Query Strategies for Priced Information*

Moses Charikar † Ronald Fagin ‡ Venkatesan Guruswami § Jon Kleinberg ¶
Prabhakar Raghavan ‖ Amit Sahai **

November, 2000

Abstract

We consider a class of problems in which an algorithm seeks to compute a function f over a set of n inputs, where each input has an associated *price*. The algorithm queries inputs sequentially, trying to learn the value of the function for the minimum cost. We apply the competitive analysis of algorithms to this framework, designing algorithms that incur large cost only when the cost of the cheapest "proof" for the value of f is also large. We provide algorithms that achieve the optimal competitive ratio for functions that include arbitrary Boolean AND/OR trees, and for the problem of searching in a sorted array. We also investigate a model for pricing in this framework and construct, for every AND/OR tree, a set of prices that satisfies a very strong type of equilibrium property.

*A preliminary version of this paper appeared in the *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, Portland, Oregon, May 2000.

†Computer Science Department, Stanford University, CA 94305. Email: `moses@cs.stanford.edu`. Research supported by the Pierre and Christine Lamond Fellowship, NSF Grant IIS-9811904 and NSF Award CCR-9357849, with matching funds from IBM, Mitsubishi, Schlumberger Foundation, Shell Foundation, and Xerox Corporation. Most of this work was done while the author was visiting IBM Almaden Research Center.

‡IBM Almaden Reseach Center, 650 Harry Road, San Jose, CA 95120. Email: `{fagin,pragh}@almaden.ibm.com`.

§Laboratory for Computer Science, MIT Room NE43-370, 200 Technology Square, Cambridge, MA 02139. Email: `venkat@theory.lcs.mit.edu`. Most of this work was done while the author was visiting the IBM Almaden Research Center.

¶Department of Computer Science, Cornell University, Ithaca NY 14853. Email: kleinber@cs.cornell.edu. Supported in part by a David and Lucile Packard Foundation Fellowship, an Alfred P. Sloan Research Fellowship, an ONR Young Investigator Award, and NSF Faculty Early Career Development Award CCR-9701399.

‖Verity, Inc. Work done while the author was at the IBM Almaden Research Center. Email: `praghava@verity.com`

** Department of Computer Science, Princeton University. Princeton, NJ 08544. Most of this work was done while the author was visiting the IBM Almaden Research Center. EMail: `sahai@cs.princeton.edu`.

1 Introduction

The potential of *priced information sources* [13, 14] that charge for usage is being discussed in a number of domains — software, research papers, legal information, proprietary corporate and financial information — and it forms a basic component of the larger area of electronic commerce [4, 6, 17, 18]. In a networked economy, we envision software agents that autonomously purchase information from various sources, and use the information to support decisions. How should one query data in the presence of a given price structure?

Previous theoretical analysis has posited settings in which there is a *target* piece of information, and the goal is to locate it as rapidly as possible; see for example the work of Etzioni et al. [5] and Koutsoupias et al. [10]. Here we take an alternate perspective, motivated by the following type of consideration. Suppose we have derived, through some pre-processing based on data mining or other statistical means, a *decision rule* that we wish to apply. To take a toy example, such a rule might look like

If Analyst A values Microsoft at $X
    or Analyst B values Netscape at $Y;
and if Analyst C values Oracle at $Z
    or Analyst D values IBM at $W;
then we should sell our shares of eBay.

The decision rule in this example depends on four available information sources, which we could label A, B, C, and D; each has a Boolean value. It is possible to evaluate the rule, under some circumstances, without querying all the information sources. If each of these pieces of information has an associated price, what is the best strategy for evaluating the decision rule?

Note the following features of this toy example. There is an underlying set of information sources, but our goal is not simply to gather *all* the information; rather it is to collect (as cheaply as possible) a subset of the information sufficient to compute a desired function f. Thus, a crucial component of our approach is the view that disparate information sources contain raw data to be *combined* to reach a decision, and it is the structure of this combination that determines the optimal strategy for querying the sources. Our setting may be further generalized to allow inputs that are entire databases, rather than bits (say, a demographic information database from a vendor such as Lexis-Nexis), and the goal is to distill valuable information from a combination of such databases; this generalization suggests an interesting direction for further work.

An Illustrative Example. In Figure 1 we depict the above toy example, with the decision rule represented by a tree-structured Boolean circuit, and with the prices $\langle 6, 3, 1, 4\rangle$ attached to the inputs. An algorithm is presented with this circuit and the vector of prices; the hidden information is the setting σ of the four Boolean variables. The algorithm must query the variables, one by one, until it learns the value of the circuit; with each variable it queries, it pays the associated cost. We could ask for an algorithm $\mathcal{A}$ that incurs the minimum worst-case cost over all settings of the variables; but this is too simplistic: many of the natural functions we wish to study (including all AND/OR trees) are *evasive* [3], so any algorithm can be made to pay for all the variables, and all algorithms perform equally poorly under this measure.

The competitive analysis of algorithms [2] fits naturally within our framework; we define the

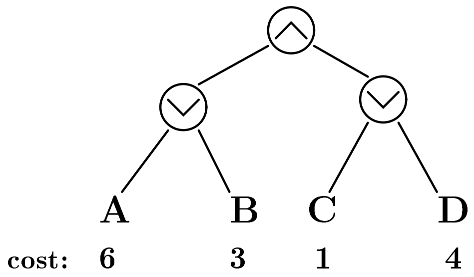

Figure 1: A Boolean function with priced inputs

performance of an algorithm $\mathcal{A}$ on a given setting σ of the variables to be the ratio of the cost incurred by $\mathcal{A}$ to the cost of the cheapest "proof" for the value of the function. The *competitive ratio* of $\mathcal{A}$ is then the maximum of this ratio over all settings σ of the variables.

In the example above, consider the algorithm $\mathcal{A}'$ that first queries C. If C is `true`, it then queries B and A (if necessary); if C is `false`, it then queries D, then B and A (if necessary). The performance of $\mathcal{A}'$ when the setting is $\langle \texttt{true}, \texttt{false}, \texttt{false}, \texttt{true} \rangle$ is $7/5$: $\mathcal{A}'$ queries all the variables and pays 14, while querying only A and D would prove the value of the function is `true`. Indeed, this is the competitive ratio of $\mathcal{A}'$, and $\mathcal{A}'$ achieves the optimal competitive ratio of any algorithm on this function, with this cost vector. Two aspects of $\mathcal{A}'$ are noteworthy: (i) it is *adaptive* – its behavior depends on the values of the inputs it has read, and (ii) it does not always read the inputs in increasing order of price.

A Framework. We now describe a general framework that captures the issues and example discussed above. We have a function f over a set $V = \{x_1, \ldots, x_n\}$ of n variables. Each variable x_i has a non-negative *cost* c_i; the vector $\mathbf{c} = \langle c_1, \ldots, c_n \rangle$ will be called the *cost vector*. A *setting* σ of the variables is a choice of a value for each variable; the partial setting restricted to a subset U of the variables will be denoted $\sigma|U$. A subset $U \subseteq V$ is *sufficient* with respect to setting σ if the value of f is determined by the partial setting $\sigma|U$. Such a U is a proof of the value of f under the setting $\sigma|U$; the cheapest proof of the value of f under σ is thus the cheapest sufficient set with respect to σ. We denote its cost by $\mathbf{c}(\sigma)$.

An *evaluation algorithm* $\mathcal{A}$ is a deterministic rule that queries variables sequentially, basing its decisions on the cost vector and the values of variables already queried. When an evaluation algorithm $\mathcal{A}$ is run under a setting σ, it incurs a cost that we denote $\mathbf{c}_{\mathcal{A}}(\sigma)$. We seek algorithms $\mathcal{A}$ that optimize the *competitive ratio* $\gamma^{\mathcal{A}}_{\mathbf{c}}(f) \stackrel{\text{def}}{=} \max_\sigma \mathbf{c}_{\mathcal{A}}(\sigma)/\mathbf{c}(\sigma)$. The best possible competitive ratio for any algorithm, then, is

$$\gamma_{\mathbf{c}}(f) \stackrel{\text{def}}{=} \min_{\mathcal{A}} \gamma^{\mathcal{A}}_{\mathbf{c}}(f).$$

The model above is general enough to include almost any problem in which an algorithm adaptively queries its input. Our approach will be to focus on simple functions that have been well-studied in the case of unit prices. We find that the inclusion of arbitrary prices on the inputs gives the problem a much more complex character, and leads to query algorithms that are novel and non-obvious.

Our primary focus will be on Boolean AND/OR trees (briefly, *Boolean trees*) — these are tree circuits with each leaf corresponding to a distinct variable, and without loss of generality we may assume that each root-to-leaf path has strictly alternating AND and OR gates at the internal nodes. One can easily build examples in which an optimal algorithm cannot follow a "depth-first search" style evaluation of variables and subtrees. Indeed, the criteria for optimality lead quickly to issues similar to those in the *search ratio problem* and *minimum latency problem* for weighted trees [1, 10] — problems for which polynomial-time algorithms are not known. It is not at all obvious that the optimal evaluation algorithm for an AND/OR tree can be found efficiently, or even have a succinct description, even in the case of complete binary trees.

We also consider functions that generalize AND/OR trees, including MIN/MAX game trees. Finally, we investigate analogues of searching, sorting, and selection within our model; here too, problems that are well-understood in traditional settings become highly non-trivial when prices are introduced.

Results

We provide a fairly complete characterization of the bounds achievable by optimal algorithms on AND/OR trees, and focus on three related sets of issues.

(1) Tractability of optimal algorithms. We show that for every AND/OR tree, and every cost vector, the optimal competitive ratio can be achieved by an efficient algorithm. Specifically, the algorithm has a running time that is polynomial in the size of the tree and the magnitudes of the costs. At a high level, the algorithm is based on the following natural *Balance Principle*: in each step, we try to balance the amount spent in each subtree as evenly as possible. However, to achieve the optimal ratio, this principle must be modified so that in fact we are balancing certain estimates on the lower bound for the cost of the cheapest proof in each subtree. These results are described in Section 2.

