gave a limited column generation based algorithm for UFL on trees. All these algorithms were $O(n^2)$.

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 [34, 27]. 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 [32, 10]. 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 [34] where $k \geq n$. Tamir's [34] 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 [34] with "undiscretized functions. In the filtering problem (section 1.2), the undiscretized representation of monotonic function as a piecewise linear function and is used and the technique involves how to quickly add two such functions and probe them. In case of UFL, 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 in the previous section do not maintain the consistency of these functions in this 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 [34] 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 $O(n \log n)$.

Several generalizations of the UFL problem on trees have been proposed. The tree partitioning problem by Cornuejols, Nemhauser and Wosley [10] was shown to be a generalization of the Economic Lot Sizing (ELS) problem as well as of UFL. Shaw [32] 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.

Chapter 2

NP-hardness of Embedding Orders into Trees

In section 2.1 we give some concepts and basic lemmas established in the field. In section 2.2, we show a generalization of the counter example for Midpath Tree Conjecture In section 2.3, we show the NP-hardness of the Triangle Ordinal Clustering problem. In section 2.4, we extend the NP-hardness result of section 2.3 to Total Ordinal Clustering.

2.1 Preliminaries

Here, we discuss some preliminary concepts and results known in this area. We shall also introduce some useful concepts that we will use in proving our main results.

2.1.1 Realizability and LP's

Define d_{T_w} to be the distance metric of a tree T under a non-negative weight function w, where $d_{T_w}(s,t)$ is sum of the weights of the edges on the unique path $P_T(s,t)$ from leaf s to leaf t in T. A partial order P on pairwise distances is said to be *realizable* as tree T if, for some weight function w on the edges of T, we get $d_{T_w}(a,b) \leq d_{T_w}(c,d)$ whenever $d(a,b) \leq_P d(c,d)$

Given a tree T, we can determine in polynomial time via linear programming whether or not the partial (or total) order P can be realized as T by checking the feasibility of the order constraints

$$d_{T_x}(c,d) - d_{T_x}(a,b) \quad \geq \delta \text{ if } d(a,b) <_P d(c,d) \tag{2.1}$$

$$d_{T_x}(c,d) - d_{T_x}(a,b) \quad = 0 \text{ if } d(a,b) =_P d(c,d) \tag{2.2}$$

$$x \geq 0 \tag{2.3}$$

where $\delta > 0$ is a constant.

2.1.2 Contractions and Expansions

A *contraction* of a tree T at the edge pq is the tree that results from removing edge pq from T and identifying vertices p and q. An *expansion* of T at a vertex v is the inverse operation of contraction,though many expansions may be possible at any node. A tree $T^{'}$ is called a contraction of T if $T^{'}$ is obtained by the contraction of T at some edge, or if $T^{'}$ is a contraction of some contraction of T. $T^{'}$ is called an expansion of T if T is a contraction of $T^{'}$.

2.1.3 Midpath Trees, Triangle order and MPT conjecture

Many assertions in this subsection are analogically true for triangle as well as total orders. In fact they would be true for any partial order at least as specified as triangle order. We call such orders as *supertriangular order*. Note that the triangle order and the total order are two extreme subcases of supertriangular order. Let T be a tree which realizes a supertriangular order Δ. For any two leaves x and y, the midpoint of the weighted path $P_T(x, y)$ is that point on $P_T(x, y)$ which is equidistant from x and y. Let u denote the edge or the vertex on which this midpoint is located. Consider $T' = T - u$. T' consists of at least two connected components. Now, since T realizes the order, the following trichotomy property holds:

- For all leaves v in the connected component of T' containing x, $d(v, x) <_\Delta d(v, y)$.
- For all leaves v in the connected component of T' containing y, $d(v, y) <_\Delta d(v, x)$.
- For all leaves v not in the connected components of T' containing x or y, $d(v, x) =_\Delta d(v, y)$.

The edge or vertex on $P_T(x, y)$ that satisfies the trichotomy property is called the *midpath* of $P_T(x, y)$ and is denoted by $M(x, y)$.

Given a tree T which realizes a triangle (or total) order Δ on pairwise distances between points in set S, along with a weight function w,consider a function $m : S \times S \rightarrow$

$E(T) \bigcup V(T)$ which maps each pair of leaves a, b to the midpath $M(a, b)$ on which the midpoint of the weighted path $P_T(a, b)$ falls. Now, contract all the (non-leaf)edges in T which do not have any midpaths falling on them to obtain an unweighted tree. This is the tree on which midpaths for all pairs of leaves exists and these midpaths satisfy the trichotomy property. We call this tree a *midpath tree* T_Δ. Note that a midpath tree is just an unweighted tree with the midpath function.

The midpath $M(a, b)$ gives a bipartition (or tripartition) of points in S based on whether they are closer to a or to b (or equidistant). Given the midpath tree with the Midpath function, we can construct the triangle order represented by it. This means the midpath tree T_Δ and midpath function M (weight function not required) can be used to represent a unique triangle (*not* total) order. The midpath tree is a minimal tree on which midpath function satisfying such a trichotomy property can be defined. As noted in the beginning of this subsection, if a supertriangular order can be realized by some tree T, then the midpath tree for this order exists. That means the existence of the midpath tree is a prerequisite for the existence of a tree T which realizes the supertriangular order order. Hence, while we don't know the algorithm to construct a tree T realizing the supertriangular order, the algorithm to construct the midpath tree is considered to be the first step in developing any such algorithm. and any such T is an expansion of T_Δ [26]. Intuitively, the existence of midpath tree indicates that the supertriangular order has a tree-like nature.

Following two lemmas (for proofs see [26]) show the importance of midpath trees.

Lemma 2.1 *If it exists, the midpath tree of a supertriangular order is unique.*

Lemma 2.2 *If T is a tree that realizes a supertriangular order Δ, then T is an expansion of midpath tree T_Δ.*

The following is an $O(n^3)$ algorithm that constructs the midpath tree: begin with a star topology and repeatedly expand until all midpaths exists. An optimal $O(n^2)$ algorithm for finding the midpath tree was given by [25]. Hence, constructing the midpath tree is considered to be the first step in finding the tree T that realizes the supertriangular order. Kearney, Hayward and Meijer [26] start with the midpath tree and then give an algorithm to expand it to obtain an unweighted tree that represents the total order, if such a tree exists.

To summarize, Given a weighted tree, the distance matrix can be can be constructed from it using pair wise pathlengths. Given a distance matrix, a total order can be induced by it and then the triangle order can be induced as a subset of this total order. Given a weighted tree, the midpath tree can be constructed from it and again triangle order can be constructed from the midpath tree. If a midpath tree exists, it can be constructed from the triangle order and hence from total order and hence form distance matrix. Lemma 3 gives and example of the midpath tree induced by a total order.

To construct the weighted tree representing the supertriangular order, it was not known whether we need to expand the midpath tree or if the midpath tree itself is the tree T that realizes the order. In case an expansion is necessary, no general methods for expanding the midpath tree are not known. This led to the conjecture [22]:

"If a supertriangular order Δ is realizable as some tree, then it is realizable as the midpath tree T_Δ"

If this conjecture were to be true, then no expansion methods would be necessary. We could just construct a midpath tree and run the linear program given by (2.1), (2.2), (2.3) and it would give us the required weighted tree. If the linear program were infeasible, then we would know that no tree realizes the order. But this conjecture is false for both total and triangle orders. In the following subsections, we shall show the counter-examples.

2.1.4 LP on Midpath tree and Dual

If the midpath tree conjecture were true, it would mean that if a triangle order $\leq_\Delta$ is realizable, it can be realized by assigning weights to the edges of midpath tree which satisfy the following linear constraints (2.5),(2.6) and (2.7).

Let $P_T^+(a,b)$ = set of edges on the path from a to mid-point $M(a,b)$, including the edge $M(a,b)$ if $M(a,b)$ is an edge. Let $P_T^-(a,b)$ be the set of edges on the path from

$M(a,b)$ to b not including $M(a,b)$. For each e edge e, let x_e be the weight variable on edge e. Then $\forall a, b \in S$,

$$\begin{aligned} &\mathit{minimize}\ 0 && (2.4)\\ \sum_{e\in P_T^+(a,b)} x_e - \sum_{e\in P_T^-(a,b)} x_e &\geq \delta \quad \mathit{if}\ M(a,b) \in E(T) && (2.5)\\ \sum_{e\in P_T^+(a,b)} x_e - \sum_{e\in P_T^-(a,b)} x_e &= 0 \quad \mathit{if}\ M(a,b) \in V(T) && (2.6)\\ x_e &\geq 0 \quad \forall e && (2.7) \end{aligned}$$

Now let's see what the dual looks like. Let y_{ab} be the weight for the midpath constraint (a,b) in above LP. Then the dual is,

$$\begin{aligned} &\mathit{maximize}\ \delta \sum_{M(a,b)\in E(T)} y_{ab} && (2.8)\\ \sum_{(a,b) ste\in P_T^+(a,b)} y_{ab} - \sum_{(a,b) ste\in P_T^-(a,b)} y_{ab} &\leq 0 \quad \forall e && (2.9)\\ y_{ab} &\geq 0 && (2.10) \end{aligned}$$

Then, to prove the infeasibility of this LP, we have to prove that its dual has unbounded maximum, because dual is always feasible with solution $y_{ab} = 0\ \forall a, b$. It can be seen from the dual that if the maximum objective value is non-zero then it has unbounded maximum. So, any multiset of constraints from 2.5, 2.6 such that at least one of them is from 2.5, which adds up to have non-positive weight on each edge, forms the witness of infeasibility of the LP. To generate, the counter-example to midpath tree conjecture, we need to work with the weighted tree, in which there is such a multiset of midpath constraints which add up positively only on some non-midpath edge. (Note that if it is a weighted tree then, any such multiset of constraints has to add up positively on at least one edge.)

