A dynamic dictionary for priced information with application*

Anil Maheshwari Michiel Smid

February 2, 2004

School of Computer Science
Carleton University
1125 Colonel By Drive
Ottawa Ontario
CANADA K1S 5B6

Email: anil@scs.carleton.ca
Web: http://www.scs.carleton.ca/~maheshwa
Email: michiel@scs.carleton.ca
Web: http://www.scs.carleton.ca/~michiel

Abstract

In this paper we design a dynamic dictionary for the priced information model initiated by Charikar, Fagin, Guruswami, Kleinberg, Raghavan and Sahai [2, 3]. Assume that a set S consisting of n elements is given such that each element has an associated price, a positive real number. The cost of performing an operation on elements of S is a function of their prices. The cost of an algorithm is the sum of the costs of all operations it performs. The objective is to design algorithms which incur low cost. In this model, we propose a dynamic dictionary, supporting search, insert and delete, for keys drawn from a linearly ordered set. As an application we show that the dictionary can be used in computing the trapezoidal map of a set of line segments, a fundamental problem in computational geometry.

Keywords: Priced information, computational geometry, dictionary, trapezoidal map, competitive ratio.

1 Priced Information Model

Consider an algorithmic problem in which each input element has an associated price, which is a positive real number. When an algorithm performs an operation, it is charged a cost which is a function of the prices of the elements involved in this operation. For example, the cost of a comparison operation between two elements could be the sum of their prices. The cost of an algorithm is the total sum of the costs of all operations it performs. In this paper we wish to design algorithms that incur low cost with respect to the "cheapest proof". A proof is a certificate that the output produced by the algorithm is correct. For example, if the problem is to search for a key in a sorted list, then the proof consists of either an element of the list that matches the query, or a pair of consecutive elements such that the key of the query is between their keys. The cost of the proof is proportional to either the price of the element that matches the query or the sum of the prices of the two neighboring elements to the query. The *competitive ratio* of an algorithm is defined to be the maximum ratio between the cost of the algorithm and the cost of the cheapest proof over all possible inputs. In this model, the solution to a problem involves (i) describing the

*Work supported by NSERC. Preliminary version of this paper appeared in the 14th Annual International Symposium on Algorithms and Computation, Kyoto, Japan, December 2003.

cost of a cheapest proof, (ii) designing a competitive algorithm and (iii) analyzing the cost of this algorithm.

In this paper we propose a dynamic dictionary and use it to design a competitive algorithm for computing the trapezoidal map of a set of line segments; a fundamental problem in computational geometry. This study is inspired by the work of Charikar et al. [2, 3] on *query strategies for priced information*. Their motivation comes from the broad area of electronic commerce, where priced information sources in several domains (e.g., software, legal information, propriety information, etc.) charge for their usage. The unit-cost comparison tree model has been traditionally used for evaluating algorithms. The work of [2, 3, 5, 6] generalizes this model to accommodate variable costs. Geometric algorithms are usually designed and proven in the conventional "Real Random Access Machine" model of computation. Key features of this model include indirect addressing of any word in unlimited memory in unit time, words stored are infinite precision real numbers, and the basic operations (e.g., add, multiply, k-th root) are performed in constant time. To correctly implement a geometric algorithm, exact computation is extremely important. Unfortunately, exact computation is expensive and simple geometric tests, which take constant time in the Real RAM model, may require several operations. One way to model this is to associate prices to elements, and an operation involving an element needs to pay a cost which is a function of its price. Then an efficient algorithm aims to minimize the total cost of all the operations it performs.

1.1 Previous Work

We outline some of the fundamental problems such as searching, maximum finding, and sorting studied under the priced information model.

Theorem 1 ([3]) *For any cost function a query element can be searched in a sorted array of n elements within a competitive ratio of $\log_2 n + O(\sqrt{\log n} \log\log n)$.*

Theorem 2 ([3]) *The maximum of n elements can be found within a competitive ratio of $2n - 3$ for any set of costs for the comparisons between pair of elements.*

What happens if the cost of the comparison operation is just the sum of the prices of the corresponding elements? In this case the problem of computing the maximum is much easier and can be solved by the following natural algorithm. First sort the elements with respect to their price. Then incrementally compute the maximum by examining the elements one by one, starting at the least cost element, and comparing the next element with the maximum element seen so far. The total cost of this algorithm is bounded by $2\sum_{i=1}^{n} c_i$, where c_i is the price of the i-th element. Hence the competitive ratio of this algorithm is at most 2.

Corollary 1 *[5] The maximum of n elements can be found within a constant competitive ratio where the cost of a comparison between two elements is the sum of their prices.*

It turns out that the general problem of sorting a set of items with arbitrary cost functions is highly non-trivial and has the flavor of the famous "Matching Nuts and Bolts" problem [7]. Given two lists of n numbers each such that one list is a permutation of the other, how should we sort the lists by comparisons only between numbers in different lists? In our setting comparisons within the list will be very expensive compared to comparisons across the list.

If we define the cost of a comparison operation to be the sum of the prices of the elements involved, then this problem can be solved quite easily. First sort the elements with respect to their prices. Now incrementally sort the elements starting with the element with the least price. For

example we can build a balanced binary search tree to maintain the sorted order. It is easy to see that the cost of inserting an element is bounded by $2\log_2 n$ times its cost, since all the elements in the tree have lower price when this element is inserted. Therefore this algorithm is $O(\log n)$ competitive.