(2) Dependence of competitive ratio on the structure of f. Much of the complexity of the AND/OR tree evaluation problem is already contained in the case of complete binary trees of depth $2d$, with $n = 2^{2d}$ inputs. When the cost vector is *uniform* (all input prices are 1) the situation has a very simple analysis: any algorithm can be forced to pay n, and the cheapest proof always has value exactly $2^d = \sqrt{n}$. A natural question is therefore the following: is there is a $\sqrt{n}$-competitive algorithm for *every* cost vector on the complete binary tree? More generally, for a given AND/OR tree T, we could consider the largest competitive ratio that can be forced by any assignment of prices to the inputs:

$$\gamma(T) \stackrel{\text{def}}{=} \sup_{\mathbf{c}} \gamma_{\mathbf{c}}(T). \tag{1}$$

This definition naturally suggests the following questions: How does the above competitive ratio depend on the topology of the underlying tree? Can we characterize the structure of the cost vector $\mathbf{c}$ that achieves $\gamma_{\mathbf{c}}(T) = \gamma(T)$? We call such a cost vector $\mathbf{c}$ an *extremal* cost vector.

We prove a general characterization theorem for $\gamma(T)$; as a corollary, we find that the uniform cost vector is in fact extremal for the complete binary tree. We say that an AND/OR tree T on n inputs can *simulate* an AND gate of size k if by fixing the values of some $(n - k)$ inputs to 0, the function induced on the remaining k inputs is equivalent to a simple AND of k variables. (We define the simulation of an OR gate analogously.) We show: $\gamma(T)$ is equal to the maximum k for which T can simulate an AND gate or an OR gate of size k (this also shows that $\gamma(T)$ is always an integer). The proof is obtained using information from the lower bound estimates that form a component of our optimal balance-based algorithm. These results are described in Section 2.

We give extensions of some of these results to more general types of functions. All of these functions are defined over a tree structure, and for each we can give an efficient algorithm whose competitive ratio is within a factor of 2 of optimal.

(a) Threshold trees. Each internal node is a threshold gate; the output is `true` iff at least a certain number of the inputs are `true`. The threshold values for different gates could be different.

(b) Game trees. The inputs are real numbers, and nodes are MIN or MAX functions.

(c) A common generalization of (a) and (b). The inputs are real numbers and the nodes are gates that return the t^{th}-largest of their input values. This threshold t could be different for different nodes.

In all of this, we have been considering deterministic algorithms only. Understanding how much better one can do with a randomized algorithm is a major open direction; this would involve a generalization of earlier results on randomized tree evaluation [8, 12, 15, 16] to the setting in which inputs have prices.

(3) Equilibrium prices for a function f. Finally, we consider a "dual" issue, motivated by the following general question. Suppose many individuals are all interested in computing a function f on variables $\{x_1, \ldots, x_n\}$, and each is employing an algorithm that adaptively buys information from the n vendors that own the values of $x_1, \ldots, x_n$. What is a "natural" set of market prices arising from this process?

There are, of course, many possible answers to this question — just as there are many models for the behavior of prices in a competitive market [11]. Intuitively, one would believe that each vendor would try to charge a high price for its input, but not so high as to price itself out of competition. If we further believe that the individuals performing the queries will be using only optimal on-line algorithms, then the vendor of x_i will not want to be "priced out" of optimal on-line algorithms.

Here we describe one set of prices motivated by this intuition; it exhibits an interesting behavior with a concrete formulation. Let us say that a cost vector $\mathbf{c}$ is *ultra-uniform* with respect to a tree T if, with input prices set according to $\mathbf{c}$, *every evaluation algorithm achieves the optimal competitive ratio.* In other words, the prices are in a state such that there is no reason, from the point of view of competitive analysis, to prefer one algorithm over any other — whether an input x_i is queried relies purely on the arbitrary choice of an optimal algorithm by the individual performing the queries. We prove: for every AND/OR tree T, there is an ultra-uniform cost vector. The construction of this vector is quite natural, and follows a direct "balancing" principle of its own. These results are described in Section 3.

Searching. We also investigate a problem of a very different character, to which the same style of analysis can be applied: suppose we are given a sorted array with n positions, and wish to determine whether it contains a particular number q. In the unit-price setting, when we simply wish to minimize the number of queries to array entries, binary search solves this problem in at most $\lceil \log_2 n \rceil$ queries.

Now suppose each array entry has a price, and we seek an algorithm of optimum competitive ratio. Here the cheapest "proof" of membership of q is simply a single query to an entry containing

q; the cheapest proof of non-membership is a pair of queries to adjacent entries containing numbers less than and greater than q, respectively.

We provide an efficient algorithm for this problem that achieves the optimal competitive ratio with respect to any given cost vector. We then consider the associated *extremal problem*: which cost vector forces the largest competitive ratio? We also give an algorithm achieving a competitive ratio of $\log_2 n + O(\sqrt{\log n} \log\log n)$ for *any* cost vector; this exceeds the competitive ratio for the uniform cost vector only by lower order terms and thus the uniform cost vector is essentially the extremal vector. Whether the uniform cost vector is in fact extremal remains an interesting open question. These results are described in Section 4.

Further Directions. Our approach raises a number of other directions for further work. We now mention some of these. Sorting items when each comparison has a distinct cost appears to be highly non-trivial. Suppose, for example, we construct an instance of this problem by partitioning the items into sets A and B, giving each A-to-B comparison a very low cost, and giving each A-to-A and B-to-B comparison a very high cost. We then obtain a very simple non-uniform cost structure in the spirit of the well-known hard problem of "sorting nuts and bolts" [9].

Binary search can be viewed as a one-dimensional version of the problem of searching for a linear separator between "red" and "blue" points in d dimensions. Determining cheap, query-efficient strategies for this problem seems a lot more challenging in high dimensions. This raises the general issue of learning hypotheses from priced information. We can also generalize the binary search problem to partially ordered sets. Here it is natural to ask what can be said about good "splitters" and "central elements" in a poset, when each item has a cost.

Finally, the problem of selecting the k^{th} largest element among n items — when each comparison has a cost — is also a challenging direction to explore. Finding the median has some of the flavor of the sorting problem discussed above; but even finding the maximum is surprisingly non-trivial in this setting. We briefly discuss some partial progress on this problem in Section 5.

2 Tree Functions

We first consider functions computed by AND/OR trees: each gate may have arbitrary fan-in, but only one output. Without loss of generality, we may assume that levels of the tree alternate between AND gates and OR gates. Let such an AND/OR tree T have n leaves labeled by variables $x_1, x_2, \ldots, x_n$. Variable x_i has an associated non-negative cost c_i for reading the value of x_i. We say a 0-*witness* (resp. 1-*witness*) for T is a *minimal* set W of leaves which when set to 0 (resp. 1) will cause T to evaluate to 0 (resp. 1). The cheapest proof which allows one to prove that T evaluates to 0 (resp. 1) is always some 0-witness (resp. 1-witness).

Minterms and Maxterms. Before describing our competitive algorithms for evaluating AND/OR trees, we review the notion of minterms and maxterms of functions, since these are intimately related to 1-witnesses and 0-witnesses and we also use this terminology in the sequel. A literal refers to either a variable or its negation. For a Boolean function f on n variables $x_1, \ldots, x_n$, a minterm of f is a conjunction of some subset S of literals that implies f, and is such that no conjunction of literals in a proper subset of S implies f. A maxterm of f is a disjunction of some subset T of literals, which is implied by f, and is such that f does not imply the disjunction of literals in any proper subset of T. As an example, let f be the parity function on two variables x_1, x_2 that is true whenever exactly one of x_1, x_2 is set to 1. Then the minterms of f are $(x_1 \wedge \bar{x}_2)$ and $(\bar{x}_1 \wedge x_2)$, and the maxterms of f are $(x_1 \vee x_2)$ and $(\bar{x}_1 \vee \bar{x}_2)$. For any monotone function, all literals occurring in a minterm (or a maxterm) are positive, and therefore this will also be the case for functions computed by AND/OR trees.

Clearly, for an AND/OR tree T, a 0-witness (resp. 1-witness) consists of leaves corresponding variables that occur in a maxterm (resp. minterm) of the Boolean function computed by T.

Before moving on to algorithms for evaluating AND/OR trees, we record the following folklore fact about minterms and maxterms. We will use this later in the remark following Corollary 2.9.

Lemma 2.1 (Folklore) *Let f be a Boolean function and let M_0 and M_1 be a maxterm and a minterm respectively of f. Then M_0 and M_1 must share a common literal.*

Proof: Suppose on the contrary that M_0 and M_1 do not share any literal. Consider an assignment $\mathbf{a}$ to the variables of f that assigns FALSE to all literals in M_0 and TRUE to all literals in M_1. This is possible since M_0 and M_1 do not share any literal. For such an assignment $\mathbf{a}$ the maxterm M_0 is falsified, and this forces $f(\mathbf{a}) = 0$. Similarly, the minterm M_1 is set to TRUE by such an assignment, implying $f(\mathbf{a}) = 1$, a contradiction. This proves that M_0 and M_1 cannot share a literal. ■

2.1 Efficient algorithm achieving $\gamma(T)$

We first investigate the competitive ratio $\gamma(T)$ for any AND/OR tree T (recall the definition of Equation (1)), where the structure of T is fixed, but leaf costs vary. We propose the following simple lower bound on $\gamma(T)$. For any AND/OR tree T, let k be the largest value for which one can simulate an AND gate of fan-in k using T by hardwiring an appropriate set S_0 of $(n-k)$ leaves of T to 0. One can compute k by giving all leaves of T a value of 1, replacing the AND and OR gates of T by SUM and MAX functions respectively, and then evaluating the resulting arithmetic circuit. The following claim will be useful later on.