2.1.5 Lemmas and Counter-Examples

Lemma 2.3 *There are total orders which are not realizable as any tree, despite the existence of midpath tree. [29]*

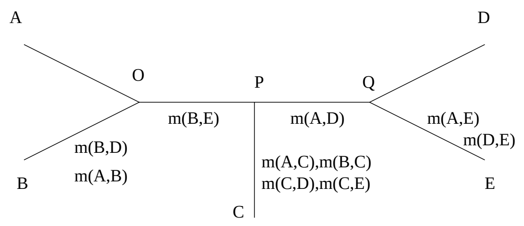


Figure 2.1: Not all total orders are realizable

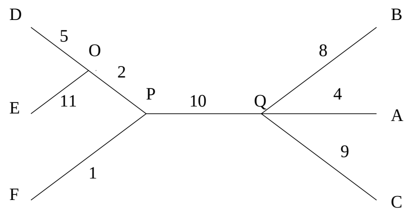


Figure 2.2: Total orders not realizable as midpath tree

Proof: Consider the total order $AD < DE < AB < BD < AE < BE < AC < CD < CE < BC$. Figure 2.1 shows the midpath tree for this total order. The mid-path tree is binary so it can not be expanded further. Following analysis shows that this tree can not be weighted to realize this total order.

$$BD < AE => BO + DQ < AO + EQ$$

$$AC < CD => AO + OP < PQ + DQ$$

$$CE < BC => PQ + EQ < BO + OP$$

summing up we get, $0 < 0$, which is contradiction. □

Lemma 2.4 *There are total orders which are realizable as some tree T but are not realizable as midpath tree. [29]*

Proof: Figure 2.2 gives the example. The total order is the one generated by pairwise distances in the weighted tree shown in the figure. Note that no mid-point falls on the edge OP i.e. $\forall x, y\ M(x, y) \neq OP$. So the midpath tree is obtained by contraction at edge OP. Let's identify O and P both by R to obtain the midpath tree. Then,

$$AC < EF \Rightarrow AQ + CQ < ER + FR$$

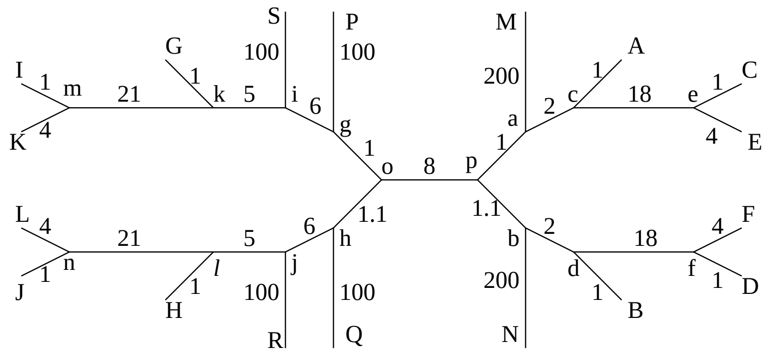


Figure 2.3: Δ not Realizable as Midpath Tree

$$DE < BC \Rightarrow DR + ER < BQ + CQ$$

$$CF < AD \Rightarrow CQ + FR < AQ + DQ$$

$$BE < CE \Rightarrow BQ < CQ$$

summing up , we get $0 < 0$. This is contradiction. Hence, this total order cannot be realized as midpath tree. □

Lemma 2.5 *There are triangle orders which are realizable as some tree but not the midpath tree. [31]*

Proof: Figure 2.3 gives the counter-example for the midpath tree conjecture for triangle order. Although a much smaller counter example for total orders exists, note that the same forms the counter example for midpath tree conjecture for total orders.

It consists of a weighted tree T which is a realization of the obvious triangle order generated by the pairwise path distances in T. It consists of 18 leaves, and no midpoint falls on edge *op* i.e. $\forall x, y\ M(x, y) \neq op$. Hence edge *op* is contracted in the midpath tree. All other edges have some midpoints on them. The question is : *is it possible to have some other weight assignment of T which will maintain the same triangle order and will have weight of edge $op = 0$?* The linear program generated is infeasible, thus establishing that the answer is no. The table 2.1 shows the witness to the infeasibility. Adding up the last column of the table we get,

pair	midpoint	inequality	that implies
AD	df	$AB <_\Delta BD$	$oA + 2pd < oD$
DE	ac	$DM <_\Delta ME$	$oD + 2pa < oE$
EG	ac	$EA <_\Delta AG$	$oE - 2oc < oG$
GJ	jl	$GR <_\Delta RJ$	$oG + 2oj < oJ$
JK	gi	$JP <_\Delta PK$	$oJ + 2og < oK$
KD	gi	$KS <_\Delta SD$	$oK - 2oi < oD$
DH	pb	$DN <_\Delta NH$	$oD - 2ob < oH$
HA	oh	$HQ <_\Delta QA$	$oH - 2oh < oA$

Table 2.1: Witness of infeasibility

$$pd + pa + oj + og < oc + oi + ob + oh$$

By a similar analysis, we can show that

$$pc + pb + oi + oh < od + oj + oa + og$$

Summing yields $op > 0$. Thus, the midpath tree cannot realize Δ even though some expansion of the midpath tree can.

□

Lemma 2.6 *There are triangle orders which are not realizable at all, despite the existence of midpath tree.*

Proof: Consider the midpath tree of triangle order in figure 2.3. This means we have to contract the edge *op*. Now, we change this triangle order slightly. We expand edge *op* vertically. And all the symmetric midpaths (e.g. M(GH), M(PQ), M(CD), ...) which were on the edges *oh* or *pb* are now moved to this new vertical edge *op*. Note that this midpath tree is binary, so it cannot be expanded any further. The witness of infeasibility in the proof of lemma 5, holds for infeasibility of this midpath tree too. This proves our lemma. □

2.1.6 Unrooted Quartet Consistency(UQC)

We present here a related problem of constructing trees using unrooted quartets, where a quartet is an unrooted tree on four leaves, v_i, v_j, v_k, v_l. Each quartet q is constrained to contain an edge e so that $q - e$ describes a partition of the four leaves into two sets of two leaves each. We indicate this by writing $q = (v_i v_j, v_k v_l)$. The unrooted quartet consistency (UQC) problem is as follows.

Problem: Unrooted Quartet Consistency.

Input: A set Q of quartets on the set of points $S = \{v_1, v_2, ..., v_n\}$.

Question: Does there exist a tree T_Q with leaves labeled by points in S such that if $q = (v_i v_j, v_k v_l) \in Q$, then there is an edge e in T_Q such that v_i, v_j are on one side of e and v_k, v_l are on the other side.

The UQC problem was shown to be NP-complete by Steel [33]. This is shown by reduction to betweenness problem. We shall use this problem to show the NP-hardness of triangle ordinal clustering as well as total ordinal clustering. The counter-example of lemma 2.5, imposes a quartet constraint and shows what edge expansion is needed. The main idea is to superimpose such counter-examples with scaled by different weights so as to generate the required quartet constraints. We need to make sure that these set of counter-examples do not interfere with each other i.e. remaining part of the order (the part not in witness) should stay in the same order after expansion. For generating such weight values for these set of trees we need well-separated numbers of the following section.

2.1.7 Well-Separated Numbers

We call a set of m integers, $a_1, a_2, ..., a_m$, well-separated if each pair of these numbers produces a unique sum and $\forall i \; a_i \leq \; poly(m)$. These numbers could be obtained as $2m^2 i + i^2$. These numbers are composed of two parts $2m^2 i$ and i^2. If two pairs of numbers have the same sum in one part, they cannot have the same sum in the other part. And in the sense of addition, the two parts of the numbers are scaled appropriately so that they do not interfere. Similarly, we define k-weighted set of well-separated integers $a_1, a_2, ...a_m$ such that for all $u, v \leq k$, $ua_i + va_j$ is unique for each quadruple u, v, i, j and $\forall i a_i \leq \; poly(m)$. Note that k is considered to be a constant here. These numbers could be described as base $2k$ numbers. The numbers a_i consists of lower $\log m$

digits from the bits in binary representation of i, $(\log m+1)$-th digit as 1 and leftmost part as ($(\log m+2)$-th digit onwards) $4m^2ki+i^2$. That is, if $b_{\log m}, ..., b_1, b_0$ is the binary representation of i, then a_i is $\sum_{0\leq j\leq \log m}(2k)^j b_j + (2k)^{\log m+1} + (2k)^{logm+2}(4m^2ki+i^2)$. Given $ua_i + va_j$ we can uniquely determine u, v from the right $\log m+1$ digits and then a_i, a_j can be uniquely determined from the remaining part. Hence, each weighted sum is unique. These numbers are polynomial in m although exponential in the highest weight k.

We shall be using these numbers to construct the weights in polynomial time reduction to show the NP-hardness results. This requires the numbers not to be super-exponential. However, numbers exponential in m are would still maintain the polynomial time reducibility. We choose to stick with the polynomial weights though, due to its relation with unweighted (unit-weighted) case. This would imply that if the phylogenetic tree were allowed to have degree two vertices, then even the unit weighted case would be NP-hard.

2.2 Generalized Counter-Example to Midpath Tree Conjecture

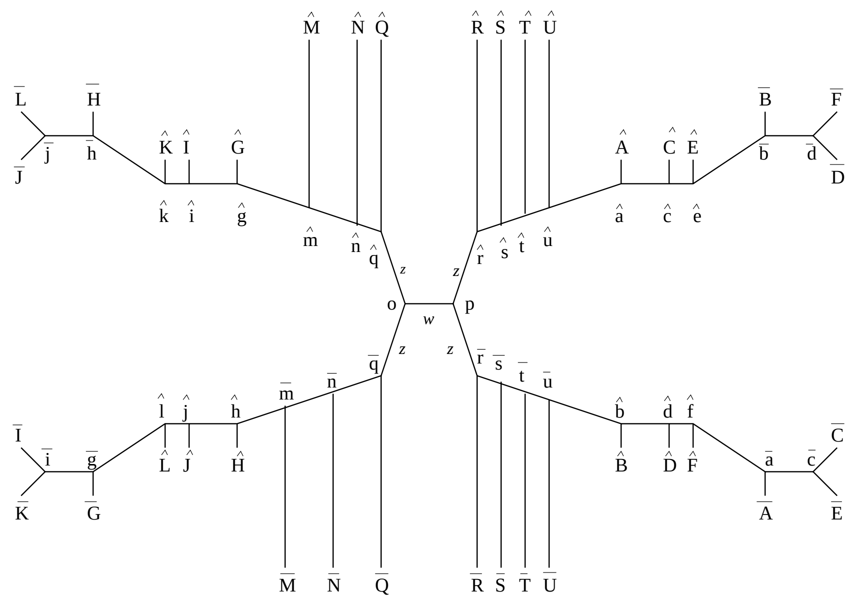

$\hat{M}$ $\hat{N}$ $\hat{Q}$
$\hat{R}$ $\hat{S}$ $\hat{T}$ $\hat{U}$
$\bar{L}$ $\bar{H}$
$\bar{B}$ $\bar{F}$
$\bar{j}$ $\bar{h}$ $\hat{K}$ $\hat{I}$ $\hat{G}$
$\hat{A}$ $\hat{C}$ $\hat{E}$ $\bar{b}$ $\bar{d}$
$\bar{J}$ $\bar{D}$
$\hat{k}$ $\hat{i}$ $\hat{g}$
$\hat{a}$ $\hat{c}$ $\hat{e}$
$\hat{m}$ $\hat{n}$ $\hat{q}$ z
z $\hat{r}$ $\hat{s}$ $\hat{t}$ $\hat{u}$
o w p
z z
$\bar{q}$ $\bar{r}$ $\bar{s}$ $\bar{t}$ $\bar{u}$
$\bar{m}$ $\bar{n}$
$\hat{l}$ $\hat{j}$ $\hat{h}$
$\hat{b}$ $\hat{d}$ $\hat{f}$
$\bar{I}$ $\bar{C}$
$\bar{i}$ $\bar{g}$ $\hat{L}$ $\hat{J}$ $\hat{H}$
$\hat{B}$ $\hat{D}$ $\hat{F}$ $\bar{a}$ $\bar{c}$
$\bar{K}$ $\bar{G}$ $\bar{A}$ $\bar{E}$
$\bar{M}$ $\bar{N}$ $\bar{Q}$
$\bar{R}$ $\bar{S}$ $\bar{T}$ $\bar{U}$