Claim 1 *A set of n elements can be sorted within a competitive ratio of $O(\log n)$ in the priced information model, where the cost of a comparison between two elements is the sum of their prices.*

1.2 New Results

In this paper we propose $O(\log n)$ competitive algorithms in the priced information model for the following problems; the cost of an operation is proportional to the sum of the prices of the elements involved in that operation.

1. A dynamic dictionary supporting searching, insertion and deletion of a key value in a linearly ordered set consisting of n elements (Section 2).

2. The trapezoidal map of a set of n line segments (Section 3).

The first result can be viewed as a generalization of Theorem 1. Here we discuss dynamic dictionaries, whereas Theorem 1 is for static search queries. To the best of our knowledge the second result is the first instance where a problem from computational geometry has been studied under the priced information model.

2 Dynamic Dictionary

In this section we describe a dynamic search data structure in the priced information model. The elements are drawn from a total order. We allow a sequence of operations comprising of searching a query value, insertion of an element and deletion of an element. Each element has an associated price, and the cost of accessing an element is proportional to its price. Without loss of generality we will refer to the key value associated with an element x by x itself. Assume that the least possible price is 1. First we introduce an abstract data type and show how the various operations are performed and then outline how they are realized using 2-3 search trees.

2.1 Hierarchical Structure

Assume that currently the data structure consists of a set S of n elements. The elements are partitioned into cost groups, and elements within cost group i have prices in the range $(2^{i-1}, 2^i]$ for $i \geq 0$. Let $g(x)$ denote the cost group of the element $x \in S$. The elements are placed in a hierarchical structure $\mathcal{H}$ represented as a tree. The top level of $\mathcal{H}$ represents the whole set S. The hierarchy is described by the following recursive procedure, which is invoked by the call Hierarchy(S,0). In a nutshell the main idea is to partition S recursively using the key values of groups. First partition S using the elements of the "cheapest" group (namely the elements of the cost group 0) to obtain subsets $Z_0, \cdots, Z_k$. These subsets are recursively partitioned by the elements of "expensive" groups (i.e., groups consisting of elements with geometrically increasing prices). For an illustration see Figure 1. To simplify notation, a leaf node storing the value y_i will be referred to as the node y_i.

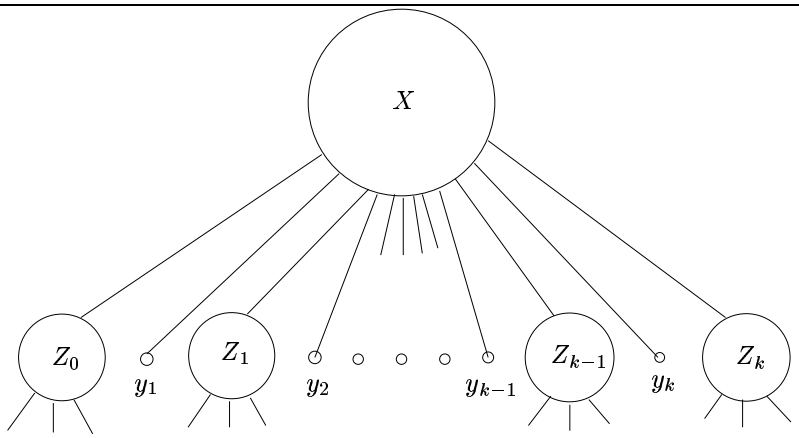

Figure 1: Illustration of Hierarchy (X, i).

Procedure Hierarchy (X,i)
Input: A non-empty set X, such that $X \subseteq S$ and all elements of X are in the cost group i or bigger (i.e. each element of X has price $> 2^{i-1}$.)
Output: The hierarchical structure $\mathcal{H}$, for X, represented as a tree.

1. Compute the set $Y = \{x \in X : x \text{ in cost group } i\}$.
2. If $Y = \emptyset$ then Hierarchy $(X,i+1)$.
3. If $Y \neq \emptyset$ then
   (a) Let $k = |Y|$ and $Y = \{y_1, y_2, \cdots, y_k\}$, such that $y_1 < y_2 < \cdots < y_k$.
   (b) Compute the sets $Z_j = \{z \in X \setminus Y : y_j < z < y_{j+1}\}, j = 1, 2, \cdots, k-1$.
   Compute $Z_0 = \{z \in X \setminus Y : z < y_1\}$ and $Z_k = \{z \in X \setminus Y : z > y_k\}$.
   (c) For each $j = 0, 1, \cdots, k$, if $Z_j \neq \emptyset$ then Hierarchy$(Z_j, i+1)$.
   (d) Create a node representing X. Give this node children $r_0, y_1, r_1, \cdots, r_{k-1}, y_k, r_k$, where r_j $(0 \leq j \leq k)$ is the root of the tree representing Z_j.

The hierarchical structure $\mathcal{H}$ output by Hierarchy$(S, 0)$ has the following properties.

1. All elements of S appear as leaves in $\mathcal{H}$.
2. A postorder traversal of the leaves of $\mathcal{H}$ results in the sorted order of the elements of S.
3. A node at level i in $\mathcal{H}$ (the root is at level 0) represents only elements of cost groups bigger than i.

2.2 Search

Next we describe the search procedure. The objective of the search for a query value q is to locate an element $x \in S$ such that $q = x$. In case there does not exists such an element in the set S, then report two elements $x, y \in S$, such that $x < q < y$, and x and y are the left and the right neighbors

of q in the sorted order among the elements of $S \cup \{q\}$, respectively. The main idea is to sieve through the hierarchy, starting at the top level, and descending down by the aid of the y_j-values. The search terminates when q is found or when a leaf node is reached. The search procedure is described next.