Claim: *Such a k is also the size of a largest minterm in the Boolean function computed by T.*

Proof: Let S be the set of variables in a largest minterm in the function computed by T. Clearly, setting all variables outside S to 0 reduces the function computed by T to an AND of the variables in S. Thus k is at least the size of a largest minterm of the function computed by T. Conversely, suppose setting all variables in S_0 to 0 reduces T to an AND gate of a subset S of k inputs. Then the conjunction of variables in S is clearly a minterm of size k of the function computed by T, and thus k is at most the size of a largest minterm of the function computed by T. ■

Now, consider the following cost vector $\mathbf{c}$: $c_i = 0$ whenever $x_i \in S_0$, else $c_i = 1$. Clearly, a 0-witness for T would now have cost exactly 1, as it would only need to contain one non-zero cost leaf whose value is 0. On the other hand, any deterministic algorithm could easily be made to pay k, simply by setting all but the last non-zero cost leaf queried to have value 1. Hence, k is a lower bound on $\gamma(T)$.

One can similarly show that the largest value ℓ for which T can simulate an OR gate of fan-in ℓ by hardwiring a set of $(n-\ell)$ leaves of T to 1 (or, equivalently, ℓ is the size of the largest maxterm in the function computed by T) is also a lower bound on $\gamma(T)$. Thus we conclude:

Lemma 2.2 *Let T be an AND/OR tree and let k, ℓ be defined as above. Then $\gamma(T) \geq \max\{k, \ell\}$.*[1]

Somewhat surprisingly this simple lower bound turns out to be tight, as we show by presenting an algorithm with competitive ratio $\max\{k, \ell\}$ for *any* setting of leaf costs. The idea behind the algorithm, which we call WEAKBALANCE, is the following: At each node in the tree, we *balance* the investment on leaves in each of the subtrees – scaling this balancing act using the lower bound ideas above. This ensures that we do not leave a potential cheap proof unexplored in any subtree.

Algorithm WEAKBALANCE: Each node x in the tree keeps track of the total cost Cost_x that the algorithm has incurred in the subtree rooted at x. At each step, the algorithm decides which leaf to read next by a process of passing recommendations up the tree: Each (remaining) leaf L passes (to its parent) a recommendation (L, c_L) to read L at cost c_L. For an internal node x, we will consider two cases: (a) Suppose x is an AND node with children $x_1, \ldots, x_t$ and it receives recommendations $(L_1, c_{L_1}), \ldots, (L_t, c_{L_t})$ from them. Let $k_1, \ldots, k_t$ be the sizes of the largest AND gates that can be induced in the subtrees rooted at $x_1, \ldots, x_t$, respectively. Then x passes the recommendation (L_i, c_{L_i}) up such that $(\mathrm{Cost}_{x_i} + c_{L_i})/k_i$ is minimized; (b) If x is an OR node, then the same process occurs with $k_1, \ldots, k_t$ replaced with the sizes of the largest inducible OR gates $\ell_1, \ldots, \ell_t$, and the recommendation passed upward is the one minimizing $(\mathrm{Cost}_{x_i} + c_{L_i})/\ell_i$. Finally, the root of the tree T decides on some recommendation (L, c_L). This leaf L is read at cost c_L, and all local total costs Cost_x's are updated, and the tree is partially evaluated as much as possible from the value of L. When the tree is fully evaluated, the algorithm terminates.

Lemma 2.3 *For any AND/OR tree T, let k and ℓ be defined (as above) as the sizes of the largest induced* AND *and* OR*, respectively. If there exists a 0-witness (resp. 1-witness) of cost c, then* WEAKBALANCE *will spend at most kc (resp. ℓc) before finding this witness.*

Proof: We proceed by induction on the size of the tree T. Clearly this holds for trees of size 1. Consider the case where the root of the tree is an AND node with children $x_1, \ldots, x_t$. Let $k_1, \ldots, k_t$ be the sizes of the largest induced AND gates rooted at each child node, and let $\ell_1, \ldots, \ell_t$ be the sizes of the largest OR gates. Observe that $k = \sum_i k_i$ while $\ell = \max_i\{\ell_i\}$.

Any 0-witness for T of cost c consists of a single 0-witness (of cost c) for a subtree rooted at some x_i. Now suppose that WEAKBALANCE has spent at least kc overall, and still has not found a 0-witness. Then, by induction hypothesis, we must have that WEAKBALANCE has spent less than $k_i c$ on node x_i. This means that for some $x_j \neq x_i$, the algorithm has spent more than $k_j c$ on x_j. Consider the last recommendation (L_j, c_{L_j}) accepted from x_j – it must be that $(\mathrm{Cost}_{x_j} + c_{L_j}) > k_j c$; on the other hand, since there is a 0-witness of cost c rooted at x_i that has not been found, by induction hypothesis, the recommendation (L_i, c_{L_i}) from x_i must be such that $(\mathrm{Cost}_{x_i} + c_{L_i}) \leq k_i c$. This is a contradiction, since the balancing rule would require the recommendation from x_i to take precedence over the one from x_j. Hence, if WEAKBALANCE spends at least kc on T, it will uncover any 0-witness of cost c. Now consider the case of a 1-witness for T of cost c, which must consist of 1-witnesses of cost c_i rooted at *every* child node x_i, with $\sum_i c_i = c$. By induction hypothesis, we know that as soon as WEAKBALANCE spends at least $\ell_i c_i$ on the subtree rooted at x_i, it will uncover the 1-witness at x_i, upon which the rest of the subtree rooted at x_i will be pruned. Thus,

[1] It is easy to see that $\max\{k, \ell\}/2$ is also a lower bound on the expected competitive ratio of any randomized algorithm.

regardless of the balancing, as soon as WeakBalance spends $\sum_i \ell_i c_i$ on T, the entire 1-witness will be uncovered. Recall that $\ell = \max_i \ell_i$, and thus $\sum_i \ell_i c_i \leq \ell \sum_i c_i = \ell c$, as desired.

An analogous argument holds for the case of an Or node, except in this case, balancing is important for finding a 1-witness, but not for finding a 0-witness. ■

Theorem 2.4 *Let k and ℓ be as in Lemma 2.3. Then, $\gamma(T) = \max\{k, \ell\}$, and* WeakBalance *runs in polynomial time and achieves a competitive ratio of $\gamma(T)$.*

Proof: Follows immediately from Lemma 2.2 and Lemma 2.3. ■

Corollary 2.5 *Let $L_1, \ldots, L_k$ $(M_1, \ldots, M_\ell)$ be the leaves corresponding to a largest induced* And *(resp.* Or*) in T. Let $\mathbf{c}_0$ (resp. $\mathbf{c}_1$) be the cost vector that assigns cost 1 to leaves $L_1, \ldots, L_k$ (resp. $M_1, \ldots, M_\ell$) and cost 0 to all other leaves. If $k > \ell$, then $\mathbf{c}_0$ is extremal for T; otherwise $\mathbf{c}_1$ is extremal for T. That is, either $\gamma_{\mathbf{c}_0}(T)$ or $\gamma_{\mathbf{c}_1}(T)$ equals $\gamma(T)$.*

Proof: It is clear that $\gamma_{\mathbf{c}_0}(T) = k$ since there exists a 0-witness of cost 1 while an algorithm can always be made to read all the k cost 1 leaves before it can figure out the value of T. Similarly, $\gamma_{\mathbf{c}_1}(T) = \ell$. By Lemma 2.2, $\gamma(T) = \max\{k, \ell\}$, and hence one of $\mathbf{c}_0$ or $\mathbf{c}_1$ is extremal for T. ■

Corollary 2.6 *If T is a complete binary tree with $n = 2^{2d}$ leaves, then $\gamma(T) = \sqrt{n}$. Hence, for such trees, the all-ones cost vector is extremal.*

Proof: It is straightforward to prove by induction (on d) that the size of every minterm and the size of every maxterm of the function computed by T equals $\sqrt{n}$, and hence using Lemma 2.2, $\gamma(T) = \sqrt{n}$.

Now consider the situation where the leaf costs are all 1. Every algorithm can be forced to read all the leaves (and thus incur a cost of n) before it can figure out the value of T. The cost of every 0-witness and 1-witness of T is exactly $\sqrt{n}$ (since all minterms and maxterms of the function computed by T have size $\sqrt{n}$). It follows that every algorithm has a competitive ratio of $\gamma(T) = \sqrt{n}$, and thus the all-ones cost vector is an extremal vector. ■

Remark: For any monotone Boolean function $f(x_1, x_2, \ldots, x_n)$, one can prove that the following simple algorithm achieves a competitive ratio of $(2\max\{k, l\})$ for any cost vector. Pick the cheapest minterm and maxterm of f, and read all variables in the cheaper of the two; if this proves that f evaluates to 0 or 1 stop, else replace f by the function f' obtained by setting the variables just read to their values, and continue with f'. The key to proving the claimed bound is the simple fact proved in Lemma 2.1 that any minterm-maxterm pair of f must share a variable, and hence the algorithm never reads more than l minterms or k maxterms. How do we compute the cheapest minterm and maxterm? For AND/OR trees this computation is actually easy, and this gives a simple polynomial-time $(2\max\{k, l\})$-competitive algorithm for AND/OR tree evaluation, for any cost vector. (We achieve the stated competitive ratio because the costs incurred in reading the variables involved in the minterms we pick and those involved in the maxterms we pick add up, but each of these costs is at most $\max\{k, \ell\}$ times the cost of the cheapest witness.) WeakBalance does not lose a factor 2 in the competitive ratio, and more importantly, generalizing its approach enables us to devise an algorithm Balance that is optimal for any given cost vector, as is described in the next section.

2.2 Optimal Algorithm for given cost vector

For a particular vector $\mathbf{c}$ of costs, the optimal competitive ratio $\gamma_{\mathbf{c}}(T)$ can be much less than $\gamma(T)$, the ratio guaranteed by WEAKBALANCE. These observations lead us to more exact lower bounds and to our algorithm BALANCE that, for any tree T and cost vector $\mathbf{c}$, achieves the optimal competitive ratio $\gamma_{\mathbf{c}}(T)$. The key to developing this algorithm is to define certain *lower bound functions* that are more refined than the minterm-maxterm based lower bounds of WEAKBALANCE. For any AND/OR tree T and cost vector $\mathbf{c}$, we define functions $f_0^T(x)$ and $f_1^T(x)$ representing lower bounds on the cost that any deterministic algorithm must incur in finding a 0-witness (or 1-witness, respectively) of S of cost at most x.[2] These functions imply that for any tree T, every deterministic algorithm must have a competitive ratio of at least the maximum of $\max_x\{f_0^T(x)/x\}$ and $\max_x\{f_1^T(x)/x\}$.