Figure 2.4: Δ not Realizable as Midpath Tree

Figure 2.4 gives a generalized counter-example for the midpath tree conjecture.It consists of a weighted tree T which is a realization of the obvious triangle order generated by the pairwise path distances in T. The weights on the tree are according to the following tables. Here, we show the distances of leaves from respective points o or p for the convenience of analysis. This could fully determine all the edge weights if needed. The weights are in terms of parameters n, m, i which would be defined in section 2.3.

$$op = w$$

$p\hat{A} = x$	$p\hat{B} = x + 2z + 1$
$o\hat{G} = x + 4z + 2 + w$	$o\hat{H} = x + 6z + 3 + w$
$p\hat{C} = x + 8z + 4 + 2w$	$p\hat{E} = x + 14z + 8 + 2w$
$p\hat{D} = x + 10z + 5 + 2w$	$p\hat{F} = x + 12z + 7 + 2w$
$o\hat{I} = x + 16z + 9 + 3w$	$o\hat{K} = x + 20z + 12 + 3w$
$o\hat{J} = x + 18z + 10 + 3w$	$o\hat{L} = x + 22z + 13 + 3w$

Similarly, for distances of $\bar{A}$ to $\bar{L}$ replace x by y.

$$p\hat{r} = p\bar{r} = o\hat{q} = o\bar{q} = z$$

$p\hat{s} = p\bar{s} = 2z + 1.25$	$p\hat{t} = p\bar{t} = 3z + 2.25$	$p\hat{u} = p\bar{u} = 5z + 3.25$

$o\hat{n} = z + 0.75$	$o\hat{m} = 5z + 3.75$	$o\bar{n} = 3z + 1.75$	$o\bar{m} = 7z + 4.75$

for $\hat{A}, , \hat{J}$ and $\bar{A}, \ldots, \bar{J}$ in (alphabetical) order $\hat{a}\hat{A} = 1$. Also,

$\hat{m}\hat{M} = \hat{n}\hat{N} = \hat{q}\hat{Q} = 10000nm + 400ni + 40n$
$\bar{m}\bar{M} = \bar{n}\bar{N} = \bar{q}\bar{Q} = 10000nm + 400ni + 80n$
$\hat{r}\hat{R} = \hat{s}\hat{S} = \hat{t}\hat{T} = \hat{u}\hat{U} = 10000nm + 400ni + 120n$
$\bar{r}\bar{R} = \bar{s}\bar{S} = \bar{t}\bar{T} = \bar{u}\bar{U} = 10000nm + 400ni + 160n$

For parameters, we choose $z = 4n, w = 1, x = 2000ni, y = x + 500n$. For the purpose of this section assume n is much bigger than 1. i could take any value between $1, .., n$. Although, w is chosen to be 1, for any $1 \leq w \leq n$ this counter example is still valid and it maintains the same order. Parameters n and m will be useful in the next

pair	midpoint	inequality	that implies
$\hat{A}\hat{D}$	$\bar{u}\hat{b}$	$\hat{A}\bar{U} < \bar{U}\hat{D}$	$o\hat{A} + 2p\bar{u} < o\hat{D}$
$\hat{D}\hat{E}$	$\hat{s}\hat{t}$	$\hat{D}\hat{S} < \hat{S}\hat{E}$	$o\hat{D} + 2p\hat{s} < o\hat{E}$
$\hat{E}\hat{G}$	$\hat{t}\hat{u}$	$\hat{E}\hat{U} < \hat{U}\hat{G}$	$o\hat{E} - 2o\hat{u} < o\hat{G}$
$\hat{G}\hat{J}$	$\bar{m}\hat{h}$	$\hat{G}\bar{M} < \bar{M}\hat{J}$	$o\hat{G} + 2o\bar{m} < o\hat{J}$
$\hat{J}\hat{K}$	$\hat{n}\hat{m}$	$\hat{J}\hat{N} < \hat{N}\hat{K}$	$o\hat{J} + 2o\hat{n} < o\hat{K}$
$\hat{K}\hat{D}$	$\hat{n}\hat{m}$	$\hat{K}\hat{M} < \hat{M}\hat{D}$	$o\hat{K} - 2o\hat{m} < o\hat{D}$
$\hat{D}\hat{H}$	$\bar{r}\bar{s}$	$\hat{D}\bar{S} < \bar{S}\hat{H}$	$o\hat{D} - 2o\bar{s} < o\hat{H}$
$\hat{H}\hat{A}$	$\bar{q}\bar{n}$	$\hat{H}\bar{N} < \bar{N}\hat{A}$	$o\hat{H} - 2o\bar{n} < o\hat{A}$

Table 2.2: Witness of infeasibility:Inner

pair	midpoint	inequality	that implies
$\hat{B}\hat{C}$	$\hat{t}\hat{u}$	$\hat{B}\hat{T} < \hat{T}\hat{C}$	$o\hat{B} + 2p\hat{t} < o\hat{C}$
$\hat{C}\hat{F}$	$\bar{s}\bar{t}$	$\hat{C}\bar{S} < \bar{S}\hat{F}$	$o\hat{C} + 2p\bar{s} < o\hat{F}$
$\hat{F}\hat{H}$	$\bar{s}\bar{t}$	$\hat{F}\bar{T} < \bar{T}\hat{H}$	$o\hat{F} - 2o\bar{t} < o\hat{H}$
$\hat{H}\hat{I}$	$\hat{m}\hat{g}$	$\hat{H}\hat{M} < \hat{M}\hat{I}$	$o\hat{H} + 2o\hat{m} < o\hat{I}$
$\hat{I}\hat{L}$	$\bar{n}\bar{m}$	$\hat{I}\bar{N} < \bar{N}\hat{L}$	$o\hat{I} + 2o\bar{n} < o\hat{L}$
$\hat{L}\hat{C}$	$\bar{n}\bar{m}$	$\hat{L}\bar{M} < \bar{M}\hat{C}$	$o\hat{L} - 2o\bar{m} < o\hat{C}$
$\hat{C}\hat{G}$	$\hat{r}\hat{s}$	$\hat{C}\hat{S} < \hat{S}\hat{G}$	$o\hat{C} - 2o\hat{s} < o\hat{G}$
$\hat{G}\hat{B}$	$\hat{q}\hat{n}$	$\hat{G}\hat{N} < \hat{N}\hat{B}$	$o\hat{G} - 2o\hat{n} < o\hat{B}$

Table 2.3: Witness of infeasibility:Outer

section and will be introduced then. The counter-example consists of 38 leaves, and no midpoint falls on edge op i.e. $\forall x, y\ M(x, y) \neq op$. Hence edge op is contracted in the midpath tree. Also, no midpoint fall on edges $o\bar{q}, o\hat{q}, p\bar{r}, p\hat{r}$. All other edges have some midpoints on them. The question is : *is it possible to have some other weight assignment of T which will maintain the same triangle order and will have weight of edge $op = 0$?* The linear program generated is infeasible, thus establishing that the answer is no. Following tables 2.2 and 2.3 show the witness to the infeasibility.

table 2.2 implies,

$$p\bar{u} + p\hat{s} + o\bar{m} + o\hat{n} < o\hat{u} + o\hat{m} + o\bar{s} + o\bar{n}$$

table 2.3 implies,

$$p\hat{t} + p\bar{s} + o\hat{m} + o\bar{n} < o\bar{t} + o\bar{m} + o\hat{s} + o\hat{n}$$

adding up,

$$p\hat{t} + p\hat{u} + p\hat{s} + p\bar{s} < o\hat{u} + o\bar{t} + o\bar{s} + o\hat{s}$$

replacing $\hat{A}$ by $\bar{A}$, and similarly,

$$p\bar{t} + p\bar{u} + p\bar{s} + p\hat{s} < o\bar{u} + o\hat{t} + o\hat{s} + o\bar{s}$$

adding up,

$$op > 0$$

This implies that we need to expand the midpath tree, in order to realize the triangle order. Also, this means that any expansion which realizes the triangle order must have an non-zero weighted edge *op*, which imposes, to obtain the tree which realizes the triangle order, the expansion of the midpath tree in which there is an edge which separates leaves $\hat{A}, \hat{B}$ from leaves $\hat{G}, \hat{H}$. In this sense, this (counter-) example imposes a quartet constraint on leaves $\hat{A}, \hat{B}, \hat{G}, \hat{H}$ such that if the triangle order was realizable then there must be a quartet separating edge e such that $T - e$, i.e. tree T cut at the edge e will have $\hat{A}, \hat{B}$ in one component and $\hat{G}, \hat{H}$ in the other. Also, the added advantage of using this (counter-) example over the previous one (lemma 2.5) is that this allows a region around the quartet edge (*op*) where no midpoint falls. This can be used by other super-imposed quartet constraints, to impose further expansions of the tree. We shall see this in the next section.

2.3 NP-hardness of Triangle Ordinal Clustering(TOC)

In this section, we show that given a triangle order Δ, it is NP-hard to determine whether there exists a weighted tree T which realizes Δ. Before we show the reduction, we introduce few definitions. Let T be the weighted tree in the generalized counter example of the previous section.