Procedure Search$(q, \mathcal{H})$
Input: $\mathcal{H}$=Hierarchy$(S, 0)$ and a query value q.
Output: An element $x \in S$ such that $q = x$, if such x exists. Otherwise, the left and the right neighbors of q in S.

1. left-neighbor := nil and right-neighbor := nil.
2. $X := S$ and Found := False.
3. While $X \neq \emptyset$ and Found = False do
   Let $Z_0, y_1, Z_1, y_2, Z_2, \cdots, Z_{k-1}, y_k, Z_k$ be the children of the node representing X in the hierarchy $\mathcal{H}$, where $y_1 < y_2 < \cdots < y_k$, and all elements in the set Z_j, $j \in \{1, \cdots, k-1\}$, have values in the range (y_j, y_{j+1}), elements in Z_0 have values smaller than y_1 and elements in Z_k have values larger than y_k (see Figure 1).
   (a) If $q = y_j$, for some $j \in \{1 \cdots k\}$ then report y_j and Found := True.
   (b) If $q < y_1$ then right-neighbor := y_1 and $X := Z_0$.
   (c) If $q > y_k$ then left-neighbor := y_k and $X := Z_k$.
   (d) If $y_j < q < y_{j+1}$ then left-neighbor := y_j, right-neighbor := y_{j+1} and $X := Z_j$.

Claim 2 *(a) Let $q \in S$. Then the algorithm* Search$(q, \mathcal{H})$ *does not visit elements of any cost group bigger than $g(q)$, where $g(q)$ denotes the cost group of q.*
(b) Let $q \notin S$ and let x and y be the neighbors of q in $S \cup \{q\}$. Then the algorithm Search$(q, \mathcal{H})$ *does not visit elements of any cost group bigger than* $\max\{g(x), g(y)\}$.

Proof: First consider the case where $q \in S$. Let node X be the parent of the leaf node representing q in the hierarchy $\mathcal{H}$. Observe that the search procedure visits X while searching for q. All the elements visited by the search procedure while it descends down $\mathcal{H}$, starting from the root till it reaches X, belong to cost groups less than $g(q)$. When the search procedure is exploring the node X, Step 3(a) is executed since $q = y_j$ for some $j \in \{1, \cdots, k\}$. This results in the termination of the "while loop" and the search procedure. Therefore no element of any cost group bigger than $g(q)$ is ever visited during the search.

Now consider the case where $q \notin S$. Let q have two neighbors $x, y \in S$. The case where q has only one neighbor can be handled similarly. Without loss of generality assume that $x < y$. Assume that $g(x) \geq g(y)$; the other case follows along the same argument. The search starts at the root node of $\mathcal{H}$ representing the whole set S and then sieves through $\mathcal{H}$ using y_j values of groups starting with cost group 0. In each step we consider the current set X which represents elements of cost groups at least, say, i. We locate q among its children $Z_0, y_1, Z_1, y_2, Z_2 \cdots, Z_{k-1}, y_k, Z_k$, and proceed with one of the sets Z_j. Note that $Z_j \subseteq X$ and Z_j represents elements of cost groups at least $i + 1$. Consider the scenario when the search reaches the elements of cost group $g(y)$. Since $q < y$ and there is no element of S between q and y, the search procedure assigns y as the right neighbor of q (Step 3b or 3d). After that in each iteration of the while loop Step 3c will be executed till the set Z_k becomes empty. At that point the left neighbor of q will be x, since there are no

elements of S between x and y, and $x < y$. Hence the search procedure only visits elements up to the cost group $g(x)$. □

2.3 Insert

Next we discuss the procedure for inserting an element q belonging to the cost group $g(q)$ in the hierarchy $\mathcal{H}$=Hierarchy$(S, 0)$. We first locate the left and the right neighbors, say x and y, respectively, of q in S using the algorithm Search$(q, \mathcal{H})$. Assume that $g(x) \geq g(y)$ as the other case is analogous. Depending upon the relative order of $g(q)$, $g(x)$ and $g(y)$, we have three cases.

Case 1: $g(x) \geq g(y) \geq g(q)$.

Starting at the leaf node containing x follow the path upwards in the hierarchy $\mathcal{H}$ and stop at the last node, say X, that represents only elements of cost groups at least $g(q)$. Let Z_j be the child of X that contains x as one of its descendants. See Figure 2 for an illustration. Let y_j and y_{j+1} be the left and the right neighbors (siblings) of Z_j among the children of X, following the notation in Figure 1. It is possible that none or only one of these neighbors exist; those cases can be handled using a similar approach. By our assumptions either $g(y) = g(q)$ or $g(y) > g(q)$.

Case 1a: $g(y) = g(q)$ (see Figure 2a)

In this case $y_{j+1} = y$. The insertion of q is performed by forming q as a leaf node and inserting it as a child of node X between the nodes Z_j and y_{j+1}.

Case 1b: $g(y) > g(q)$ (see Figure 2b)

In this case y is a descendent of Z_j, since $y_j < x < q < y < y_{j+1}$. The insertion of node q is achieved by performing the following two operations:
(a) Remove all elements in the set Z_j that have a larger key value than q. Form a new set Z'_j consisting of the removed elements.
(b) Insert two new nodes between Z_j and y_{j+1} as children of X. The first one is the leaf node q, and the second one is Z'_j representing the hierarchy of elements in the set Z'_j.