Lower Bound Functions. For an AND/OR tree T, the functions f_0^T and f_1^T are computed in a bottom-up manner moving from the leaves to the root of the tree.

- For a leaf L with cost c, we have

$$f_0^L(x) = f_1^L(x) = \begin{cases} 0 & \text{if } x < c \\ c & \text{if } x \geq c. \end{cases}$$

- For a subtree S, let r_S denote the root of S, and let $S_1, S_2, \ldots, S_t$ be the subtrees rooted at the children of r_S. Suppose we already know the functions $f_0^{S_i}$ and $f_1^{S_i}$; our goal is to compute f_0^S and f_1^S from these functions. There are two cases that arise now depending upon whether r_S is an AND node or an OR node.

  (1) r_S is an AND node: A minimal 0-witness for S consists of exactly one 0-witness for some subtree. The adversary can thus choose to "hide" this witness in any of the subtrees, which suggests the bound (2) we define below. On the other hand, a minimal 1-witness for S consists of 1-witnesses from each of the subtrees. Thus, the adversary's only choice is to pick such 1-witnesses in a manner that maximizes any deterministic algorithm's expenditure, which suggests the other bound (3) we define below. Formally, we define[3]

$$f_0^S(x) = \sum_{1 \leq i \leq t} f_0^{S_i}(x). \tag{2}$$

$$f_1^S(x) = \max_{\substack{\{x_i : 1 \leq i \leq t\} \\ \sum_i x_i = x}} \Big(\sum_{1 \leq i \leq t} f_1^{S_i}(x_i) \Big). \tag{3}$$

  (2) r_S is an OR node: Here the situation is exactly reversed from that of an AND node. Thus, we define[4]

$$f_1^S(x) = \sum_{1 \leq i \leq t} f_1^{S_i}(x). \tag{4}$$

[2]These functions are actually functions of $\mathbf{c}$ as well; we omit this dependence for notational convenience.

[3]In Equation (3), the max operator is taken only over those x_i such that there can exist a 1-witness in S_i of cost at most x_i. If no such $x_1 \ldots x_t$ exist for a particular x, then $f_1^S(x) = 0$.

[4]In Equation (5), the max operator is taken only over those x_i such that a 0-witness can exist in S_i of cost at most x_i. If no such $x_1 \ldots x_t$ exist for a particular x, then $f_0^S(x) = 0$.

$$f_0^S(x) = \max_{\substack{\{x_i:1\le i\le t\}\\ \sum_i x_i = x}} \Big(\sum_{1\le i\le t} f_0^{S_i}(x_i) \Big). \tag{5}$$

Remark: It is easy to see that the definitions above imply $f_0^T(c) = 0$ (resp. $f_1^T(c) = 0$) if T has no 0-witness (resp. 1-witness) of cost c or less.

Time complexity of computing f_0^T and f_1^T: The functions f_1^L and f_0^L are *step functions* when L is a leaf and therefore it is easy to see that the functions f_1^T and f_1^T are also step functions for any AND/OR tree T. Hence all the functions above have a *compact* (of size polynomial in the number of leaves and the sum of the costs) representation as a table of values. Moreover, this representation can be computed efficiently: It is clear that the operations of Equations (2) and (4) can be performed efficiently. For Equation (3) (Equation (5) is similar), clearly the computation can be done in polynomial time, say τ, when $t = 2$. For larger values of t, we can first compute a table of values for $f^{S'}$ where S' is a (virtual) subtree with an AND node as root and S_{t-1} and S_t as children, and f_1^S can now be expressed in terms of only $(t-1)$ functions $f_1^{S_1}, \ldots, f_1^{S_{t-2}}, f_1^{S'}$. Repeating above $(t-1)$ times in all, we can thus evaluate the table of values corresponding to f_1^S in time polynomial in the number of leaves and the sum of the costs of the leaves.

Later, in the specification of our algorithm, we will also be referring to the inverses $(f_0^T)^{-1}$ and $(f_1^T)^{-1}$ of these functions. Since these functions are not injective, this is loose notation. By $f^{-1}(y)$, we actually mean $\min\{x : f(x) \ge y\}$. Also, for ease of notation, we sometimes refer to f_0^S and f_1^S for a subtree rooted at a node x also as f_0^x and f_1^x respectively.

We now claim that the above are actually lower bound functions which have some additional nice properties.

Proposition 2.7 *If T is an arbitrary tree, then $f_0^T(c)$ (resp. $f_1^T(c)$) is a lower bound on the cost any algorithm must incur in the worst case in order to find a 0-witness of cost at most c (resp. 1-witness of cost at most c). More specifically, there is an adversary strategy that ensures that, as long as any algorithm has incurred a cost strictly less than $f_0^T(c)$ (resp. $f_1^T(c)$):*

(1) It does not find a 0-witness (resp. 1-witness) of cost at most c.

(2) The partial assignment to the leaves that have been read can be extended so that a 0-witness (resp. 1-witness) of cost at most c exists, and can also be extended so that every 0-witness (resp. 1-witness), if any at all, has cost strictly more than c.

Proof: The proof works by inductively moving upward from the leaves to the root of the tree T. For the leaves, the claim of the proposition is clearly satisfied; if c is the cost of the leaf, then the cost of a 0-witness and 1-witness are both c. Unless an algorithm incurs a cost of c, the adversary can always set the leaf to be 0 when it is queried, thereby creating a 0-witness of cost c, and can instead set it to 1 in which case there is no 0-witness at all (and therefore trivially every 0-witness has cost more than c).

Suppose S is a subtree whose root r_S is an AND node with subtrees $S_1, S_2, \ldots, S_t$ rooted at its t children. We want to prove that, assuming $f_0^{S_i}$ and $f_1^{S_i}$ satisfy the conditions of the proposition, the definition of f_0^S and f_1^S as per the Equations (2) and (3) above also satisfies the requirement of the Proposition.

We first consider the case when the algorithm is trying to find a 0-witness of cost at most c. Note that since r_S is an AND node, the 0-witness is simply a 0-witness of one of the subtrees S_i. The adversary strategy to "hide" a 0-witness of cost at most c is as follows: The basic idea is to use, for each subtree S_i, the strategy for S_i guaranteed by induction. More specifically, for the first $(t-1)$ subtrees S_j (excluding S_k for some k) for which the algorithm ends up spending an amount that is at least $f_0^{S_j}(c)$, ensure that there is **no** 0-witness for S_j of cost at most c. This can be done using part (2) of the induction hypothesis, since as long as the algorithm has spent *strictly less than* $f_0^{S_j}(c)$, the adversary strategy ensures that: (a) it does not evaluate S_j, and (b) the partial assignment can be extended so that when the algorithm eventually ends up spending at least $f_0^{S_j}(c)$, there is *no* 0-witness for S_j of cost at most c. For the "last" subtree S_k, use the inductive strategy for S_k to hide a 0-witness of cost c till the algorithm spends $f_0^{S_k}(c)$.

Now suppose an algorithm has spent a total cost C which is less than the "lower bound function" $f_0^S(c) = \sum_i f_0^{S_i}(c)$ as per Equation (2). Then there exists a k, $1 \leq k \leq t$, such that the algorithm has spent less than $f_0^{S_k}(c)$ on S_k, and hence the above adversary strategy ensures that the algorithm has not found a 0-witness for S. It is also clear that the adversary has the option of either extending the partial assignment so that a 0-witness of cost at most c exists, or so that every 0-witness for S, if any at all, has cost more than c.

Now we consider the case when the algorithm is trying to find a 1-witness of cost at most c. We may assume that $f_1^S(c) > 0$ for otherwise the statement of the Proposition holds vacuously. Note that a 1-witness of cost c for S consists of 1-witnesses for S_i of cost c_i for $1 \leq i \leq t$ with $\sum_i c_i = c$. Let us pick $c_1, c_2, \ldots, c_t$ for which the maximum in Equation (3) is attained. By our assumption on Equation (3), there exist 1-witnesses for S_i of cost at most c_i for every $i \in \{1, \ldots, t\}$. The adversary strategy now is as follows: for the first $(t-1)$ subtrees S_j (excluding S_k for some k), for which the algorithm incurs a cost of at least $f_1^{S_j}(c_j)$, the adversary causes S_j to evaluate to 1 through a 1-witness of cost at most c_j (using the strategy for each subtree guaranteed by the induction hypothesis), and thus it reduces the value of S to the value of S_k. Meanwhile, for S_k, the adversary also uses the strategy for S_k to hide a witness of cost c_k until the algorithm spends $f_1^{S_k}(c_i)$. As long as any algorithm has incurred a cost (strictly) less than $f_1^S(c)$, this strategy leaves the adversary with the option of either creating a 1-witness of cost at most c or ensuring that every 1-witness of S has cost more than c. This completes the proof for the case when S is rooted at an AND node; the other case when it is rooted at an OR node is handled similarly. ■

The BALANCE **Algorithm.** We now show how to use the lower bound functions described above to derive an algorithm, which we call BALANCE, that achieves the best possible competitive ratio for any fixed cost vector. The high level idea behind BALANCE is the same as WEAKBALANCE: At each intermediate node, we *balance* the amount spent on reading leaves in each of the subtrees – by "balancing" we do not necessarily mean that the exact amounts spent are all nearly equal, rather we mean that the costs of the possible witnesses that can still be found in all the subtrees are of nearly equal cost, so that after spending a huge amount, we do not still leave the possibility of there existing a cheap witness in some unexplored part of the tree. BALANCE actually uses the above lower bound functions f_0^T and f_1^T for the balancing criterion. The algorithm is formally described in Figure 2.