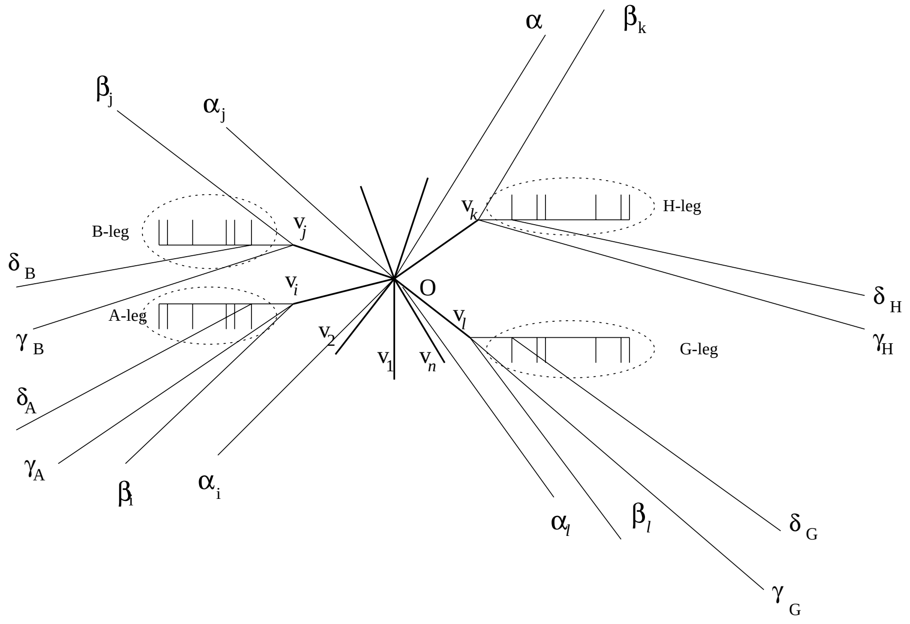


Figure 2.5: Construction

Definition 2.1 *The subtree of T defined by cutting T at edge $p\hat{r}$ and taking the component which contains $\hat{A}$ is called $A-leg$. $B-leg, G-leg, H-leg$ are similarly defined. Note that these are weighted trees which follow the edge weights of T.*

Definition 2.2 *The leaves $\hat{M}, \hat{N}, \hat{Q}, \hat{R}, \hat{S}, \hat{T}, \hat{U}, \bar{M}, \bar{N}, \bar{Q}, \bar{R}, \bar{S}, \bar{T}, \bar{U}$ are classified as dummy points, while others are non-dummy points.*

Definition 2.3 *Given a weighted tree T on the vertex set V and a subset S of V (V consists of internal as well as leaf nodes.) , $T(S)$ is the minimal steiner subtree of T which connects the vertices in S. Moreover, 2-degree vertices in this steiner tree are eliminated, replacing two edges with a new edge. The weight of this new edge is the sum of the weights of the previous two.*

Theorem 2.1 *TOC is NP-hard.*

Proof: The reduction is from UQC problem. Given an instance of UQC, we show how to construct an instance of TOC. Let $I = (S, Q)$ be an instance of UQC. $|S| = n, |Q| = m$. From I we construct a midpath tree T_Δ which would uniquely represent the triangle order Δ. We start with a star on n leaves in $S = \{v_1, v_2, ..., v_n\}$. Assign unit weight to each of these edges. Let the center of these star be o. Attach $2n$ more leaves α_i, β_i with $2n$ new edges of the form $o\alpha_i$ and $v_i\beta_i$ for all $i \in \{1, .., n\}$. Let $W(o\alpha_i) = W(v_i\beta_i) = 200000nm + 8ni$ for each i. α_i, β_i are dummy points.

Now, for each quartet $q_i = (v_{i_1}v_{i_2}, v_{i_3}v_{i_4})$ construct a weighted tree T_i as in the figure for generalized counter example. All of these T_i's have the same structure but they differ in weights according to parameter i. Now, take the four legs $A_i - leg, B_i - leg, G_i - leg, H_i - leg$ and attach these with the edge of weight $z - 1$ to vertices $v_{i_1}, v_{i_2}, v_{i_3}, v_{i_4}$ as in figure []. At the endpoints of each of these attachment edges attach two more leaves (these are also dummy points) $\gamma_{A_i}, \delta_{A_i}$ or $\gamma_{B_i}, \delta_{B_i}$ or $\gamma_{G_i}, \delta_{G_i}$ or $\gamma_{H_i}, \delta_{H_i}$ depending on the leg being attached.
$W(v_{i_1}, \gamma_{A_i}) = W(\hat{r}_i, \delta_{A_i}) = 300000nm + 400mi + 40n$
$W(v_{i_2}, \gamma_{B_i}) = W(\bar{r}_i, \delta_{B_i}) = 300000nm + 400mi + 80n$
$W(v_{i_3}, \gamma_{G_i}) = W(\hat{q}_i, \delta_{G_i}) = 300000nm + 400mi + 120n$
$W(v_{i_4}, \gamma_{H_i}) = W(\bar{r}_i, \delta_{H_i}) = 300000nm + 400mi + 160n$
Once we have done this for all the quartets, the resulting weighted tree is a superimposition of T_i's with their respective edges o_ip_i contracted to a single vertex o. And also they share edges ov_j's according to quartet constraints. Next we define, mid-points over this tree $T^{'}$. For any leaves a, b belonging to same T_i the midpoint $M_{T_i}(a, b)$ is the edge on which the midpoint of weighted path $P_{T_i}(a, b)$ falls. The midpoint $M(a, b)$ is the edge in $T^{'}$ corresponding to $M_{T_i}(a, b)$. If a, b belong to T_i, T_j respectively with $i < j$ then $M(a, b)$ is defined according to weights in $T^{'}$. Note that this means that if a, b are not dummy points,the midpoint will be either of the edges $\hat{u}_j\hat{a}_j, \bar{u}_j\hat{b}_j, \hat{m}_j\hat{g}_j, \bar{m}_j\hat{h}_j$ depending on which leg of T_j b belongs. Now, observe that each non-leaf edge of $T^{'}$ has some midpoint falling on it. So, T_Δ is in fact same as $T^{'}$ along with the midpoint function M. The triangle order Δ is as defined by this midpath tree T_Δ.

Now, the question is: is this midpath tree expandable? i.e. is this triangle order Δ realizable?

The following two lemmas would complete the proof of the theorem. $\square$

Lemma 2.7 *If the triangle order Δ is realizable then answer to UQC question is Yes.*

Proof: Let T be the tree which realizes Δ. T is an expansion of T_Δ. Consider the constraints imposed by each T_i for all $i \in \{1, .., m\}$. These imply that there is an edge e such that $\hat{A}_i, \hat{B}_i$ are on one side of e and $\hat{G}_i, \hat{H}_i$ are on the other. Since o is the only point where such an expansion is possible, this edge e would also separate v_{i_1}, v_{i_2} from v_{i_3}, v_{i_4}. This means all the quartet constraints are satisfied in $T(S)$. □

Lemma 2.8 *If the answer to UQC question is Yes, then Δ is realizable.*

Proof: Consider T_Δ constructed as above. $T_\Delta(S)$ is a star with vertex o in the center and vertices $v_1, v_2, .., v_n$ as leaves. Now, since the answer to UQC question is Yes, consider the expansion of this star which provides solution to UQC problem. Assign a unit weight to each of the newly expanded edges as well as the leaf edges. Note that there are at most n new edges introduced since we are expanding a star on n leaves. Now consider the corresponding expansion in T_Δ, call this tree T. We show that T, indeed, is the tree which realizes Δ. The remaining part of T comes from different T_i's corresponding to each quartet. For weight function on those edges consider, $T(V(T_i))$. Figure 2.6 shows the sketch of $T(V(T_i))$. There is a unique edge, say e_i, separating $\hat{A}_i, \hat{B}_i$ from $\hat{G}_i, \hat{H}_i$. Set the parameter w to weight of e_i (instead of unit) in the corresponding *leg*'s. Adjust the weights of $v_{i_1}\hat{r}_i, v_{i_1}\bar{r}_i, v_{i_3}\hat{q}_i, v_{i_4}\bar{q}_i$ so that distances of $\hat{r}, \bar{r}, \hat{q}, \bar{q}$ from edge e_i equals $z(= 4n)$.

This weighted tree T indeed represents the same triangle order Δ. This can be verified by checking that all the midpoints remain on the same edges, giving the same triangle order. Note that no midpoint falls on the newly expanded parts. For the midpoints of the pair of leaves belonging to the same T_i, they still remain on the same edges giving the same order, since as noted in section 2.2, the value of parameter w is within the range of 1 and n. The midpoints among the pair of dummy points remain at the stem of the dummy point which had larger stem earlier. The midpoints of among the pair of dummy points with equal stems, originally, stay on the same edges. The midpoints of non-dummy points belonging to two different T_i and T_j, $i < j$, remain on the same edge which is one of $\hat{u}_j\hat{a}_j, \bar{u}_j\hat{b}_j, \hat{m}_j\hat{g}_j, \bar{m}_j\hat{h}_j$.

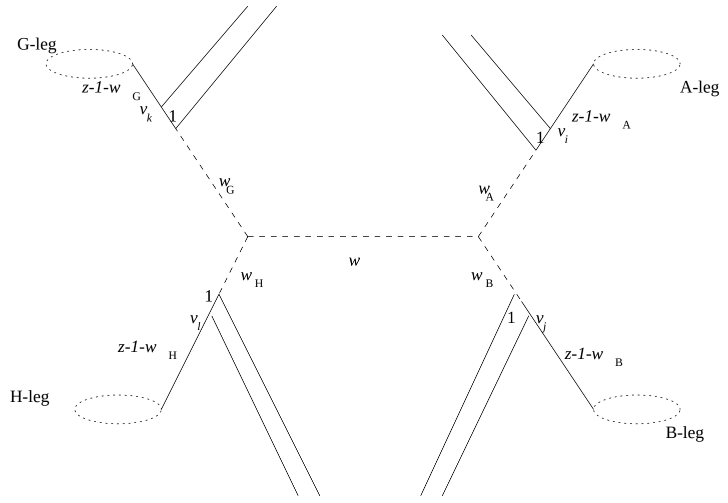


Figure 2.6: UQC implies TOC

Hence, T realizes Δ. □

2.4 NP-hardness of Total Ordinal Clustering (OC)

In this section we show that given a total order τ on pairwise distances it is NP-hard to determine whether there exists a tree T which realizes τ. Again, as in the previous section the reduction is from the UQC problem. Given an instance of UQC, we follow exactly the same construction here, except that the weights involved are slightly different. The parameter z, which was the same $(4n)$ for all T_i's in TOC, is different for different T_i's. So we call it z_i. This is done to force the strict total order on non-critical (not involved in counter example table) distance pairs. We choose z_i's for $1 \leq i \leq m$ such that for $0 \leq p, q, p', q' \leq 400$ we get $|(pz_i + qz_j) - (p'z_{i'} + q'z_{j'})| \geq 1000n$ for any i, j, i', j', unless $i = i', j = j', p = p', q = q'$. These numbers are the the so-called k-weighted well-separated numbers, each multiplied by the factor of $1000n$, and $k = 400$ here. Let z_m be the highest of these z_i's. Then, we define z'_i for $1 \leq i \leq 8m + n$ as the (unweighted) well separated numbers each multiplied by $2z_m$. Now, for each T_i we choose $x_i = 50z_i$ and $y_i = 100z_i$. Each of the distance,