Case 2: $g(q) \geq g(x) \geq g(y)$.

Let node X be the parent node of the leaf node containing x in the hierarchy $\mathcal{H}$. Refer to node x as y_j and consider the node Z_j following the terminology used in Figure 1. Since the right neighbor of x is y before the insertion of q, and $g(x) \geq g(y)$, this implies that $Z_j = \emptyset$. The insertion of q is achieved by inserting q in the set Z_j.

Case 3: $g(x) \geq g(q) \geq g(y)$.

Starting at the leaf node x traverse the path upwards in the hierarchy and stop at the last node, say X, which represents only elements of cost groups up to and including $g(q)$. Let Z_j be the child of X that contains x as one of its descendants. If $g(q) = g(y)$ then insert q as a child of X between the nodes Z_j and $y(= y_{j+1})$. If $g(q) > g(y)$ then insert q as the rightmost child at the node X.

Claim 3 *Let q be the element to be inserted in the hierarchy $\mathcal{H}$=*Hierarchy$(S, 0)$*. Let its left and right neighbors in S be x and y, respectively. The insertion of q results in a new hierarchy $\mathcal{H}'$ which is equal to* Hierarchy$(S \cup \{q\}, 0)$*. The algorithm only visits elements up to the cost group* $\max\{g(x), g(y), g(q)\}$.

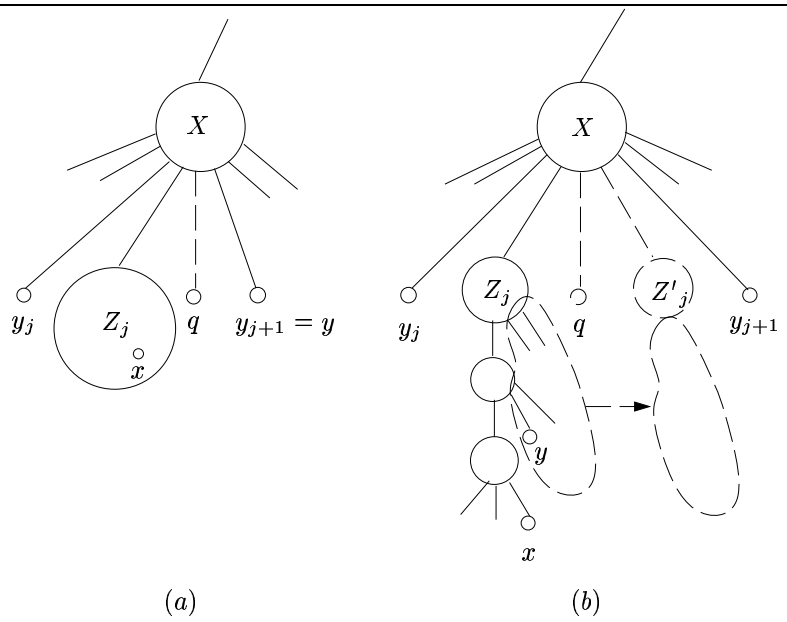

Figure 2: Illustration of Case 1 of the insertion of q in the hierarchy $\mathcal{H}$. (a) Case 1a: $g(x) \geq g(y) = g(q)$; q is inserted as a leaf node of X between Z_j and y_{j+1}. (b) Case 1b: $g(x) \geq g(y) > g(q)$. After inserting q, X obtains two new children q and Z'_j, between Z_j and y_{j+1}, and part of the hierarchy associated at node Z_j becomes the new hierarchy for Z'_j.

Proof: In each of the above cases q is inserted as a child of node X where the elements represented by X belong only to cost groups at least $g(q)$. Observe that in the postorder traversal of the leaves of $\mathcal{H}'$, x and y are the neighbors of q. This can be argued as follows.

In Case 1a the largest value among the descendants of Z_j is x. In Case 1b, the largest value left among the descendants of Z_j is x and the smallest value among the descendants of Z'_j is y.

In Case 2, before the insertion of q, elements x and y are consecutive elements of set S in the sorted order. If $g(x) = g(y)$ then both x and y are children of node X. If $g(x) > g(y)$ then x is the rightmost child of X. Observe that q is inserted in an empty node Z_j following the node x in X and hence in the postorder traversal of $\mathcal{H}'$, the leaf nodes x, q and y appear in the desired order.

In Case 3, observe that the largest value among the descendants of the node Z_j is x. □

2.4 Delete

In this section we outline the procedure for deleting an element $q \in S$ from the hierarchy $\mathcal{H}=$ Hierarchy$(S, 0)$ storing the elements of the set S. First we locate the neighbors x and y of q using the search procedure Search$(q, \mathcal{H})$. This requires modifying the Search procedure as follows. After finding the element y_j that equals q among the children of the node X, the search continues for the left neighbor and the right neighbor of q. In the following we assume that $g(x) \geq g(y)$, the other case is analogous. As in the case of insertion, we have three cases depending upon the order of the values $g(x)$, $g(y)$ and $g(q)$.

Case 1: $g(x) \geq g(y) \geq g(q)$.

