A Dynamic Dictionary for Priced Information with Application*

Anil Maheshwari and Michiel Smid

School of Computer Science, Carleton University, Ottawa, Canada.
{anil,michiel}@scs.carleton.ca

Abstract. In this paper we design a dynamic dictionary for the priced information model initiated by [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 insert, delete, and search 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.

1 Introduction

Priced Information Model: Assume that each input element has an associated price, a positive real number. When an algorithm performs an operation on an element, then it is charged a cost which is a function of the price of the element. 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 cost of a cheapest proof, (ii) designing a competitive algorithm and (iii) analyzing its cost.

In this paper we propose a dynamic dictionary and use this 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

* Work supported by NSERC.

T. Ibaraki, N. Katoh, and H. Ono (Eds.): ISAAC 2003, LNCS 2906, pp. 16–25, 2003.

motivation comes from the broad area of electronic commerce, where the 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 [2,3,4,5] 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, the exact computation is very 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.

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

Theorem 1 ([2]). *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 ([2]). *The maximum of n elements can be found within a competitive ratio of $2n - 3$ for any set of costs for the comparisons.*

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 following a natural algorithm. Sort the elements w.r.t. their cost. Incrementally compute the maximum by examining the elements one by one, starting at the least cost element. 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. 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 [6]. 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 modify the cost of the 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 the price. Now incrementally sort the elements starting with the element with the least price. For example we can build a 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.

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 insertion, deletion and searching of a key value in a linear ordered set consisting of n elements (Section 2).
2. The trapezoidal map of a set of n line segments (Section 3).

Result 1 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 Result 2 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 insertion of an element, deletion of an element and searching a query value. 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 group 0 values) 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.

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

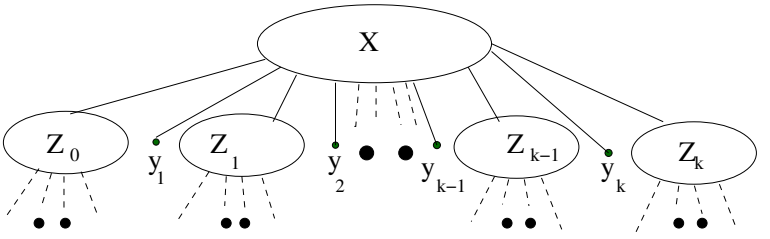

Fig. 1. Illustration of the hierarchy $\mathcal{H}$.

a) Let $Y = \{y_1, y_2, \cdots, y_k\}$, such that $y_1 < y_2 < \cdots < y_k$, for some $k \geq 1$.
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 j, $j = 0, 1, \cdots, k$, 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_i $(0 \leq i \leq k)$ is the root of the tree representing Z_i.

The hierarchical structure $\mathcal{H}$ has the following properties. All elements of S appear as leaves in $\mathcal{H}$. A postorder traversal of the leaves of $\mathcal{H}$ results in the sorted order of the elements of S, and a node at level i in $\mathcal{H}$ (root is at level 0) represents only elements of groups $\geq i$.

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 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-values. The search terminates when it reaches a leaf node. The search procedure is described next.

Procedure Search(q, S)
Input: Set S stored in a hierarchical structure 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_i, $i \in \{1, \cdots, k-1\}$, have values in the range (y_i, y_{i+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 $X := Z_j$.

4. If Found = False then report left-neighbor and right-neighbor.

Lemma 1. *Let x and y be the neighbors of q in $S \cup \{q\}$, and let the price of x be at least the price of y. Let $g(x)$ denote the group of x. Then* Search(q, S) *does not visit an element of any group having index bigger than $g(x)$.*

Proof Sketch: Consider the case where $q \notin S$. Let q have two neighbors $x, y \in S$. W.l.o.g. assume that $x < y$. Let x and y be in groups $g(x)$ and $g(y)$, resp. Observe that $g(y) \in \{0, \cdots, g(x)\}$. The search starts at S and then sieves through $\mathcal{H}$ using the y_i values of groups starting with group 0. In each step we consider the current set X which consists of elements of groups $\geq i$. 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 depending upon the outcome we either terminate the search or proceed with one of the sets Z_j's. Note that $Z_j \subseteq X$ and elements of Z_j are of groups $\geq i+1$. Consider the scenario when the search reaches the elements of 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_j 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 only visits elements up to the group $g(x)$. □