$\hat{m}\hat{M} = \hat{n}\hat{N} = \hat{q}\hat{Q} = z'_{4i-3}$	$\bar{m}\bar{M} = \bar{n}\bar{N} = \bar{q}\bar{Q} = z'_{4i-2}$
$\hat{r}\hat{R} = \hat{s}\hat{S} = \hat{t}\hat{T} = \hat{u}\hat{U} = z'_{4i-1}$	$\bar{r}\bar{R} = \bar{s}\bar{S} = \bar{t}\bar{T} = \bar{u}\bar{U} = z'_{4i}$

While constructing T', which is the weighted tree, we make the following weight changes, $W(o\alpha_i) = W(v_i\beta_i) = z'_{8m+i}$
$W(v_{i_1}, \gamma_{A_i}) = W(\hat{r}_i, \delta_{A_i}) = z'_{4m+4i-3}$
$W(v_{i_2}, \gamma_{B_i}) = W(\bar{r}_i, \delta_{B_i}) = z'_{4m+4i-2}$
$W(v_{i_3}, \gamma_{G_i}) = W(\hat{q}_i, \delta_{G_i}) = z'_{4m+4i-1}$
$W(v_{i_4}, \gamma_{H_i}) = W(\bar{r}_i, \delta_{H_i}) = z'_{4m+4i}$

Apart from these weight changes the construction is exactly the same. Now, any pairwise distance in T' consists of weighted sum of at most two z_i's and at most two z_i''s. In any T_i, the distance of each point from the center edge o_ip_i, differs in the weight of z_i. Hence, each distance in T', which has two end point in different T_i's, has unique composition in terms of z_i's,z_i''s and their respective weights. The well-separation property guarantees that any pair of distances, if it doesn't have all 4 end points in the same T_i, differs by at-least $500n$. This is because the component of the distances due to w_i's as well as the constant part is much smaller than the one due to z_i's and z_i''s. (Note that any distance in this tree consists of the weighted sum of four parts: w, z, z' and a constant part.)

Now, to define the total order we first give the pairwise distance values and then the total order will be defined by these values. For any two leaves v_p, v_q belonging to the same T_i, the distance $dist(v_p, v_q) = d_{T_i}(v_p, v_q)$ and if they belong to different $T_i's$ then $dist(v_p, v_q) = d_{T'}(v_p, v_q)$. Again, we define midpoints on T'. For any leaves v_p, v_q belonging to same T_i the midpoint $M_{T_i}(v_p, v_q)$ is the edge on which the midpoint of weighted path $P_{T_i}(v_p, v_q)$ falls. $M(v_p, v_q)$ is the edge in T' corresponding to $M_{T_i}(v_p, v_q)$. If v_p, v_q belong to different $T_i's$ then $M(v_p, v_q)$ is the midpoint of the weighted path $P_{T'}(v_p, v_q)$ in T'. Note that T' taken as unweighted tree is indeed the midpath tree T_τ with midpath function defined as above. Also, because of the well-separation property, no midpoint is within the distance $100n$ from the center point o in T'.

The proof of lemma 2.7 is applicable as it is over here. So if the total order is realizable then the answer to the UQC question is Yes.

For proving the other way round again we do the same construction as in the proof of lemma 2.8. If the answer to the UQC question is Yes, then we expand the center vertex o according to the UQC solution and assign unit weight to each of these newly expanded edge. Then, for each T_i, take w as the number of edges that separate vertices v_{i_1}, v_{i_2} from v_{i_3}, v_{i_4}. Note that this will also change the corresponding weights in T'. Now, no distance in T' changes by more than $10n$ (since $w \leq n$) by this expansion and corresponding increments in w values. So, due to well-separatedness, the same order is maintained for the distance pairs which do not consist of four leaves in the same T_i. For distance pair within the same T_i, whether $w = 1$ or w is number between 1 and n maintains the same order. Also, the order of distances involving $\alpha_i, \beta_i, \delta_A, \gamma_A, ...$ (dummy points not belonging to any T_i) remains unaltered during the expansion. Hence this expanded weighted tree T, represents the desired total order.

Theorem 2.2 *Total Ordinal Clustering (OC) is NP-hard.*

□

Chapter 3

Undiscretized Dynamic Programming: Faster Algorithms for Filtering

In this chapter, we consider the minimizing delay from [30] and show that the complexity of the dynamic programming algorithm can be improved.

In section 3.1 , we show the formulation of the problem. We discuss the preliminaries and assumptions. We go over the dynamic programming algorithm described in [30] and give the intuition which motivates faster algorithms. In sections 3.2 and 3.3, 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 . The detailed description of data structures and operation involved is given in chapter 5.

3.1 Preliminaries

3.1.1 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 of information that node v gets from its parent, if the parent has a filter. We assume that $f(v)$ for each v is known.

3.1.2 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 (typically) a big constant.[1]
- 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.

3.1.3 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 the number of leaves is a constant for a given multicast tree M, we can think of minimizing the total delay, where the 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 $Tree(v)$, assuming $A(v) = p$.

[1]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[34]. This will not change the algorithm.

Let $v_1, v_2, \ldots, v_{c(v)}$ be the 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 in table 3.1.

```
if $v$ is a leaf then
    $D(v,p) = 0$ for all $p$
else
    $D(v,p) = \min\{$
        $C_v + \sum_{i=1}^{c(v)} D(v_i, v_i)$, if $v$ has a filter
        $f(p).E_v + \sum_{i=1}^{c(v)} D(v_i, p)$, otherwise
    $\}$
end if
```

Table 3.1: Discrete Dynamic Programming

The optimal placement can be found using dynamic programming as noted in [30]. However, 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 as in table 3.2

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

Table 3.2: Undiscretized Dynamic Programming

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 the 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 3.1 *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[2] 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 3.2 $|D_v| \leq n(v)$.

[2]Height of v = 1+ max{Height of u | u is a child of v}. Height of a leaf is 0.

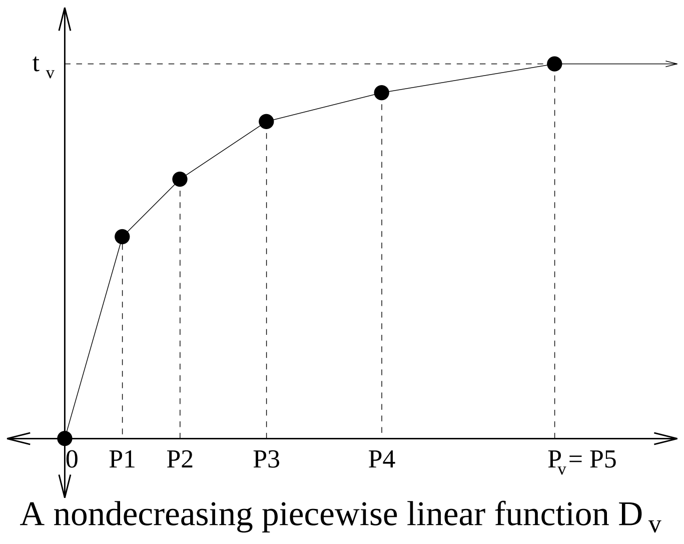


Figure 3.1: Piecewise Linear Function

Proof: By induction on height of v. Claim is trivially true for v if it is 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.

3.2 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.

```
Algorithm-1 {
    A($r$);
    M-DFS($r$, $I$);
}
M-DFS ($v$, $p$) {
    if $c(v) == 0$ then
        return;
    end if
    if $p > p_v$ then
        place filter at $v$;
        for $i = 1$ to $c(v)$ do
            M-DFS($v_i$, $f(v_i)$);
        end for
    else
        for $i = 1$ to $c(v)$ do
            M-DFS($v_i$, $p$);
        end for
    end if
    return;
}
```

```
A($v$) {
    if $c(v) == 0$ then
        $p_v = +\infty$;
        return create(0,0);
    else
        for $i = 1$ to $c(v)$ do
          $q_i = A(v_i)$;
        end for
        $t_v = C_v$;
        for $i = 1$ to $c(v)$ do
          $t_v = t_v$+probe($q_i$,$f(v_i)$);
        end for
        $z$ = create($E_v$, 0);
        for $i = 1$ to $c(v)$ do
          $z$ = add_merge($z$, $q_i$);
        end for
        $p_v$ = truncate($z, t_v$);
        return $z$;
    end if
}
```

Table 3.3: Undiscretized Algorithm

3.2.1 Algorithm

3.2.2 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_1, q_2)**:** Returns a piecewise linear function which is the sum of q_1 and q_2. 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_1 and q_2 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.

3.2.3 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 3.3 *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 $D_{v_1}, D_{v_2}, \ldots, D_{v_{c(v)}}$ sequentially. Let $s_i = \sum_{j=1}^{i} n(v_i)$. 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 $n(v) \log n(v)$. □

3.3 Algorithm-2

3.3.1 Motivation

We observe that lemma 3.2 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 [30]. 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 the 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 larger than H.

3.3.2 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 $X = x_1, x_2, ..., x_k$ 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 $X_1 \bigcap X_2$.

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: $O(\log |f_X|)$.

3.3.3 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 $v_1, v_2, \ldots, v_{c(v)}$, have less than H breakpoints in their respective data-structures. We now start building D_v by merging $D_{v_1}, D_{v_2}, \ldots$ one by one. Suppose after merging D_{v_1} through D_{v_i} for the 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 $j \in \{i+1, \ldots, c(v)\}$. 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.

3.3.4 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 3.2.$|Y| \leq \frac{N}{H}$

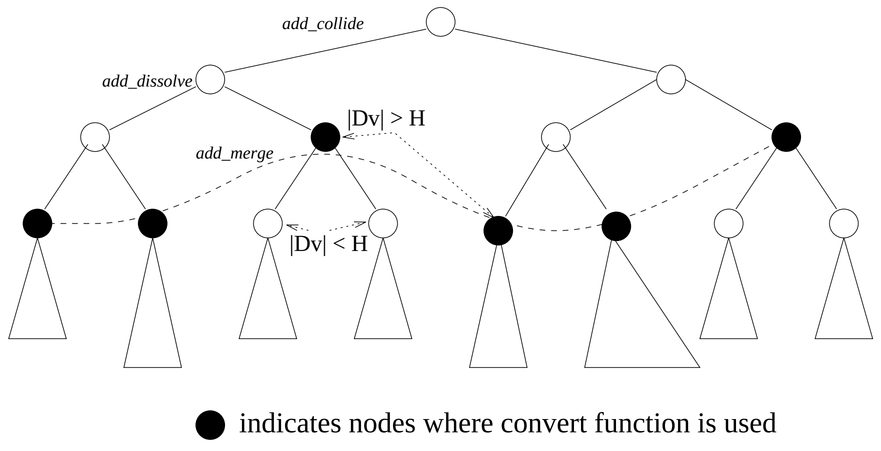


Figure 3.2: Three types of add operations

because for any two $u, v \in Y$, $Tree(u) \bigcap Tree(v) = \phi$.