Refer to Figure 3. Let node X be the parent of the leaf node containing $q = y_j$. Following the notation of Figure 1 let $Z_0, y_1, Z_1, \cdots, y_{j-1}, Z_{j-1}, q = y_j, Z_j, y_{j+1}, \cdots, Z_{k-1}, y_k, Z_k$ be the

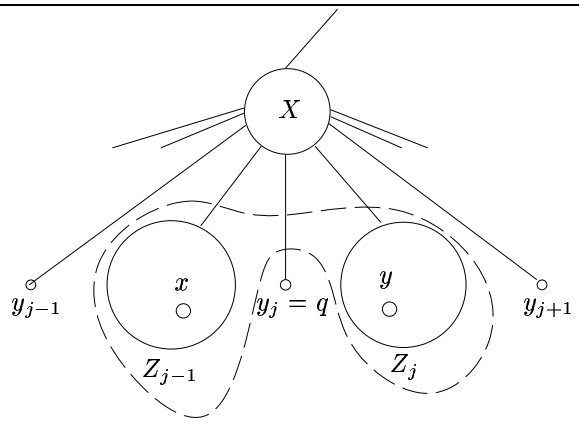

Figure 3: Illustration of the deletion of q in the hierarchy $\mathcal{H}$. Then neighbors of q, namely x and y, are in bigger cost groups compared to q, and $g(x) \geq g(y) \geq g(q)$. The deletion of q requires merging the hierarchies Z_{j-1} and Z_j.

children of the node representing X, where $y_1 < y_2 < \cdots < y_k$ and all elements in the set Z_i have values in the range (y_i, y_{i+1}) for $i \in \{1, k-1\}$, Z_0 has values smaller than y_1 and Z_k has values larger than y_k. Deletion of q can be achieved by removing the leaf node q and forming a new hierarchy by taking the union of the two hierarchies Z_{j-1} and Z_j.

Case 2: $g(q) \geq g(x) \geq g(y)$.
Let node X be the parent node of the leaf node containing x in the hierarchy $\mathcal{H}$. Refer to the node x as y_{j-1} following the notation in Figure 1. If $g(q) = g(x)$ then $y_j = q$ and deletion of q is achieved by deleting the leaf node y_j. Now consider the case where $g(q) > g(x)$. Note that q is stored in the hierarchy associated with the node Z_{j-1}, and moreover this is the only element stored in this hierarchy, since $g(x) \geq g(y)$. Deletion of q is achieved by setting $Z_{j-1} := \emptyset$.

Case 3: $g(x) \geq g(q) \geq g(y)$.
Remove the leaf node q.

Claim 4 *Let $q \in S$ be the element to be deleted in the hierarchy $\mathcal{H}$=*Hierarchy$(S, 0)$. *Let its neighbors in S be x and y, where $x < q < y$. The deletion of q results in a new hierarchy $\mathcal{H}'$=*Hierarchy$(S \setminus \{q\}, 0)$. *The algorithm only visits elements up to the cost group* $\max\{g(x), g(y), g(q)\}$.

Proof: Similarly to the proof for insertion, we will show that elements x and y are consecutive elements in the postorder traversal of the leaves of $\mathcal{H}'$. Observe that in Case 1, before the deletion of q from the node X, x (respectively, y) is the largest (respectively, smallest) element among the set of elements represented by the node Z_{j-1} (respectively, Z_j). The deletion results in merging these two sets and results in the elements x and y to be consecutive in the postorder traversal of the leaves of hierarchy $\mathcal{H}'$. In Case 2 after deletion of q, the node Z_{j-1} becomes empty. If $g(x) > g(y)$, then in $\mathcal{H}'$, x is the rightmost child of X, and x and y are consecutive elements in the postorder traversal of the leaves of $\mathcal{H}'$. If $g(x) = g(y)$ then x and y are adjacent children of node X in $\mathcal{H}'$. Consider Case 3. Let the parent of q be X in $\mathcal{H}$, and let Z_{j-1} be the child node of X such that x is one of the descendants of Z_{j-1}. Observe that the largest element among the descendants of

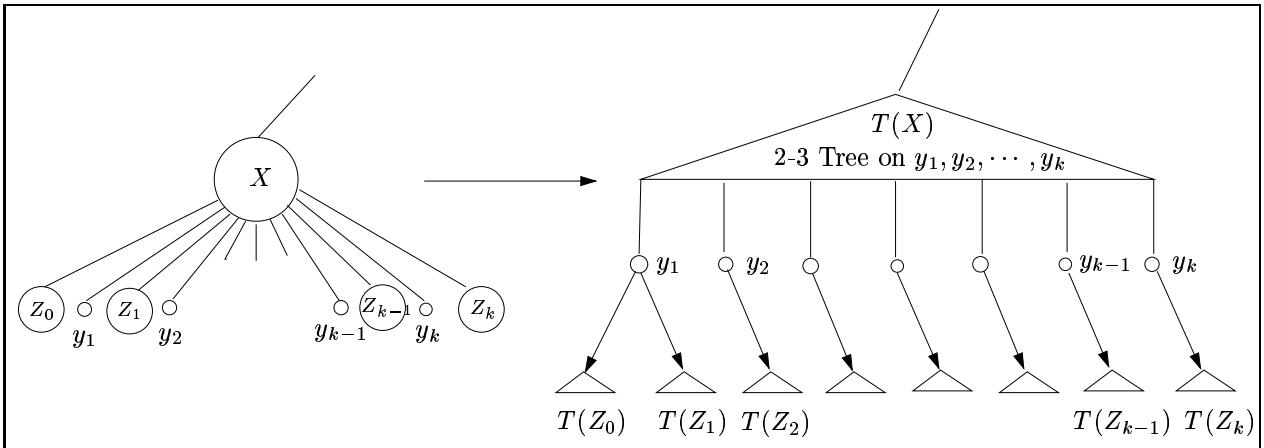

Figure 4: Figure on the left illustrates a node X and its children in the hierarchy $\mathcal{H}$. Figure on the right illustrates its equivalent 2-3 tree $T(X)$. Elements $y_1, \cdots, y_k$ of X form a 2-3 tree and are stored at the leaves. Each leaf node y_i has a pointer to the corresponding tree node $T(Z_i)$ which is a recursively defined 2-3 tree. Leaf node y_1 has an additional pointer to $T(Z_0)$.