We want to prove that BALANCE indeed achieves the optimal competitive ratio $\gamma_{\mathbf{c}}(T)$ for every

Algorithm BALANCE:

Input: A AND/OR tree T with a cost vector $\mathbf{c}$ on its n leaves.
Output: The value of the tree T.

/* For each node x, we keep track of the total cost Cost_x incurred on the subtree rooted at x. */

Let $\mathrm{Cost}_x = 0$ for all nodes x in the tree.

Compute the lower bound functions f_0^x and f_1^x for all nodes x of T. (Actually we will only be referring to the "inverses" of these functions.)

While T is not fully evaluated

1. Moving up the tree from the leaves to the root:

   (a) Each leaf L which has not been read or pruned yet passes a recommendation $R_L = (L, c_L)$ up to its parent. (c_L is the cost of leaf L.)

   (b) Each internal node x of the tree that receives recommendations $R_1, R_2, \ldots, R_t$, with $R_i = (L_i, c_{L_i})$, from its t (not yet pruned) children $x_1, x_2, \ldots, x_t$ chooses one of its children as follows:
   (i) If x is an AND node, choose the child x_q with the minimum value of $(f_0^{x_q})^{-1}(c_{L_q} + \mathrm{Cost}_{x_q})$.
   (ii) If x is an OR node, choose the child x_q with the minimum value of $(f_1^{x_q})^{-1}(c_{L_q} + \mathrm{Cost}_{x_q})$.
   (ties are broken arbitrarily)

   Node x then propagates the recommendation R_q from x_q up to its parent
   (unless x is the root in which case goto Step 2)

   /* At this point recommendations have passed upward to the root from the leaves. */

2. /* Now we are at the root r and say it chose a recommendation $R_L = (L, c_L)$. */
   The value of the leaf L is read at a cost of c_L.

3. For all ancestors y of L in T the total cost incurred on their subtree is increased by c_L, i.e perform $\mathrm{Cost}_y = \mathrm{Cost}_y + c_L$.

endWhile

Output the value of the tree T.

Figure 2: The BALANCE Algorithm.

AND/OR tree T and cost vector $\mathbf{c}$. For this we prove below that if there is a witness (for T evaluating to either 0 or 1) of cost at most c, then BALANCE discovers the witness by spending a total cost that is at most $\max\{f_0^T(c), f_1^T(c)\}$. In conjunction with Proposition 2.7, note that this immediately implies that BALANCE achieves the optimum competitive ratio possible for any deterministic algorithm; indeed any deterministic algorithm has a competitive ratio of at least $\max\left[\max_x\{f_0^T(x)/x\}, \max_x\{f_1^T(x)/x\}\right]$, and BALANCE achieves this competitive ratio.

Theorem 2.8 *If there exists a* 0*-witness (resp.* 1*-witness) for* T *of cost at most* c*, then* BALANCE *proves that* T *evaluates to* 0 *(resp.* 1*) by spending at most* $f_0^T(c)$ *(resp.* $f_1^T(c)$*).*

Proof: The proof once again works by inductively moving up the tree from the leaves to the root. When T just consists of a leaf L, the statement of the theorem clearly holds. Now suppose the root r of T is an AND node (the other case can be handled similarly) with children $x_1, x_2, \ldots, x_t$. Let T_i be the subtree rooted at x_i, for $1 \leq i \leq t$. We will prove the that if BALANCE ever spends an amount *strictly greater than* $f_0^T(c)$ (resp. $f_1^T(c)$) then T has **no** 0-witness (resp. 1-witness) of cost at most c, and this will clearly imply the statement of the theorem.

First, suppose BALANCE spends an amount strictly greater than $f_1^T(c)$ when evaluating T, and yet T has a 1-witness W of cost at most c. Since r is an AND node, W is a collection of 1-witnesses W_i of cost c_i for T_i, $1 \leq i \leq t$, with $c = \sum_{i=1}^t c_i$. By the definition of $f_1^T(c)$ in Equation (3), this implies that there exists k, with $1 \leq k \leq t$, such that BALANCE spends more than $f_1^{T_k}(c_k)$ on reading leaves in T_k. By induction, however, this implies that T_k has **no** 1-witness of cost c_k or less, a contradiction to the existence of W_k. Hence if BALANCE spends more than $f_1^T(c)$, then it rules out the possibility of T having any 1-witness of cost c or less. Note that the above argument did not use any specific properties of BALANCE; this is due to the special structure of a 1-witness at an AND node, but the “balancing” principle is crucially used below for the case of 0-witnesses at an AND node.

We now consider the case of 0-witnesses. Suppose BALANCE has spent an amount more than $f_0^T(c) = \sum_{i=1}^t f_0^{T_i}(c)$ and yet there is a 0-witness W of cost c; we will then arrive at a contradiction. Using the fact that r is an AND node, the witness W is simply a 0-witness W_i of cost c for *some* i, $1 \leq i \leq t$, say for definiteness, it is a 0-witness W_t for T_t. Consider the first time when BALANCE goes over $f_0^T(c)$ in its total expenditure. By induction, we know that BALANCE never spends more than $f_0^{T_t}(c)$ on T_t (or else there could not be a 0-witness W_t of cost at most c). Formally, this means that if (L_t, c_{L_t}) is the current recommendation from x_t to the root r, then we have $\mathrm{Cost}_{x_t} + c_{L_t} \leq f_0^{T_t}(c)$. Since on the whole BALANCE has spent more than $\sum_{i=1}^t f_0^{T_i}(c)$, there must exist a j, $1 \leq j < t$, say for definiteness $j = 1$, such that BALANCE has spent more than $f_0^{T_1}(c)$ on T_1. Now consider the point when BALANCE chose the recommendation $R_1 = (L_1, c_{L_1})$ from T_1 and went above $f_0^{T_1}(c)$ on its expenditure on T_1, so that $\mathrm{Cost}_{x_1} + c_{L_1} > f_0^{T_1}(c)$. At this point, it rejected the recommendation $R_t = (L_t, c_{L_t})$ from T_t which we know satisfies $\mathrm{Cost}_{x_t} + c_{L_t} \leq f_0^{T_t}(c)$. But we then have $(f_0^{T_t})^{-1}(\mathrm{Cost}_{x_t} + c_{L_t}) \leq c < (f_0^{T_1})^{-1}(\mathrm{Cost}_{x_1} + c_{L_1})$. Thus BALANCE would have never chosen the recommendation from T_1 over that of T_t (here we are using the fact at levels where the parent is an AND node, BALANCE uses the function f_0^T to decide whose recommendation to take), a contradiction. Hence there *cannot* be a 0-witness of cost at most c as we supposed, and we are done. ∎

Corollary 2.9 *For any AND/OR tree T and cost vector $\mathbf{c}$,* Balance *achieves a competitive ratio of $\gamma_{\mathbf{c}}(T)$.*

Remark: We will claim statements similar to the above Corollary in Sections 2.3 and 2.4, but those will guarantee only a competitive ratio of $2\gamma_{\mathbf{c}}(T)$, i.e. we will lose a factor of two in the competitive ratio. This does not happen for AND/OR trees due to their special structure which allows us to use *only one* of the functions f_0 or f_1 in the balancing criterion at each level. In the case of threshold trees which we consider next, this will no longer be the case, and we will need to use *both* f_0 and f_1 at each level, and this will incur a factor two loss in the competitive ratio.

2.3 Threshold Trees

Observe that And and Or gates are both *threshold gates*, i.e., their output is 1 provided sufficiently many of its inputs are set to 1. It turns out the Balance algorithm of the previous sections can be modified to competitively evaluate *threshold trees* as well: a threshold tree is a tree where each internal node is a threshold (t,p)-gate for some values of t,p, where the output of a (t,p)-gate is 1 if and only if at least p of its t inputs are 1. The values of the threshold p can vary over the nodes of the tree. The algorithm for evaluating threshold trees is similar to Balance with appropriate lower bound functions defined for threshold gates akin to the functions defined for And and Or gates. The structure of witnesses is more general than for AND/OR trees, and we discuss this next.

Structure of Witnesses for Threshold Trees: One important change in the case of threshold trees is that the structure of 0-witnesses and 1-witnesses get more complicated compared to the AND/OR tree case. In the AND/OR tree case, since And and Or gates alternated between levels, a 0-witness (and also a 1-witness) had the structure that at alternate levels either all children are picked or only one of the children is picked. This implied that in Balance, at each node only one of the two functions f_0, f_1 had to be used to decide which recommendation to accept, and we could just go ahead and use that function to pick the appropriate recommendation. In the case of threshold trees, however, this nice structure does not exist, and hence we need to run two algorithms in parallel (balancing the cost they incur at any point), one of which uses f_1 and the other uses f_0 as the balancing criterion. This could leave up to a factor 2 loss in the competitive ratio of the algorithm. We next specify the lower bound functions for general threshold gates.

Lower Bound Function for Threshold Gates: Suppose a subtree S of a threshold tree has a (t,p)-gate at its root r and let $S_1,\ldots,S_k$ be the subtrees rooted at the children of r. We define:[5]

$$f_1^S(x) = \max_I \Big[\max_{\substack{x_1,\ldots,x_{p-1}:\\ \sum_j x_j \le x}} \{ f_1^{S_{i_1}}(x_1) + \cdots + f_1^{S_{i_{p-1}}}(x_{p-1}) + \textstyle\sum_{i \notin I} f_1^{S_i}(x - \sum_j x_j) \} \Big] \tag{6}$$

[5]In Equation (6), the first max operator is taken over choices of $I = \{i_1, i_2, \ldots, i_{p-1}\} \subseteq \{1,\ldots,t\}$, and the second max operator is taken only over choices of $x_1,\ldots,x_{p-1}$ such that: (a) there exist 1-witnesses in $S_{i_1},\ldots,S_{i_{(p-1)}}$ of cost at most $x_1,\ldots,x_{p-1}$, respectively; and (b) there exists some $i \notin I$ such that a 1-witness can exist in S_i of cost at most $x - \sum_j x_j$. Again, if no such $x_1 \ldots x_{p-1}$ exist for a particular x, then the value of the max is 0. Similarly, in Equation (7), the first max operator is taken over choices of $I = \{i_1, i_2, \ldots, i_p\} \subseteq \{1, 2, \ldots, t\}$, and the second max operator is taken only over choices of $x_1,\ldots,x_p$ such that there exist 1-witnesses in $S_{i_1},\ldots,S_{i_p}$ of cost at most $x_1,\ldots,x_p$, respectively. If no such $x_1 \ldots x_p$ exist for a particular x, then the value of the max is 0.