2.3 Insert

Next we discuss the procedure for inserting an element q belonging to the group $g(q)$ in the hierarchy $\mathcal{H}$. We first locate the left and the right neighbors, say x and y, respectively, of q in S using the Search(q, S) procedure. Without loss of generality we assume that $g(x) \geq g(y)$. Depending upon the relative order of $g(q)$, $g(x)$ and $g(y)$ we have different 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 groups $\geq g(q)$. Let Z_i be the child of X that contains x as one of its descendants. Let y_i and y_{i+1} be the left and the right neighbors of Z_i among the children of X. It is possible that none or only one of these neighbors exist; those cases can be handled using a similar approach. Observe that if $g(y) = g(q)$ then $y = y_{i+1}$, and if $g(y) > g(q)$ then y is a descendent of the node Z_i, since $y_i < x < q < y < y_{i+1}$. The insertion of node q is achieved by performing the following two operations: (a) Remove all elements in the set Z_i that have larger value than q. Form a new set Z_i' consisting of the removed elements. (b) Insert two new nodes between Z_i and y_{i+1} as children of X. The first one is the leaf node consisting of the element q, and the second one is Z_i' representing the hierarchy of elements in the set Z_i'.

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_i and consider the node Z_i 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 $Z_i = \emptyset$. Insertion of q is achieved by simply inserting q in the set Z_i.
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 groups up to and including $g(q)$. If $g(x) = g(q) = g(y)$ then insert q as a child between x and y at the node X. If $g(q) > g(y)$ then insert q as the rightmost child at the node X.

Lemma 2. *Let q be the element to be inserted in the hierarchy $\mathcal{H}$. Let its neighbors in S be x and y, where $x < q < y$, and let $g(x) \geq g(y)$. Insertion of q, depending upon one of the above three cases, results in a new hierarchy $\mathcal{H}'$. The algorithm only visits elements up to the cost group* $\max\{g(x), g(q)\}$.

2.4 Delete

In this section we outline the procedure for deleting an element q from the hierarchy $\mathcal{H}$ storing the elements of the set S. First locate the neighbors x and y of q using the search procedure $\mathsf{Search}(S, q)$. This requires modifying the Search procedure as follows. After finding an element y_i that equals q among the children of the node X, the search continues for the left neighbor $x = \max\{Z_{i-1}\}$ and the right neighbor $y = \min\{Z_i\}$ of q. In the following we assume that $g(x) \geq g(y)$, the other case $g(y) \geq g(x)$ can be handled similarly.
Case 1: $g(x) \geq g(y) \geq g(q)$. Let node X be the parent of the leaf node containing $q = y_{i+1}$. Following the notation of Figure 1 let $\{Z_0, y_1, Z_1, \cdots, y_i, Z_i, q = y_{i+1}, Z_{i+1}, y_{i+2}, \cdots, Z_{k-1}, y_k, Z_k\}$ be the 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 $i \in \{1, k-1\}$, and 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_i and Z_{i+1}.
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_i and consider the node Z_i following the terminology used in Figure 1. Note that q is stored in the hierarchy at Z_i, and moreover this is the only element stored in this hierarchy, since $g(x) \geq g(y)$. Deletion of q is achieved by setting $Z_i := \emptyset$.
Case 3: $g(x) \geq g(q) \geq g(y)$. Remove the leaf node q.

Lemma 3. *Let q be the element to be deleted in the hierarchy $\mathcal{H}$. Let its neighbors in S be x and y, where $x < q < y$, and let $g(x) \geq g(y)$. Deletion of q, depending upon one of the above three cases, results in a new hierarchy $\mathcal{H}'$. The algorithm only visits elements up to the cost group* $\max\{g(x), g(q)\}$.

2.5 Implementation

We have described an abstract data type, called hierarchies, and the associated operations insert, delete and search. In this section we illustrate how we can

realize this using *2-3 search trees*. A 2-3 tree is a tree in which each vertex, 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 the right order. Each internal node stores a set of intervals describing the range of key values in the left, middle and the 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) Insertion of an element. (b) Deletion of an element. (c) Searching for a key value. (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 $\leq q$ and the other tree will consist of all elements with key values $> q$.*

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 groups $\geq i$ (see Figure 1 and the procedure $\mathsf{Hierarchy}(X, i)$). Those children of node X which are leaves, namely $y_1, \cdots, y_k$, are represented in a 2-3 tree $T(X)$. 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.

Searching: For searching an element $q \in S$, we follow the procedure $\mathsf{Search}(q, S)$ on the tree $T(S)$ corresponding to the hierarchy $\mathcal{H}$ 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 need to perform the search in the hierarchy Z_j, i.e. in the corresponding tree $T(Z_j)$, recursively.