Z_{j-1} is x. Hence after deleting the node q, x and y will be adjacent in the postorder traversal of the leaves of $\mathcal{H}'$. □

2.5 Implementation

We have described an abstract data type, called hierarchies, and the associated operations search, insert and delete. In this section we illustrate how we can realize this using *2-3 search trees*. Let us briefly recall the 2-3 tree data structure. A 2-3 tree is a tree in which each node, that is not a leaf, has 2 or 3 children, and every path from the root to a leaf is of the same length. A 2-3 tree can be used to store elements from a totally ordered set. This can be done by assigning the elements to the leaves of the tree in the left to right order. Each internal node stores a set of intervals describing the range of key values in the left, middle and right subtrees. The following theorem summarizes the relevant results on 2-3 trees.

Theorem 3 *([1]) Given a 2-3 tree on n elements, drawn from a totally ordered universe, each of the following operations can be performed in $O(\log n)$ time: (a) Searching for a key value. (b) Insertion of an element. (c) Deletion of an element. (d) Merging two 2-3 trees where all the key values in one tree are smaller than all the key values in the other tree. (e) Splitting a 2-3 tree into two 2-3 trees based on a key value, say q, where one tree will consist of all elements whose key values are at most q and the other tree will consist of all elements with key values larger than q.*

We represent the hierarchy $\mathcal{H}$ using 2-3 trees as follows. Recall that $\mathcal{H}$ was recursively defined, where a node X represents elements of cost groups at least i (see Figure 1 and the procedure Hierarchy(X, i)). Those children of node X which are leaves, namely $y_1, \cdots, y_k$, are represented in a 2-3 tree $T(X)$. See Figure 4 for an illustration. In $T(X)$ the elements are stored at the leaves. Each child y_i of X is stored as a leaf node in $T(X)$, since a 2-3 tree stores elements only in its leaves. The leaf node corresponding to y_i, $1 \leq i \leq k$, in $T(X)$ stores a pointer to $T(Z_i)$, which is the recursively defined 2-3 tree for Z_i. The leaf node corresponding to y_1 stores an additional pointer to $T(Z_0)$. Next we illustrate how search, insert and delete can be performed.

2.5.1 Searching

For searching an element q, we follow the procedure Search$(q, \mathcal{H})$ corresponding to the hierarchy $\mathcal{H}$=Hierarchy$(S, 0)$ representing the set S. Refer to Figure 1 for the notation. The leaf nodes $y_1, \cdots, y_k$ of S are represented in a 2-3 tree $T(S)$. We locate q in $T(S)$ and as a result we either find a leaf node $y_j = q$ and the search terminates or find two leaf nodes y_j and y_{j+1} such that $y_j < q < y_{j+1}$. In that case we perform the search in the hierarchy for Z_j, i.e., in the corresponding tree $T(Z_j)$, recursively.

Lemma 1 *An element can be searched in the hierarchical data structure storing the elements of the set S within a competitive ratio of $O(\log n)$, where $n = |S|$.*

Proof: The cheapest proof for membership of $q \in S$ is an element y_j that equals q, and the cheapest proof of non-membership is a pair of queries to two adjacent elements y_j and y_{j+1} such that $y_j < q < y_{j+1}$. Hence the cost of the cheapest proof, in the priced information model, is either the price of y_j or the sum of the prices of y_j and y_{j+1}.

Recall from the procedure Hierarchy that the leaf nodes associated to node X belong to a fixed cost group, say i, i.e., the prices of these nodes are in the range $(2^{i-1}, 2^i]$. Let the search procedure examine elements in groups $0, 1, 2, \cdots, l$. The total number of elements in these groups is at most n, and the total number of comparisons made within a group with respect to the query q is at most $2 \log n + C$, for some constant C. Hence the cost of searching using the 2-3 tree data structure will be at most

$$(2 \log n + C) \sum_{i=0}^{l} 2^i \leq (2 \log n + C) 2^{l+1}. \tag{1}$$

As noted in Claim 2, the search does not examine any elements of cost groups larger than that of the neighbors of q in S. Therefore the cost of the cheapest proof is at least 2^{l-1}. Hence the competitive ratio, i.e., the ratio of the cost of the algorithm to the cost of the cheapest proof, is at most

$$\frac{(2 \log n + C) 2^{l+1}}{2^{l-1}} = 8 \log n + 4C. \Box \tag{2}$$

2.5.2 Insert

Insertion of an element q in the hierarchy $\mathcal{H}$=Hierarchy$(S, 0)$ requires searching for the neighbors of q, possibly splitting part of the hierarchy from a leaf node up to an ancestor node (Case 1, Section 2.3), and inserting q as a leaf node. We have already discussed how searching can be realized in Section 2.5.1.