In Proposition 2.11, we will prove that the above is a lower bound function for the cost of finding 1-witnesses in threshold trees. We first prove that the above equation is equivalent to another form which will be useful for proving the optimality of the modified BALANCE algorithm.

Lemma 2.10 *Let $f_1^S(x)$ be defined as in Equation (6). Then*

$$f_1^S(x) = \max_I \Big[\max_{\substack{x_1,\dots,x_p:\\ \sum_j x_j = x}} \{ \quad f_1^{S_{i_1}}(x_1) + \cdots + f_1^{S_{i_p}}(x_p) + \textstyle\sum_{i \notin I} f_1^{S_i}(\max x_j) \} \Big] \tag{7}$$

Proof: Let $F_1^S(x)$ denote the function defined on the right hand side of Equation (7). We want to prove that the functions F_1^S and f_1^S are equal. We first show $f_1^S(x) \geq F_1^S(x)$ for every x. Indeed, let $I = \{i_1, \dots, i_p\}$ and $x_1, \dots, x_p$ attain the maximum in Equation (7), and let $x_p = \max x_j$ for definiteness. Then $I' = \{i_1, \dots, i_{p-1}\}$ and $x_1, \dots, x_{p-1}$ attain the same value in Equation (6).

Conversely, let $I' = \{i_1, \dots, i_{p-1}\}$ and $x_1, \dots, x_{p-1}$ attain the maximum in Equation (6), and let $x_p = x - \sum_{j=1}^{p-1} x_j$. Let i_p be any element of $\{1, 2, \dots, t\} \setminus I'$. Now consider the value attained by Equation (7) for the choices $I = I' \cup \{i_p\}$ and $x_1, \dots, x_p$. This equals $\sum_{j=1}^{p} f_1^{S_{i_j}}(x_j) + \sum_{i \notin I} f_1^{S_i}(\max x_j)$ which is certainly at least $\sum_{j=1}^{p-1} f_1^{S_{i_j}}(x_j) + \sum_{i \notin I'} f_1^{S_i}(x_p)$. By the choice of I' and $x_1, \dots, x_{p-1}$, this latter quantity equals $f_1^S(x)$. Thus $F_1^S(x) \geq f_1^S(x)$ as well, and we conclude that $F_1^S(x) = f_1^S(x)$ for every x. ∎

The equations for f_0^S are obtained by writing the above equation with $p' = t - p + 1$ instead of p since the complement of a (t, p)-gate is a $(t, t - p + 1)$-gate.[6]

Modified Balance for Threshold Trees: There are two algorithms BALANCE$_0$ and BALANCE$_1$ running in parallel. BALANCE$_0$ uses f_0 and attempts to find a 0-witness, and BALANCE$_1$ uses f_1 and attempts to find a 1-witness. Below specify how BALANCE$_0$ passes recommendations up from nodes to parents in selecting which leaf to evaluate next.

Each internal node x of the tree that receives recommendations $R_1, R_2, \dots, R_t$, with $R_i = (L_i, c_{L_i})$, from its t (not yet pruned) children $x_1, x_2, \dots, x_t$ chooses the child x_q with the minimum value of $(f_0^{x_q})^{-1}(c_{L_q} + \mathrm{Cost}_{x_q})$.

BALANCE$_1$ is similar to BALANCE$_0$ with f_0 replaced with f_1. We keep track of the total cost incurred by BALANCE$_0$ and BALANCE$_1$ so far. At every stage, both BALANCE$_0$ and BALANCE$_1$ separately recommend a leaf to be evaluated next. We choose the recommendation that minimizes the total cost incurred by BALANCE$_0$ or BALANCE$_1$, where the total cost includes the cost of the new recommendation.

Proposition 2.11 *If T is an arbitrary threshold tree, then $f_1^T(x)$ (resp. $f_0^T(x)$) is a lower bound on the cost any algorithm must incur in the worst case in order to find a 1-witness (resp. 0-witness) of cost at most x. More specifically, there is an adversary strategy that ensures that, as long as an algorithm has incurred a cost strictly less than $f_1^T(x)$ (resp. $f_0^T(x)$):*

[6]For our algorithm, it is important that these functions f_0^S and f_1^S can also be computed in polynomial time; this turns out to be true using an argument similar to but more complicated than the one we used for the AND/OR tree case.

(1) It does not find a 1*-witness (resp.* 0*-witness) of cost at most* x*.*

(2) The partial assignment to the leaves that have been read can be extended so that a 1*-witness (resp.* 0*-witness) of cost at most* x *exists, and also be extended so that no* 1*-witness (resp.* 0*-witness) exists.*

Proof: We will describe an adversary strategy that forces any evaluation algorithm for threshold tree T to spend at least $f_1^T(x)$ in finding a 1-witness of cost at most x for T. The proof of the proposition for $f_0^T(x)$ follows in a similar fashion. Our proof proceeds by induction on the tree structure proceeding bottom up from the leaves to the root. For the leaves, the claim of the Proposition is clearly true (see the base case in the proof of Proposition 2.7).

Let S be a subtree whose root r_S is a (t, p) threshold node with subtrees $S_1, S_2, \ldots, S_t$, such that the proposition holds for $f_1^{S_i}$. Consider the subset $I = \{i_1, i_2, \ldots, i_{p-1}\} \subseteq \{1, 2, \ldots, t\}$ that maximizes the argument to the first max operator in the expression for $f_1^T(x)$ (Equation (6)), and the values $x_1, x_2, \ldots, x_{p-1}$ that maximize the argument to the second max operator. Let $x' = x - \sum_{j=}^{p-1} x_j$. The adversary strategy for subtree S is obtained by appropriately combining the adversary strategies for the subtrees S_i (guaranteed by the inductive hypothesis). For each of the subtrees $S_{i_r}, i_r \in I$, the adversary hides a 1-witness of cost x_r till the algorithm spends $f_1^{S_{i_r}}(x_r)$. In addition, the adversary hides a 1-witness of cost at most x' in one of the remaining $t - p + 1$ subtrees $S_i, i \in \{1, \ldots, t\} \setminus I$. For the first $(t - p)$ of these subtrees S_i for which the algorithm ends up spending at least $f_1^{S_i}(x')$, the adversary ensures (using part (2) of the inductive hypothesis) that there is no 1-witness of cost at most x'. For the "last" subtree S_i, the adversary uses the inductive strategy for S_i to hide a 1-witness of cost x' till the algorithm spends $f_1^{S_i}(x')$.

Suppose the algorithm has spent a total cost less than the lower bound function $f_1^S(x)$. Then either

1. there exists an $r \in \{1, \ldots, p-1\}$ such that the algorithm has spent less than $f_1^{S_{i_r}}(x_r)$ on S_r, or
2. there exists an $i \in \{1, \ldots, t\} \setminus I$ such that the algorithm has spent less than $f_1^{S_i}(x')$ on S_i.

Hence the above strategy ensures that the algorithm has not found a 1-witness for S. Also, the adversary has the option of either extending the partial assignment so that a 1-witness of cost at most x exists, or so that there is no 1-witness for S (i.e. S evaluates to 0). ■

Theorem 2.12 *If there exists a* 1*-witness (resp.* 0*-witness) for* T *of cost at most* x*, then* Balance$_1$ *(resp.* Balance$_0$*) proves that* T *evaluates to* 1 *(resp.* 0*) by spending at most* $f_1^T(x)$ *(resp.* $f_0^T(x)$*).*

Proof: We will describe the proof for 1-witnesses; the proof for 0-witnesses is similar. We will prove that if Balance$_1$, when running on $(T, \mathbf{c})$, spends an amount which is *strictly greater than* $f_1^T(x)$, then there exists **no** 1-witness for T which has cost at most x. This will clearly imply the statement of the theorem. The proof again works by induction on the tree structure proceeding bottom up from the leaves to the root.

Let S be a subtree whose root r_S is a (t, p) threshold node with subtrees $S_1, S_2, \ldots, S_t$. For $1 \leq i \leq t$, let S_i be rooted at x_i, and assume that the proposition holds for $f_1^{S_i}$. Now, suppose Balance$_1$ spends an amount strictly greater than $f_1^S(x)$ in evaluating subtree S, and yet there exists a 1-witness W for S of cost at most x. Since the root of S is a (t,p) threshold node, W consists of 1-witnesses $W_{i_1},\ldots,W_{i_p}$ for p of the subtrees $S_{i_1},\ldots,S_{i_p}$ (say W_{i_r} has cost x_r). Let $I=\{i_i,\ldots,i_p\}$ and $x'=\max_{1\le j\le r} x_j$. By the definition of $f_1^T(x)$ (Equation 7), we have

$$f_1^S(x) \ge f_1^{S_{i_1}}(x_1)+\cdots+f_1^{S_{i_p}}(x_p)+\sum_{i\notin I} f_1^{S_i}(x') \ .$$

Since the algorithm spends more than $f_1^S(x)$ on subtree S, either

1. for some $r\in\{1,\ldots,p\}$, it spends more than $f_1^{S_{i_r}}(x_r)$ on subtree S_r, or
2. for some $i\notin I$, it spends more than $f_1^{S_i}(x')$ on subtree S_i.

We will consider both cases:

Case 1: Since subtree S_{i_r} has a 1-witness W_{i_r} of cost x_r for $r\in\{1,\ldots,p\}$, the induction hypothesis implies that BALANCE$_1$ does not spend more than $f_1^{S_{i_r}}(x_r)$ on S_{i_r}, a contradiction.