Lemma 4. *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 x_j that equals q, and the cheapest proof of non-membership is a pair of queries to two adjacent elements x_j and x_{j+1} such that $x_j < q < x_{j+1}$. Hence the cost of the cheapest proof is either the price of x_j or sum of the prices of x_j and x_{j+1}. Recall from the procedure $\mathsf{Hierarchy}$ that the leaf nodes associated to node X belong to the cost group 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 $(2 \log n + C) \sum_{i=0}^{l} 2^i \leq (2 \log n + C) 2^{l+1}$. As noted in Lemma 1,

the search does not examine any elements of the groups larger than that of the neighbors of q in S. Therefore the cost of the cheapest proof is $> 2^{l-1}$. Hence the competitive ratio, i.e. the ratio of the cost of the algorithm to the cost of the cheapest proof is at most $\frac{(2\log n+C)2^{l+1}}{2^{l-1}} = 8\log n + 4C$.□

Insert: Insertion of an element q in the hierarchy $\mathcal{H}$ requires searching for 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.

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 1 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 groups $\geq g(q)$. 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 represent 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. The actual insertion of q in $\mathcal{H}$ can be done by inserting q in a 2-3 tree corresponding to the group $g(q)$, as dictated by one of the cases in Section 2.3.

The analysis of 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 actual insertion. The searching is performed among the 2-3 trees representing groups $0, \cdots, g(x)$, where x is the neighbor of q with the highest price. The split is performed on the 2-3 trees representing groups $g(x)$ up to $g(q)$. Insertion of q is performed in a 2-3 tree representing the group $g(q)$. Hence each of the standard 2-3 tree operations (search, insert, split) are performed on 2-3 trees representing groups $0, \cdots, \max\{g(y), g(x), g(q)\}$, where x and y are neighbors of q in $S \cup \{q\}$. Similar to searching, the insertion of an element in the set S, $|S| = n$, can be done within a competitive ratio of $O(\log n)$.

Delete: Deletion of an element q from the hierarchy $\mathcal{H}$ requires searching for q and its neighbors in S, merging of two sub-hierarchies, and the deletion of the leaf node q. 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, recall that x and y are neighbors of q in S, $x < y$, and $g(x) \geq g(y)$. The parent of the leaf node containing q is X in $\mathcal{H}$, and x (resp. y) is contained in the child node Z_i (resp. Z_{i+1}) of X. For each internal node X' (resp., Y') in $\mathcal{H}$ on the path from the leaf containing x (resp., y) up to Z_i (resp. Z_{i+1}), there is an associated 2-3 tree $T(X')$ (resp., $T(Y')$). 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_i and Z_{i+1} 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.

Theorem 4. *A dynamic dictionary representing a set consisting of n priced elements drawn from a total order can be maintained. It supports insertion, deletion and searching 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

Let S be a set of n non-crossing line segments in the plane enclosed in a bounding box R. The *trapezoidal map* $\mathcal{T}(\mathcal{S})$ of S is obtained by drawing two vertical extensions from each endpoint of every 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 $\mathcal{T}(\mathcal{S})$ has one or two vertical sides and exactly two non-vertical sides, and hence each face is either a trapezoid or a triangle. The problem is to compute a trapezoidal map $\mathcal{T}(\mathcal{S})$ 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 $\mathcal{T}(\mathcal{S})$. 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 end points of the segments with respect to increasing x-coordinate and insert them into an event queue. Sweep a vertical line from the left to the right and 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 end-point 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 cost 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) inserting a new segment (i.e. insert a key value) (b) deleting an existing segment (i.e. delete a key value) (c) searching for neighbors of a query value. In addition to this we need an event list, which consists of end-points of segments in the x-sorted order.

Next we discuss the computation of the trapezoidal map $\mathcal{T}(\mathcal{S})$ of a set S consisting of n (possibly intersecting) segments. 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 and 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). 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 (resp. 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 (resp. b) is just above (resp. 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 5. *The trapezoidal map of a set of n priced line segments can be computed within a competitive ratio of $O(\log n)$.*

References

1. Aho, Hopcroft, and Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.
2. Charikar, Fagin, Guruswami, Kleinberg, Raghavan, and Sahai. Query strategies for priced information. *Journal of Computer and System Sciences 64* (2002).
3. 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.
4. Gupta, and Kumar. Sorting and selection with structured costs. In *Proc. IEEE Symp. on Foundations of Comp. Sci.* (2001).
5. Kannan, and Khanna. Selection with monotone comparison costs. In *Proc. ACM-SIAM Symp. on Discrete Algorithms* (2003).
6. Komlos, Ma, and Szemeredi. Matching nuts and bolts in O(n log n) time. *SIAM Journal on Discrete Mathematics 11* (1998).