The splitting can be realized as follows. Assume that we wish to split $\mathcal{H}$ with respect to the key value q as in Case 1b of the insert procedure in Section 2.3. Recall that each (non-leaf) node X of $\mathcal{H}$ is realized by a 2-3 tree, $T(X)$, and the leaves of $T(X)$ point to recursively defined 2-3 trees of the sub-hierarchies associated with X. Splitting $\mathcal{H}$ with respect to the key value q amounts to splitting each of the associated 2-3 trees on the path from the leaf node containing x up to the node in $\mathcal{H}$ representing cost groups at least $g(q)$. Each of these 2-3 trees can be split with respect to the key value q resulting in two 2-3 trees. One of these trees represents key values smaller than q and the other one represents key values larger than q. The number of operations performed for each split is at most logarithmic in the size of the tree.

Lastly, the actual insertion of q in $\mathcal{H}$ can be accomplished by inserting q in a 2-3 tree corresponding to the cost group $g(q)$, as dictated by one of the cases in Section 2.3.

The analysis of an insertion is similar to that of searching. The cheapest proof involves the price of q and the sum of the prices of the neighbors of q in $S \cup \{q\}$. The total cost of performing the insertion is the sum of the costs of searching the neighbors of q and then performing the split and the actual insertion. The searching is performed among the 2-3 trees representing groups $0, \cdots, \max\{g(x), g(y)\}$, where x and y are the neighbors of q. The split is performed on the 2-3 trees representing groups $\max\{g(x), g(y)\}$ down to $g(q)$. The insertion of q is performed in a 2-3 tree representing the cost group $g(q)$. Hence each of the standard 2-3 tree operations (search, insert, split) are performed on 2-3 trees representing cost groups $0, \cdots, \max\{g(y), g(x), g(q)\}$. Hence using Equations (1) and (2) we obtain that insertion of an element in the set S, $|S| = n$, can be done within a competitive ratio of $O(\log n)$.

2.5.3 Delete

Deletion of an element q from the hierarchy $\mathcal{H}$=Hierarchy$(S, 0)$ requires searching for q and its neighbors in S, merging of two sub-hierarchies, and the deletion of the leaf node q. The searching of q and its neighbors was discussed in Section 2.5.1. Merging of the two sub-hierarchies as required in Case 1 of Section 2.4 can be achieved as follows.

Following the notation of Section 2.4 (see Figure 3), recall that x and y are the neighbors of q in S, $x < y$. Assume without loss of generality that $g(x) \geq g(y)$. The parent of the leaf node in $\mathcal{H}$ containing q is X, and x (respectively, y) is contained in the child node Z_{j-1} (respectively, Z_j) of X. For each internal node X' (respectively, Y') in $\mathcal{H}$ on the path from the leaf containing x (respectively, y) up to Z_{j-1} (respectively, Z_j), there is an associated 2-3 tree $T(X')$ (respectively, $T(Y')$). Moreover the key values stored in $T(X')$ are smaller than in $T(Y')$ and each 2-3 tree represents key values of a particular cost group. Merging of the sub-hierarchies Z_{j-1} and Z_j is achieved by merging the corresponding 2-3 trees which belong to the same cost group. The total number of operations required to merge two 2-3 trees is given by Theorem 3.

The actual deletion of q in $\mathcal{H}$ corresponds to deleting q from a 2-3 tree representing elements of the cost group $g(q)$, as given by one of the cases in Section 2.4.

Similar to insertion, deletion only requires performing standard 2-3 tree operations on elements belonging to the cost groups $0, \cdots, \max\{g(q), g(x), g(y)\}$. Hence the analysis using Equations (1) and (2) can be applied and will result in an $O(\log n)$ competitive algorithm for the deletion of an element of the set S. The following theorem summarizes the results of this section.

Theorem 4 *A dynamic dictionary representing a set consisting of n priced elements drawn from a total order can be maintained. It supports searching, insertion and deletion of a key value and each of these operations can be achieved within a competitive ratio of $O(\log n)$ in the priced information model. The cost of an operation is the sum of the prices of the elements involved in that operation.*

3 Trapezoidal Maps

In this section we use the dynamic search structure of the previous section to compute the trapezoidal map of a set of line segments in the priced information model. Let S be a set of n non-crossing line segments in the plane enclosed in a bounding box R. The *trapezoidal map* $\mathcal{T}(S)$ of S is obtained by drawing two vertical extensions from every endpoint p of each segment s in S. One of the extensions goes upwards and the other one downwards till it reaches either a segment of S

or the boundary of R. The trapezoidal map is the subdivision induced by the segments in S, the bounding box R and the vertical extensions. Observe that each face of $\mathcal{T}(S)$ has one or two vertical sides and exactly two non-vertical sides, and hence each face is either a trapezoid or a triangle. Trapezoidal maps are fundamental geometric structures and are required in a preprocessing step for solving several computational geometry problems (see [4]).

The problem is to compute a trapezoidal map $\mathcal{T}(S)$ of a set S of n non-crossing line segments where each segment has an associated price. The cost of the cheapest proof for the trapezoidation is at least the total sum of the prices of all the boundary elements in all faces of $\mathcal{T}(S)$. The cost of a vertical extension l is the sum of the prices of the two segments in S to which l is incident.

Traditionally, in the uniform cost model, this problem is solved using the plane sweep paradigm as follows. Sort the endpoints of the segments with respect to increasing x-coordinates and insert them into an event queue. Sweep a vertical line from the left to the right maintaining the invariant that at each event point, the trapezoidal map of all the segments to the left has been computed. Maintain the y-sorted order of all segments intersecting the sweep line. At an event point either a segment is inserted in the sweep line data structure or it is deleted from it. When a segment is inserted/deleted the vertical extensions of its endpoint are computed by finding out its neighbors in the y-sorted order of the segments maintained on the sweep line. Moreover after inserting/deleting the sweep line data structure is suitably updated.