Case 2: By induction hypothesis, we know that, for $r\in\{1,2,\ldots,p\}$, BALANCE$_1$ never spends more than $f_1^{S_{i_r}}(x_r)$ on subtree S_{i_r} (since it has a 1-witness of cost x_r). Also, if it does spend $f_1^{S_{i_r}}(x_r)$, then it is guaranteed to find a 1-witness in subtree S_{i_r}. We assume that the algorithm has not yet found a 1-witness in S. Hence, there exists an $r\in\{1,\ldots,p\}$ such that the algorithm has spent *strictly less than* $f_1^{S_{i_r}}(x_r)$ and has not found a 1-witness in subtree S_{i_r}. On the other hand, the algorithm spends more than $f_1^{S_i}(x')$ on subtree S_i for some $i\notin I$. Consider the point when BALANCE$_1$ chose the recommendation (L_i,c_{L_i}) from S_i and exceeded $f_1^{S_i}(x')$ in its expenditure on subtree S_i, so that $\mathrm{Cost}_{x_i}+c_{L_i}>f_1^{S_i}(x')$. At this point, it rejected the recommendation $(L_{i_r},c_{L_{i_r}})$ from S_{i_r} which we know satisfies $\mathrm{Cost}_{x_{i_r}}+c_{L_{i_r}}\le f_1^{S_{i_r}}(x_r)$. But then, $(f_1^{S_{i_r}})^{-1}(\mathrm{Cost}_{x_{i_r}}+c_{L_{i_r}})\le x_r\le x'<(f_1^{S_i})^{-1}(\mathrm{Cost}_{x_i}+c_{L_i})$ (here we used the fact that $x'=\max_{1\le j\le p}x_j$). Thus, BALANCE$_1$ would never have chosen the recommendation from S_i over that of S_{i_r}, a contradiction.

The contradiction in both cases proves that there cannot be a 1-witness for S of cost at most x, and we are done. ∎

Theorem 2.13 *For any threshold tree T and any cost vector $\mathbf{c}$, there is a polynomial time algorithm for evaluating T with competitive ratio at most $2\gamma_{\mathbf{c}}(T)$.*

2.4 Game Trees

We can in fact generalize BALANCE to competitively evaluate *game trees* (also called MIN/MAX trees). A MIN/MAX tree has real values on its leaves and the internal nodes are MIN and MAX functions; our goal is to evaluate the value of the root.

Modified Balance for Game Trees:

We generalize the notion of a 0-witness and a 1-witness for AND/OR trees to a U-witness (upper bound witness) and an L-witness (lower bound witness) for MIN/MAX trees. The generalization comes from the fact that AND/OR trees are MIN/MAX trees in the restricted setting where all inputs are 0/1. A 0-witness can be viewed as a proof that the value of the AND/OR tree is at most 0 (i.e. an upper bound witness) and a 1-witness can be viewed as a proof that the value of the AND/OR tree is at least 1 (i.e. a lower bound witness). A U-witness that proves an upper bound UB on the value of the MIN/MAX tree is a set of leaves with an assignment of values to them that causes the MIN/MAX tree to evaluate to at most UB irrespective of the values of the remaining leaves. In general, since the value of the MIN/MAX tree is monotone in the value of each of the leaves, we can compute the upper bound UB corresponding to a U-witness by evaluating the AND/OR tree for the assignment specified by the upper bound witness on the subset of leaves in the witness and setting the remaining leaves to $+\infty$. A U-witness for a tree rooted at a MIN node x consists of a U-witness for a subtree rooted at one of the children x_i of x; on the other hand, if the tree is rooted at a MAX node x, a U-witness consists of U-witnesses for the subtrees rooted at each of the children x_i of x. Note that a U-witness has the same structure as a 0-witness. Similarly, an L-witness has the same structure as a 1-witness.

The lower bound functions used are exactly the same as in the algorithm for evaluating AND/OR trees. For computing the lower bound functions, a MIN node is treated as an AND node and a MAX node is treated as an OR node. The function f_0^T will be referred to as f_U^T as it is used in proving *upper* bounds on the value of the MIN/MAX tree. On the other hand, f_1^T will be used in proving lower bounds on the value of the MIN/MAX tree and will be referred to as f_L^T. The fact that AND/OR trees are a special case of MIN/MAX trees immediately implies, by Proposition 2.7, that f_U^T (resp. f_L^T) are valid lower bound functions on the (worst-case) cost that has to be incurred by *every* algorithm, in order to prove upper bounds (resp. lower bounds) on the value of T.

We describe how a modified BALANCE algorithm, call it BALANCE$_U$, is used to compute an upper bound on the value of the MIN/MAX tree. For every node x in the tree, the algorithm maintains an upper bound UB_x on the value of the MIN/MAX tree rooted at x. This is updated as leaves are examined by the algorithm (the upper bound is initialized to ∞).

Each internal node x of the tree that receives recommendations $R_1, R_2, \ldots, R_t$, with
$R_i = (L_i, c_{L_i})$, from its t children $x_1, x_2, \ldots, x_t$ chooses one of its children
as follows:

(i) If x is a MIN node, choose the child x_q with the minimum value of $(f_U^{x_q})^{-1}(c_{L_q} + \text{Cost}_{x_q})$.
(ii) If x is a MAX node, choose the child x_q with the maximum value UB_{x_q}.
(ties are broken arbitrarily)

The modified BALANCE algorithm, call it BALANCE$_L$, that computes a lower bound on the value of the MIN/MAX tree is similar. For every node x in the tree, the algorithm maintains a lower bound LB_x on the value of the MIN/MAX tree rooted at x. This is updated as leaves are examined by the algorithm.

Each internal node x of the tree that receives recommendations $R_1, R_2, \ldots, R_t$, with
$R_i = (L_i, c_{L_i})$, from its t children $x_1, x_2, \ldots, x_t$ chooses one of its children
as follows:

(i) If x is a MAX node, choose the child x_q with the minimum value of $(f_L^{x_q})^{-1}(c_{L_q} + \text{Cost}_{x_q})$.
(ii) If x is a MIN node, choose the child x_q with the minimum value LB_{x_q}.
(ties are broken arbitrarily)

Theorem 2.14 *If there exists a U-witness (resp. L-witness) for tree T of cost at most c that proves an upper bound UB (resp. lower bound LB) on the value of T, then* BALANCE$_U$ *(resp.* BALANCE$_L$*) proves that T evaluates to at most UB (resp. at least LB) by spending at most $f_U^T(c)$ (resp. $f_L^T(c)$).*

Proof: We prove the result only for U-witnesses; the proof for L-witnesses is identical. The proof works by induction on the height of the tree. Consider a tree T rooted at x with children $x_1, x_2, \ldots, x_t$. Let T_i be the subtree rooted at x_i.

Suppose x is a MAX node. Assume for contradiction that the algorithm spends more than $f_U^T(c)$ in proving an upper bound of UB for tree T and yet there exists a U-witness W of cost at most c that proves that the value of T is at most UB. Since x is a MAX node, W consists of a collection of U-witnesses W_i of cost c_i for each subtree T_i with $c = \sum_{i=1}^{t} c_i$. Witness W_i proves that the value of the subtree T_i is at most UB. By the definition of $f_U^T(c)$, there exists k, with $1 \leq k \leq t$, such that the algorithm spends more than $f_U^{T_k}(c_k)$ on the subtree T_k. Consider the first time τ when the algorithm spends *more than* $f_U^{T_k}(c_k)$ on the subtree T_k. Since the algorithm always picks the subtree with the *maximum* current upper bound, it follows that the upper bound on the value of the subtree T_k just prior to the time τ is *strictly greater* than UB. Now the algorithm has spent more than $f_U^{T_k}(c_k)$ on T_k just after time τ (which is the *first* time when the algorithm proves an upper bound of UB on the value of T_k), and this implies that the algorithm spends more than $f_U^{T_k}(c_k)$ in proving an upper bound UB on the value of the subtree T_k. By the induction hypothesis, this is a contradiction since witness W_k has cost c_k and proves an upper bound of UB on the value of T_k.

Nest, suppose x is a MIN node. Assume for contradiction that the algorithm has spent more than $f_U^T(c) = \sum_{i=1}^{T} f_U^{T_i}(c)$ and yet there exists a U-witness W of cost c which proves that the value of T is at most UB. Since x is a MIN node, the U-witness W is a U-witness W_i of cost c for some subtree T_i. Say for concreteness, it is a U-witness W_t for T_t. By the induction hypothesis, the algorithm does not spend more than $f_U^{T_t}(c)$ on T_t. Hence the algorithm must spend more than $f_U^{T_i}(c)$ for some subtree T_i. Say for concreteness, $i = 1$ and the algorithm spends more than $f_U^{T_1}(c)$ on T_1. Consider the point where the algorithm chose the recommendation $R_1 = (L_1, c_{L_1})$ from T_1 and went above $f_U^{T_1}(c)$ on its expenditure on T_1, so that $\mathrm{Cost}_{x_1} + c_{L_1} > f_U^{T_1}(c)$. At this point, it rejected the recommendation $R_t = (L_t, c_{L_t})$ from T_t which we know satisfies $\mathrm{Cost}_{x_t} + c_{L_t} \leq f_U^{T_t}(c)$. But we then have $(f_U^{T_t})^{-1}(\mathrm{Cost}_{x_t} + c_{L_t}) \leq c < (f_U^{T_1})^{-1}(\mathrm{Cost}_{x_1} + c_{L_1})$. Thus the algorithm would have never chosen the recommendation from T_1 over that of T_t (here we are using the fact at levels where the parent is a MIN node, the algorithm uses the function f_U^T to decide whose recommendation to take), a contradiction. Hence there cannot be a U-witness of cost at most c as we supposed, and we are done. ■

To evaluate the MIN/MAX tree, we will run BALANCE$_U$ and BALANCE$_L$ in "parallel", (roughly) balancing the cost they have incurred at any point, till the upper bound found by BALANCE$_U$ and the lower bound found by BALANCE$_L$ match. We lose at most a factor two in the competitive ratio due to this.