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

For all nodes $v \in U$, we will run the normal *add_merge* procedure. Since final D_v's constructed for each $v \in U$ have sizes less than H, the total cost of building them would be $|U| \log H \leq N \log H$.

For each node $v \in Y$, we do a few *add_merge* operations, followed by a *convert* operation, followed by few *add_dissolve* and *add_collide* operations. Let X be the sorted set of $f(p)$ values for the ancestors of v. $1 \leq |X| \leq H$. Since the overall effect of *add_merge* operations leads to a data-structure of size at most $2H$, total cost is $O(H \log H)$. *convert* will cost at most $|X| \log \frac{2H}{|X|} = O(H)$.

If we sum over all *add_dissolve*s performed during the run of the algorithm, it is easy to see that at most N breakpoints will be *dissolve*d in data-structure of size H. So the total cost of *add_dissolve*s is at most $N \log H$.

Further, there are at most N/H "converted" functions, and each *add_collide* takes $O(H)$ time and causes one less "converted" function. Hence, the total cost of *add_collide* is $O(N)$.

Thus, the overall cost is at most $N \log H + \frac{N}{H} \cdot (H \log H + H) + N \log H + N$, that is $O(N \log H)$.

Theorem 3.1 *Given the multicast tree M having N nodes and height H with source at root r disseminating I amount of information, along with values $f(v)$ which is the minimum amount of information required at node v of the multicast tree, the placement of filters in the multicast tree to minimize total delay can be computed in $O(N \log H)$ time.*

$\square$

Chapter 4

Faster Algorithms for Facility Location on Trees and Related Problems

In this chapter we extend the undiscretization technique from chapter 3 to uncapacitated facility location problem on trees. In section 4.1, we define our notation and describe the $O(n^2)$ dynamic programming algorithm for the problem given by [34]. We also show the "undiscretization" of the dynamic programming functions using the techniques of [28] and prove some necessary lemmas. In section 4.2 we describe the algorithm. In section 4.3 we list the data structure operations and their complexity and in section 4.4 we derive the complexity bound for the algorithm. Again, the detailed description of data structure and operations can be found in chapter 5.

4.1 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 [34]. 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 and hence does not affect complexity. Hence, for the rest of the chapter, 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 [34] 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. These are

```
**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**
```

Table 4.1: Dynamic Programming Algorithm

simply Tamir's [34] dynamic programming recurrences written in terms of the "undiscretized" parameter x.

Lemma 4.1 *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, $|F_v| \leq s_v$. $\square$

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 4.2 *G_v is a piecewise non-increasing step function(PDSF) with number of*

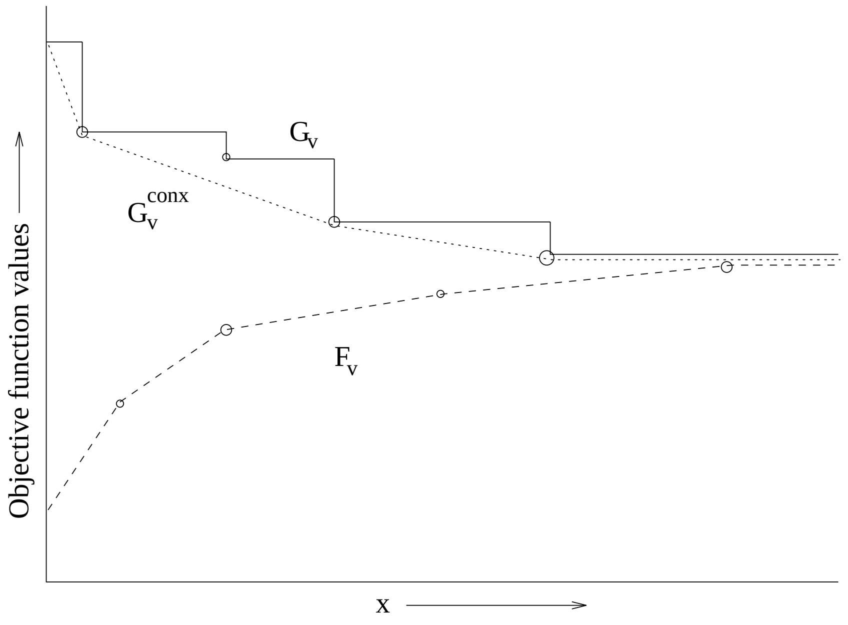


Figure 4.1: Undiscretized Functions

steps $|G_v| \leq s_v$.

Proof: The only values of x where $G_v(x)$ can change value are those where 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 4.1 illustrates these three functions.

Lemma 4.3 *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 4.4 *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 the 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 $x > t$. □

4.2 Algorithm

We are now ready to present the algorithm to compute the functions defined in section 4.1. 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 $s_l \geq s_r$.

The algorithm presented in table 4.2 above is nothing but a translation of the dynamic programming recurrences shown in section 4.1. The procedures in the algorithm

```
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 4.2: Undiscretized 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.

4.3 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 chapter 5.

$createG(c)$**:** 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, 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 (subtract from x coordinate) 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.

4.4 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 4.1 *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 $O(s_v \log s_v)$. □

Chapter 5

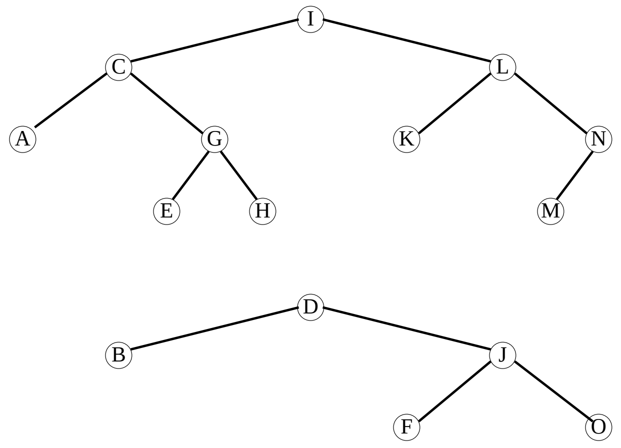
Data Structures and Operations

Here, we describe the data structures used to store piecewise linear functions involved in each of the algorithm previously discussed. We also show how the operations defined on these data structures are carried out and give the complexity bound for each operation.

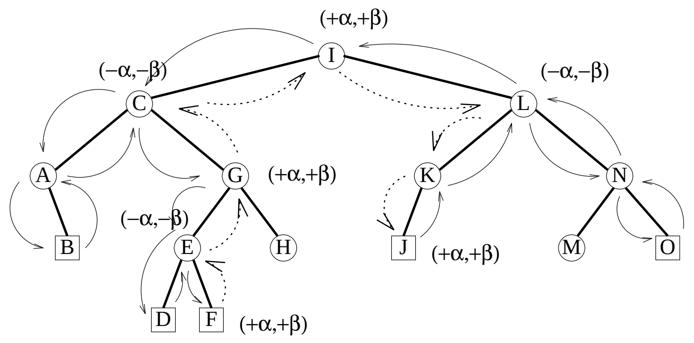
Particularly, we describe how to store piecewise linear functions, the "converted" functions and the functions F_v and G_v^{conx} which are PLNCF and PLDXF respectively. The main data-structure is a height balanced binary search tree. We shall use AVL trees [1, 4, 11] which can be merged fast using Brown and Tarjan's fast merging algorithm [7].

5.1 Fast Merging Algorithm

Brown and Tarjan [7] 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 $m \geq n \geq 2$



Sorted lists represented as height-balanced trees



Merging by sequential insertions (square nodes have been inserted)

Figure 5.1: Fast Merging Algorithm

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.[1]

5.2 Data Structure for Filtering Algorithms

In the previous sections, we have assumed the existence of a data structure to maintain non-decreasing piecewise-linear functions. Here, we describe the data structure along

[1]We had independently proved the amortized $O(n \log(m/n))$ bound for the fast merging using $2-3$ trees which is sufficient for our purposes.

with the implementation of the operations.

The data structure will maintain the breakpoints (or each value in X for a converted function) sorted by x coordinate in an AVL tree [1, 11]. An AVL tree is a balanced binary search tree in which for any node, the height difference between its left and right subtrees is at most one. Along with the x coordinate of the breakpoint, each node will also contain two real numbers a and b such that the linear segment to the left of the breakpoint is of 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. 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.

The data structure will use the AVL tree merging algorithm of Brown and Tarjan [7] to implement add_merge, convert and add_dissolve. 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 α and β 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 a right child after some (non-empty) sequence of left child choices, we subtract the tuple (α, β) at that node. Similarly, whenever we choose the left child after a (non-empty) sequence of right child choices, we add the tuple (α, β) to the node where choice is made. 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 the rotations and double rotations in the tree for the AVL insertion. The figure 5.1 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 one node at $x = +\infty$, and set its a and b values.

add_merge(f_1, f_2)**:** 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, find the linear segment for which $Az + B = t$, and thus find z. 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_X, g)**:** Just like in *add_merge*, but do not insert the breakpoints, just update the a and b values of the existing breakpoints.

add_collide(f_{X_1}, g_{X_2})**:** Find the values of f and g on $X_1 \bigcap X_2$ and construct a new AVL tree as in *convert*.

truncate_converted(f_X, t)**:** As in *truncate*, find z. But in this case do not insert the new break-point. Also do not delete the break-points in f_X after z. Invalidate the (a, b) values of the remaining points by marking the right child of the nodes traversed from the left and also 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, walk down to $+\infty$, validating the a and b values on that path. Then set the a and b values at $+\infty$ such that $A = 0$ and $B = t$. It returns z.

5.3 Data Structure for 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 break-point 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 chapter 4 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 in the data structure for F_2 and list all the equations of the line segments in increasing order. Subtract $(x_2^{off} - x_1^{off})$ from each x coordinate and for each line segment $Ax + B$, 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, the addition of 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 is required. 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.

5.4 Data Structure for PLDXF

Here, again we maintain the breakpoints of PLDXF G in the AVL tree. Also, the tuple (a, b) is stored along with its x value. However, unlike PLNCF, the tuple 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, delete points from behind (right to left) till reach a point u, the point to the left of which has higher y value. Then, 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 go rightwards and delete all the points which have y values higher than u. Now, traverse leftwards in the AVL tree and check the points in decreasing order of x coordinates. 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 delete the breakpoint to the right of that segment. And continue, till we find the decreasing slope. Then stop. If the inserted point is a leftmost point then 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. Figure 5.2 illustrates the convexifying step involved.

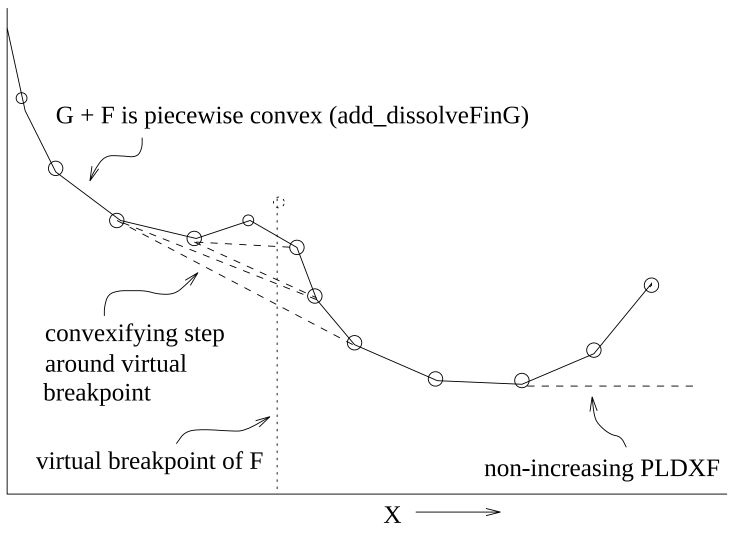


Figure 5.2: add_dissolveFinG

Chapter 6

Covering and Lot Sizing Problems

In this chapter, we consider some of the facility location related problems to which our method of undiscretization applies. We do not separately describe the data structure operation for these because they are almost same as those found in the previous algorithms.

6.1 Facility Constrained Covering Problem

This problem was first studied by Kolen [27]. 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 $T - T_v$. 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. 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. The figure 6.1 shows illustrates these functions.

6.1.1 Undiscretized Dynamic Programming for FCC

Let 0^- denote any number less than 0. So here, $F_v(0^-) = G_v(0^-)$. Let $x_1, x_2, .., x_{s_v}$ be sorted order of possible values of distances of influence covered outside T_v by each

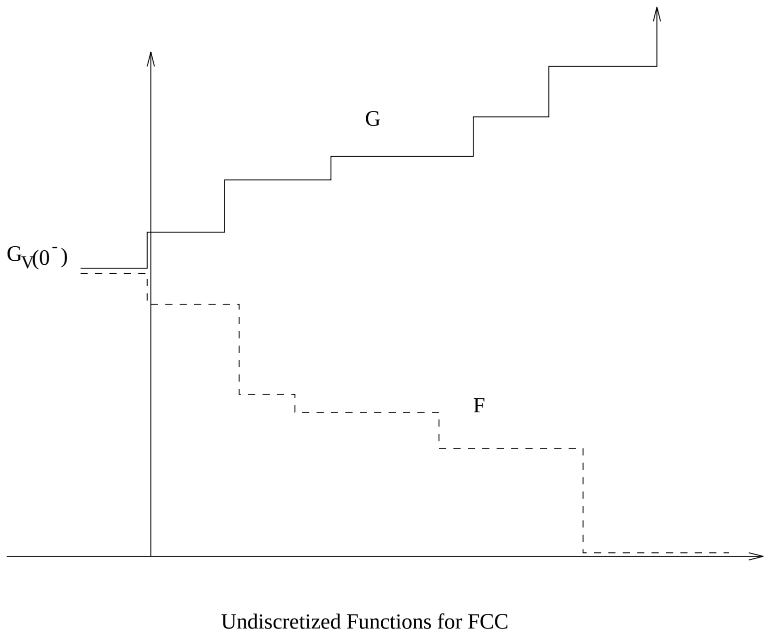


Figure 6.1: Undiscretized Functions for FCC

facility in T_v. These form breakpoints in G_v. Let $x_i = s_v$. Table 6.1 gives the dynamic programming recurrence.

Here, all the slopes are zero so equation of each line in the data structure has just one constant value. We don't need any convex hull function here. Only when G and F are added we need to do monotonizing step around each insertion to ensure that G remains monotonic non decreasing. The undiscretized algorithm is just the same as given in table 4.2. There are no add_line functions here. However there are two breakpoints introduced at each node of tree during the dynamic programming algorithm. Hence the number of breakpoints in F_v or G_v are bounded above by $2s_v$ and not s_v

6.2 Customer Constrained Covering Problem

This problem is also due to Kolen [27] 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 chapter 4. 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. Here, instead of add_line