In the priced information model we make use of dynamic dictionaries to represent the y-sorted order of the segments intersecting the sweep line. The operations that are required on this data structure includes (a) searching for neighbors of a query value (b) inserting a new segment (i.e., insert a key value) (c) deleting an existing segment (i.e., delete a key value) . In addition to this we need an event list, which consists of endpoints of segments in the x-sorted order.

Theorem 5 *The trapezoidal map of a set of n non-intersecting priced line segments can be computed within a competitive ratio of $O(\log n)$ where the cost of an operation is the sum of the prices of the elements involved in that operation.*

Proof: The correctness follows along the same lines as the correctness of the algorithm for uniform costs. For the competitive analysis observe the following. The cost W of the cheapest proof is at least the total sum of the prices of all boundary elements in all faces of $\mathcal{T}(S)$. The cost of our algorithm can be analyzed as follows. The cost of computing a vertical extension is the sum of the prices of the two segments: the segment which has been inserted/deleted, and the segment where the vertical extension ends. From Theorem 4 we know that search, insert and delete, can be performed within a competitive ratio of $O(\log n)$. Hence the total cost of maintaining the sweep line data structure is $O(W \log n)$. The event queue consists of sorted sequence of endpoints of segments in S. The cost of sorting the endpoints is $O(W \log n)$ by Claim 1. □ The trapezoidal map $\mathcal{T}(S)$ of a set S consisting of n (possibly intersecting) segments can be computed in a similar manner. In addition to computing vertical extensions of the endpoints of segments, we need to compute vertical extensions at each intersection point. Now the set of events includes endpoints of segments as well as all intersection points. Observe that if two segments a and b intersect then they will be adjacent to each other on the sweep line at least once (e.g., just before the sweep line reaches the intersection point). Now event points are processed as follows:

- Left endpoint: Suppose segment l needs to be inserted, and let a and b be the neighboring segments of l on the sweep line data structure, where a is above l and b is below l. First we locate a and b and then insert l in the sweep line data structure. Since a and l (and similarly b and l) have become adjacent, we check whether they intersect to the right of the sweep line.

If so then their intersection point is inserted in the event queue. Draw upward (downward) vertical extension from the left endpoint of l to the segment a (respectively, b).

- Right endpoint: Suppose segment l needs to be deleted and let a and b be the neighboring segments as before. First locate l and its neighbors a and b. Draw vertical extensions from the right endpoint of l to a and b and delete l. Now a and b have become adjacent and it is possible that they intersect and their intersection point is to the right of the sweep line. If a and b have been adjacent at some point on the sweep line before l was even inserted then their intersection point is already present in the event queue. Therefore we search for the intersection point in the event queue, and if it is not present then we insert it.

- Intersection point: Suppose the sweep line reaches the intersection point v of segments l and l' and let the segment a (respectively, b) be just above (respectively, below) v. Furthermore assume that l and a were adjacent, and similarly l' and b were adjacent, on the sweep line just before it reaches v. First locate v and the segments a and b on the sweep line data structure and draw the vertical extensions from v to a and b. Furthermore now the segments a and l', and similarly b and l, have become adjacent. Insert their intersection point into the event queue, if required.

Theorem 6 *The trapezoidal map of a set of n priced line segments can be computed within a competitive ratio of $O(\log n)$.*

Proof: The cost of the cheapest proof is at least the total sum of the prices of all boundary elements of all faces in $\mathcal{T}(S)$. Let W denote the cost of this proof. The cost of the algorithm is the sum of the costs of the operations required in maintaining the event queue and maintaining the sweep line data structure. The cost of maintaining the sweep line data structure is $O(W \log n)$ which can be argued as follows. The operations of search, insert and delete are $O(\log n)$ competitive and they require adding vertical extensions whose total cost is $O(W)$. The cost of maintaining the event queue is $O(W \log n)$ which is argued as follows. We know the price, and hence the cost group, of each item q that we search in the event queue. We modify the search procedure in Section 2 to stop as soon as it reaches the cost group $g(q)$; thus we do not examine elements of higher cost group than $g(q)$. Therefore the cost of the search is of the order of $\log n$ times the price of q, and the total cost of searching for all elements in the event queue is $O(W \log n)$. Therefore, each of the insertion, deletion and search operations in the event queue is $O(\log n)$ competitive. □

References

[1] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[2] Charikar, M., Fagin, R., Guruswami, V., Kleinberg, J., Raghavan, P., and Sahai, A. Query strategies for priced information (extended abstract). In *Proc. ACM Symp. on Theory of Computation* (New York, 2000), ACM, pp. 582–591.

[3] Charikar, M., Fagin, R., Guruswami, V., Kleinberg, J., Raghavan, P., and Sahai, A. Query strategies for priced information. *Journal of Computer and System Sciences 64(4)* (2002), 785–819.

[4] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. *Computational Geometry – Algorithms and Applications.* Springer Verlag, Berlin, 1997.

[5] Gupta, A., and Kumar, A. Sorting and selection with structured costs. In *Proc. IEEE Symp. on Foundations of Comp. Sci.* (2001), pp. 582–591.

[6] Kannan, S., and Khanna, S. Selection with monotone comparison costs. In *Proc. ACM-SIAM Symp. on Discrete Algorithms* (2003), pp. 10–17.

[7] Komlos, J., Ma, Y., and Szemeredi, E. Matching nuts and bolts in $O(n \log n)$ time. *SIAM Journal on Discrete Mathematics 11* (1998), 347–372.