```
**if** $v$ is a leaf **then**
  $G_v(x) = \infty$ for $x > s_v$
  $G_v(x) = f_v$ for $0 \leq x \leq s_v$
  $G_v(0^-) = \min\{q_v, G_v(0)\}$
  $F_v(x) = 0$ for $x \geq 0$
  $F_v(0^-) = G_v(0^-)$
**else**
  $G_v(0^-) = \min\{G_v(x_1), q_v + G_l(0^-) + G_r(0^-)\}$
  $G_v(x_i) = \min\{G_v(x_{i+1}), f_v + F_l(x_i - x_l) + F_r(x_i - x_r)\}$
  $G_v(x_k) = \min\{G_v(x_{k+1}), G_l(x_k + x_l) + F_r(x_k - x_r)\}$
                              whenever $x_k$ corresponds to facility in $T_l$
  $G_v(x_k) = \min\{G_v(x_{k+1}), G_r(x_k + x_r) + F_l(x_k - x_l)\}$
                              whenever $x_k$ corresponds to facility in $T_r$
  $G_v(x) = G_v(x_k)$ whenever $x_{k-1} < x < x_k$
  $F_v(x) = \min\{G_v(0^-), F_l(x - x_l) + F_r(x - x_r)\}$

**end if**
```

Table 6.1: Undiscretized Dynamic Programming for FCC

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

Table 6.2: Undiscretized Algorithm for FCC

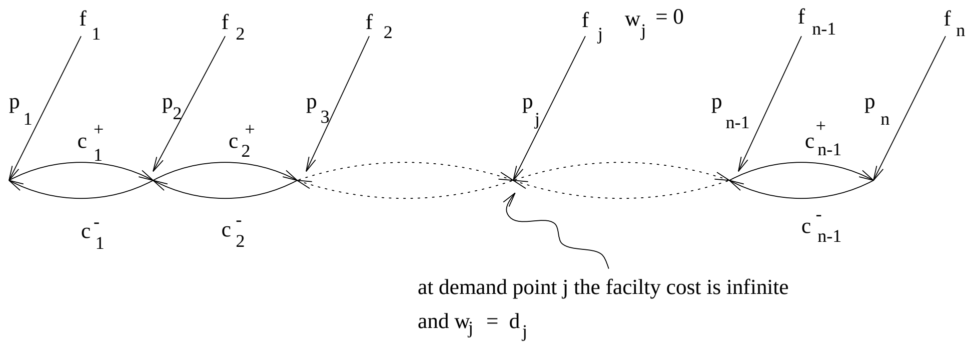


Figure 6.2: ELS as UFL

functions in UFL case, we consider penalty costs. The rest of the algorithm is similar and hence we will skip the description.

6.3 Economic Lot Sizing Problem

We note here that, the same improvement on Economic Lot Sizing (ELS) Problem, was earlier achieved by Aggarwal and Park using a different technique to speed up one-dimensional dynamic programming. In ELS, there is a 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[20], with the distance function on each edge being c_j^+ in one direction and c_j^- in the other.

As in UFL formulation, here,

$$c_{ij} = (p_j + c_j^+ + ... + c_{i-1}^+)d_i \ if \ i \geq j$$

$$c_{ij} = (p_j + c_j^- + ... + c_{i+1}^-)d_i \ if \ i < j$$

The figure 6.2 shows how ELS can be regarded as UFL on a path like tree with different costs in two directions.

$G_v(x)$ 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.

We achieve a complexity bound of $O(n \log n)$ and when p_j's satisfy certain restrictions we even get an $O(n)$ algorithm. Note that the tree structure is almost like a path so there are no merge operations involved in the undiscretized algorithm.

6.3.1 Undiscretized Dynamic Programming Algorithm

Let $F_j(x)$ be optimal subproblem solution for time periods $j, j+1, .., n$ assuming that if the i facility among $1, .., j-1$ which could supply for j-th period then $x = p_i + c_i^+ + ... + c_{j-1}^+$. Let $G_j(x)$ be optimal subproblem solution for time periods $j, j+1, .., n$ assuming that there is a production in time period i, $i \geq j$ such that $p_i + c_i^- + ... + c_{j+1}^- \leq x$. G_j is a PDSF and F_j is a PLNCF. $F_j(\infty) = G_j(\infty)$ and we want to compute $G_1(\infty)$. Let sorted order of $p_j, c_j^- + p_{j+1}, ..., c_j^- + c_{j+1}^- + ... + c_{n-1}^- + p_n$ be $x_1, x_2, x_3, ..., x_{n-j+1}$ Let k be the index such that $x_k = p_j$.

```
$G_n(x) = \infty$ for $x < p_n$
$G_n(x) = f_n + d_n p_n$ for $x \geq p_n$
$F_n(x) = \min\{d_n x, f_n + d_n p_n\}$
$G_j(x_i) = \min\{G_j(x_{i-1}), G_{j+1}(x_i - c_{j+1}^-) + d_j x_i\}$ for $i \neq k$
$G_j(x_k) = \min\{G_j(x_{k-1}), f_j + d_j p_j + F_{j+1}(p_j + c_j^+)\}$
$F_j(x) = \min\{G_j(\infty), x d_j + F_{j+1}(x + c_j^+)\}$
```

Table 6.3: UDP for ELS

Again, we define the function G^{conx}. We first construct G_n^{conx} and F_n. To construct G_j^{conx} and F_j from G_{j+1}^{conx} and F_{j+1} we do the following operations:

```
$g^0 = d_j p_j + probeF(F_{j+1}, p_j + c_j^+)$
$G^1 = shiftG(G_{j+1}^{conx}, c_{j+1}^-)$
$G^2 = add\_lineG(G^1, d_j)$
$G_j^{conx} = add\_point(G^2, (p_j, g^0))$
$g^{inf} = probeG(G_j^{conx}, \infty)$
$F_j = shiftF(F_{j+1}, c_j^+)$
$F_j = add\_line\_pruneF(F_j, d_j, g^{inf})$
```

Table 6.4: Undiscretized algorithm for ELS

6.3.2 Analysis

The data structure used to store these function is exactly the same as data structure for UFL. Only new functions which appear here are $shiftG$ and $shiftF$ which appropriately shift the x coordinate of the function. Each operation is a constant time operation except for *probeF* and *add_pointG* which take $O(\log n)$ time. Hence, the complexity of entire algorithm is $O(n \log n)$. However, if p_j follow the restricted coefficient model of [2] which is $p_j \leq c^+_{j-1} + p_{j-1}$ and $p_j \leq c^-_{j+1} + p_{j+1}$ then we can prove that all the probes in F are come in increasing order and all insertions in G are in decreasing order. This would give us linear time algorithm. Also, we could get the linear time algorithm similarly if the fixed cost f_j's are a constant independent of j.

Chapter 7

Conclusions and Future Work

Here, we present some discussion and conclusions related to the problems considered in this thesis. We also list some related problems which could be considered in the future along the similar lines.

7.1 Ordinal Clustering

7.1.1 Ordinal Clustering in other Metric Spaces

In this thesis, we considered the problem of embedding orders into trees. There are other related metric spaces closely related to trees, where embedding orders could give rise to interesting problems. For example on a path (or line), the problem can be solved in polynomial time by mid-path tree algorithm (note that the path is subcase of trees). However, if it is a partial order i.e. incompletely specified orders then again the problem on path can be shown to be NP-complete by reduction to betweenness problem. In euclidean space, this problem could be determined by semi-definite programming. Also orders could be embedded into l_∞ metric space, because any distance metric can be isometrically embedded into l_∞. It remains an interesting question to determine the lowest dimension of the target space (euclidean or l_∞) required to embed the order. In this sense we could define a property called dimensionality of an order. Since l_1 metric spaces are a superset of tree (additive) metric spaces, the methods involved here could give a good intuition for embedding orders into l_1.

7.1.2 Remarks and Future Work

Our result, in a way, ends the quest for constructing a weighted phylogeny from orders on a negative note (unless $P = NP$). There are some approximation criteria which could be considered. For example, dropping out the minimum number of leaves so that the order becomes embeddable or finding a tree embedding with the least number of inversion pairs in the order. All these criteria would become NP-hard, but one could seek some approximations there.

Also, this is the first kind of NP-hardness result known in phylogeny construction where the input data is fully specified. Most similar NP-hardness results known were for tree construction from an incomplete distance matrix or erroneous data or incompletely specified orders. Also, this is the first result where weighted phylogenies are addressed for representing orders.

7.2 Multicast Filtering Problem

From the practical point of view, proposed work for the future includes further investigations into aspects of this problem such as costs associated with the instantiation and migration of filters as well as consideration of heuristics for some practical topologies. An actual implementation using Aglets technology is also something to be done. These are not considered by [30].

Also being considered is the problem of constructing an application layer filter tree, given the user subscription pattern. This would form some kinds of common interest clusters and then the multicast tree is constructed in accordance with the information requests from these clusters.

On the algorithmic aspect the future work is to design an algorithm based on similar methods for the first model of [30]. Here, the dynamic programming algorithm runs in $O(pn^2)$ time, since the objective function at each node involves two parameters. One way of extending the second model (uncapacitated) to the first one is to try and vary the value of filter cost F and check how many filters our $O(n \log h)$ algorithm places in the network. Binary search can be used to get exactly p filters, if we know that for each k there is a value F, such that when the filter delay is F there are exactly k filters deployed. This would lead to $O(n \log h \log T)$ algorithm, where T is the total traffic on the network assuming no filtering. This might be better than $O(pn^2)$ in many cases. Particularly, if p is large (some constant fraction of n) then the dynamic programming algorithm may take time cubic in the size of input, while refinement of our technique can give an algorithm which runs in roughly quadratic time. However, if the traffic values are very big, then one may not like to have dependency of algorithm on T. The future work is to design the faster algorithm in this regard. 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.

7.3 Facility Location

Along the similar lines, the extension of faster algorithm for UFL, to develop faster algorithm for k-median on trees is an interesting area of future work. Another generalization of UFL was given by Tamir [34] 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. [34] 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. Also, related generalization is the k-forest problem [35].

Also, extending the $O(n \log n)$ algorithm for classical UFL on tree to the generalization, where the transportation costs are some concave non-decreasing functions of tree distances instead of linear, is a promising direction.

7.4 Speeding Up Dynamic Programming

There have been several algorithmic advances in speeding up typical dynamic programming algorithms[17, 16, 14, 2]. Many of these work on one-dimensional problems. Galil and Park [16] improve $O(n^2)$ one-dimensional dynamic programming algorithm to $O(n)$

assuming the concavity of the cost function. On the similar lines $O(n)$ algorithm for UFL on line was given by Hassin and Tamir [20]. We have to note that our improvement for ELS was earlier achieved by Aggarwal and Park [2]. Their algorithm also was the one which speeds up one dimensional dynamic programming on path. They exploit fast searching techniques in Monge arrays (matrix) to achieve the improvement.

For higher dimensional dynamic programming problems, the speeding algorithms, typically, exploit sparsity. For example, Tamir's [34] algorithm for k-median problem, analytically uses sparsity of tree structure and proves $O(kn^2)$ bound for brute-force $O(k^2n^2)$ dynamic programming algorithm. [17] gives improvements in dynamic programming for one-dimensional as well as higher dimensional dynamic programming problems using convexity, concavity and sparsity.

Our approach, in this sense, generalizes the one-dimensional path dynamic programming speeding approach to one-dimensional tree dynamic programming speed-up by exploiting Brown and Tarjan's [7] fast merging algorithm. Since tree is a generalization of path our technique also works for the one dimensional dynamic programming improved earlier, like in the case of ELS. To the best of our knowledge, this is the first time the dynamic programming algorithms over tree structures are improved in general.

References

[1] G. Adel'son-Vel'skii and Y. Landis. An algorithm for the organization of information. *Soviet Math. Dokl.*, 3:1259–1262, 1962.

[2] A. Aggarwal and J. Park. Improved algorithms for economic lot size problem. *Operations Research*, 41(3):549–571, May-June 1993.

[3] M. Aguilera, R. Storm, D. Sturman, M. Astley, and T. Chandra. Matching events in a content-based subscription system. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 53–61, 1999. http://www.research.ibm.com/gryphon/matching.pdf.

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

[5] F. Anjum and R. Jain. Generalized multicast using mobile filtering agents. Technical report, Telcordia Technologies, March 2000.

[6] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Storm, and D. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, 1999.

[7] M. Brown and R. Tarjan. A fast merging algorithm. *Journal of ACM*, 26(2):211–225, April 1979.

[8] P. Buneman. *Additive evolutionary trees*, pages 387–395. University Press, Edinburgh, 1971.

[9] A. Carzaniga, D. Rosenblum, and A. Wolf. Design of scalable event notification service: Interface and architecture. Technical Report CU-CS-863-98, University of Colorado, Department of Computer Science, 1998.

[10] G. Cornuejols, G.L. Nemhauser, and L.A. Wosley. The uncapacitated facility location problem. In *Discrete Location Theory (P. B. Mirchandani and R. L. Francis eds)*, pages 119–171, New York, 1990. Wiley.

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

[12] J. C. Culberson and P. Rudnicki. A fast algorithm for constructing trees from distance matrices. *Information Processing Letters*, 30, 1989.

[13] W. H. E. Day. Inferring phylogenies from dissimilarity matrices. *Bulletin of Mathematical Biology*, 49(4), 1987.

[14] D. Eppstein, Z. Galil, and R. Giancario. Speeding up dynamic programming. In *Proc. 29th Symp. Foundations of Computer Science*, pages 488–496, October 1988.

[15] M. Farach, S. Kannan, and T. Warnow. A robust model for finding evolutionary trees. *Algorithmica*, 13(1), 1995.

[16] Z. Galil and K. Park. A linear-time algorithm for concave one-dimensional dynamic programming. *Information Processing Letters*, 33:309–311, 1989-90.

[17] Z. Galil and K. Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science*, 92:49–76, 1992.

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

[19] E. Gimadi. Efficient algorithm for solving the plant location problem for serving regions connected with respect to an acyclic network. *Upravlyaemye Sistemy*, 23:12–23, 1983. (Russian).

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

[21] J. Hein. An optimal algorithm to reconstruct tree from additive matrices. *Bulletin of Mathematical Biology*, 51, 1989.

[22] S. Kannan. Personal Communication.

[23] S. Kannan and T. Warnow. Tree reconstruction from partial orders. *SIAM Journal of Computing*, 24(3):511–519, June 1995.

[24] S. Kasera, S. Bhattacharya, M. Keaton, D. Kiwior, J. Kurose, D. Towsley, and S. Zabele. Scalable fair reliable multicast using active services. *IEEE/ACM Transactions on Networking*, 3:294–310, 2000.

[25] P. Kearney. A six-point condition for ordinal matrices, 1995. Manuscript.

[26] P. Kearney, R. Hayward, and H. Meijer. Phylogenies from relative dissimilarity. *Algorithmica: Special issue on computational biology*, 25:196–221, 1999.

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

[28] S. Langerman, S. Lodha, and R. Shah. Algorithms for efficient filtering in content-based multicast. In *Algorithms - ESA 2001*, volume 2161 of *Lecture Notes in Computer Science*, pages 428–439. Springer-Verlag, 2001.

[29] D. Robak. Untitled document. Personal Communication.

[30] R. Shah, F. Anjum, R. Jain, and S. Rajagopalan. Mobile filters for efficient dissemination of personalized information using content-based multicast. Technical Report 2001-20, DIMACS, 2001.

[31] R. Shah and M. Farach-Colton. On the midpath tree conjecture: A counter-example. In *Proceedings of Symposium on Discrete Algorithms (SODA)*, 2001.

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

[33] M. A. Steel. The complexity of reconstructing trees from qualitative characters and subtrees. *Journal of Classification*, 9, 1992.

[34] A. Tamir. An $o(pn^2)$ algorithm for the p-median and related problems on tree graphs. *Operation Research Letters*, 19:59–64, 1996.

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

[36] M. S. Waterman, T. F. Smith, M. Singh, and W. A. Bayer. Additive evolutionary trees. *Journal of Theoretical Biology*, 64, 1977.

[37] H. Zhou and S. Singh. Content based multicast in ad-hoc networks. In *Proceedings of First Annual Workshop on Mobile and Adhoc Networking and Computing*, pages 51–60, 2000.

Vita

Rahul Shah

1991-97	National Talent Search (NTS) scholarship, instituted by the National Council for Educational Research and Training (NCERT), New Delhi, India.
1993	Ranked Ninth in all over India in the Joint Entrance Exam (JEE) for IITs.
1993-95	Institute academic award winner for academic excellence at Indian Institute of Technology, Mumbai, India.
1993-97	B.Tech. in Computer Science and Engineering at Indian Institute of Technology, Mumbai, India. Ranked fifth in the class.
1997-02	Teaching Assistant, Department of Computer Science, Rutgers, The State University of New Jersey.
1998	M.S. in Computer Science, Rutgers, The State University of New Jersey.
2001	Best Student Paper Award at ESA 2001 for the paper- Algorithms for Efficient Filtering in Content-based Multicast.
2002	Ph.D. in Computer Science, Rutgers, The State University of New Jersey.