Theorem 2.15 *For any MIN/MAX tree T and a cost vector $\mathbf{c}$, there is an efficient algorithm that evaluates T with a competitive ratio at most $2\gamma_{\mathbf{c}}(T)$.*

The above theorem also holds for a common generalization of threshold and MIN/MAX trees where the internal nodes are gates that return the t^{th} largest element for some t (the value of t could be different for different nodes). The details are straightforward given our analyses for threshold trees and MIN/MAX trees.

3 Ultra-uniform Prices

Given an AND/OR tree T with n leaves, we ask: how do we "fairly" price the leaves of T so that every on-line algorithm achieves the same competitive ratio? Such a price vector, if one exists, is called an *ultra-uniform* price vector. Intuitively, it means that the leaves are so evenly priced that at every stage it does not matter which leaf is queried next, from the point of view of the competitive ratio. (Intuitively, if a leaf is overpriced, an algorithm will defer reading it unless absolutely necessary; and similarly, if a leaf is underpriced it will be read right away.) It is far from clear why such a pricing, which appears to be a very strong requirement, should exist at all. We show in this section that such a pricing not only exists, but can also be found efficiently.

Theorem 3.1 *Given an AND/OR tree T with n leaves, one can find an ultra-uniform price vector for T in polynomial (in n) time.*

The proof of the theorem follows immediately from the following two lemmas.

Lemma 3.2 *Let $\mathbf{c}$ be a price vector for the leaves of an AND/OR tree such that, under the pricing $\mathbf{c}$, the costs of all 0-witnesses of T are equal and similarly the costs of all 1-witnesses of T are equal (the two costs for 0-witness and 1-witnesses need not be equal to each other). Then $\mathbf{c}$ is an ultra-uniform price vector for T.*

Proof: Let the the cost of all 0-witnesses of T (under the pricing $\mathbf{c}$) equal c_0, and let the cost of all 1-witnesses of T equal c_1 (c_0 need be equal to c_1). We wish to claim that $\mathbf{c}$ is an ultra-uniform price vector. To see this, note that tree functions are evasive and hence any algorithm can be forced to examine all the leaves, and the final value of the tree can be set to either 0 or 1 after the last leaf is read. If C is the total cost of all the leaves, any algorithm can thus be forced to have a competitive ratio of $C/\min(c_0, c_1)$. Moreover, any algorithm has a competitive ratio at most $C/\min(c_0, c_1)$, as the most an algorithm can spend is the total cost C of all the leaves, and the adversary incurs a cost at least $\min(c_0, c_1)$ for both 0-witnesses and 1-witnesses. Hence $\mathbf{c}$ is indeed an ultra-uniform price vector. ■

Lemma 3.3 *Let T be an AND/OR tree with n leaves. Then one can find, in time polynomial in n, a setting of prices for its leaves under which all 0-witnesses of T have the same cost, and all 1-witnesses of T have the same cost. Moreover such a price vector is unique up to scaling.*

Proof: We now describe how to construct prices that ensure the uniformity of the costs of 0-witnesses and 1-witnesses. It is easy to see that if this property holds for an AND/OR tree T, then it holds for all subtrees of T as well, and this shows that such a price vector is unique up to scaling. This also motivates the construction of prices in a bottom-up fashion, appropriately rescaling the prices as we move up the tree so that when we reach each intermediate node, the cost of all 0-witnesses and 1-witnesses of the subtree rooted at that node have the same cost.

We begin by setting the prices of all leaves to 1. As we move up the tree, we maintain, for each node v that has been visited, quantities $C_0[v]$ and $C_1[v]$ that represent the uniform costs of all 0-witnesses and 1-witnesses respectively in the subtree rooted at v *just after v was visited* (these quantities will change as we move further up the tree to v's ancestors). Now, suppose we move up the tree and reach an internal node u. Let us assume that u is an AND node; the proof for an OR node is similar. Let the children of u be $u_1, u_2, \ldots, u_k$ (these are all OR nodes since u is an AND node). Our goal is to construct, for the subtree of T rooted at u, a price vector that makes the costs of all 0-witness of T_u and those of all 1-witnesses of T_u the same, using price vectors $\vec{P}_i$ with a similar property for the subtrees T_{u_i}, $1 \leq i \leq k$. In order to make the cost of all 0-witnesses of T_u equal, we rescale the prices of the nodes in the T_{u_i}'s so that the cost of 0-witnesses of T_{u_i} and T_{u_j} for $1 \leq i < j \leq k$ are all the same. We can achieve this, for instance, by dividing the price vector $\vec{P}_i$ of the leaves in T_{u_i} by $C_0[u_i]$. After this rescaling, all 0-witnesses of T_u have cost 1, so we set $C_0[u] = 1$. A 1-witness of T_u is the union of 1-witnesses for $T_{u_1}, T_{u_2}, \ldots, T_{u_k}$; after the above rescaling all 1-witnesses in T_{u_i} have the same cost $C_1[u_i]/C_0[u_i]$, and hence all 1-witnesses of T_u have the same cost $C_1[u] \triangleq \sum_{i=1}^{k} C_1[u_i]/C_0[u_i]$.

When we reach the root of the tree T, we have a price vector with the required property. It is clear that this procedure can be implemented to run in $O(n^2)$ time, and the proof is complete. ■

4 Searching with Prices

In this section, we consider the problem of searching in a sorted array: suppose we are given a sorted array with n positions, and wish to determine whether it contains a particular number q. In the unit-price setting, when we simply wish to minimize the number of queries to array entries, binary search solves this problem in at most $\lceil \log_2 n \rceil$ queries.

Now suppose each array entry has a price, and we seek an algorithm of optimum competitive ratio. Here the cheapest "proof" of membership of q is simply a single query to an entry containing q; the cheapest proof of non-membership is a pair of queries to adjacent entries containing numbers less than and greater than q, respectively.

In Section 4.2, we provide an efficient algorithm for this problem that achieves the optimal competitive ratio with respect to any given cost vector. But first, we consider the associated *extremal problem*: which cost vector forces the largest competitive ratio? In Section 4.1, we give an algorithm achieving a competitive ratio of $\log_2 n + O(\sqrt{\log n} \log \log n)$ for any cost vector; this exceeds the competitive ratio for the uniform cost vector only by lower order terms. Whether the uniform cost vector is in fact extremal remains an interesting open question.

4.1 A near-optimal algorithm

We first present an algorithm for searching an n element array with competitive ratio bounded by $\log_2 n + O(\log_2^{2/3} n)$ for any cost vector on the elements of the array. Later, we will improve the algorithm to get a competitive ratio bounded by $\log_2 n + O(\sqrt{\log n} \log \log n)$. This proves that the unit price vector is *essentially* an extremal price vector for binary search, and also that the performance of our algorithm is at most off by lower order terms from the true competitive ratio.

The algorithm is motivated by two goals: **(1)** We do not examine *costly* elements until we have eliminated the possibility of the element q lying in an array location occupied by *cheaper* elements; and **(2)** to achieve a competitive ratio close to $\log_2 n$, we mimic binary search by attempting to halve the search interval with every comparison. Unfortunately, the two goals could be contradictory because the only way to halve the search interval might be to examine an expensive element.

High-level description of the algorithm. Our algorithm uses two parameters r and t. Initially costs are grouped geometrically by rounding costs up to the nearest multiple of r; the algorithm considers groups in increasing order of cost. We normalize costs so that the lowest cost is 1.[7] Let group j consist of all elements with cost between r^{j-1} and r^j. The algorithm maintains a search interval I, which is the set of possible (contiguous) locations where q could lie, and splits I into three (contiguous) intervals L, M, R where the left and right intervals L, R do not contain any element of (the current) group j and the middle interval M, referred to as the *restricted interval*, which begins and ends with an element of group j. The algorithm maintains the property that I does not contain any elements of groups $(j-1)$ or lower. We repeatedly compare q with the group j element that is closest to the middle of the restricted interval M. Such comparisons are called *regular* comparisons and later we prove that each such comparison is guaranteed to halve the size of the restricted interval. This certainly makes progress as long as the element q lies within the restricted interval. However, if q does not belong to the current group j, at some point, q could fall outside the restricted interval for group j. In such a case, we do not want to spend too much on querying group j elements. To handle this possibility, after every t regular comparisons of q with group j elements, we perform a *boundary* comparison by querying one of the extreme group j elements. This checks if q lies outside the restricted interval. If the current search interval I does not contain any element of the current group j, we move on to group $j+1$, and continue the algorithm.

We now give a formal description of the algorithm. In the algorithm, we use the notation $I \circ J$ to denote the concatenation of the intervals I and J. Also if interval J consists of a single element x, we denote $I \circ J$ by just $I \circ x$.

Algorithm Search

1. $I \leftarrow [1, n]$, $j \leftarrow 1$, $\mathit{left\_cnt} \leftarrow 0$, $\mathit{right\_cnt} \leftarrow 0$.

2. While I does not contain an element of group j
   $j \leftarrow j + 1$; $\mathit{left\_cnt} \leftarrow 0$; $\mathit{right\_cnt} \leftarrow 0$.
   endWhile

3. If $\mathit{left\_cnt} = t$,
   $\mathit{left\_cnt} \leftarrow 0$.
   Let x be the leftmost element of group j in I.
   $\mathit{type} \leftarrow \mathit{BOUNDARY}$. Jump to Step 6.

4. If $\mathit{right\_cnt} = t$,
   $\mathit{right\_cnt} \leftarrow 0$.
   Let x be the rightmost element of group j in I.
   $\mathit{type} \leftarrow \mathit{BOUNDARY}$. Jump to Step 6.

[7] We assume without loss of generality that all costs are non-zero since the comparisons involving zero cost elements can be performed right away at the beginning. Once the costs are all non-zero, we can normalize them so that the minimum cost equals 1.

5. Decompose I as $I = L \circ M \circ R$ into three intervals L, M, R such that the left and right intervals L and R do not contain any element of group j, while the middle interval M starts and ends with an element from group j. M is thus the current *restricted interval.*

   Let x be the element in group j that is closest to the middle of M, breaking ties arbitrarily. $type \leftarrow REGULAR$.

6. Let I_L and I_R be subintervals of I such that $I = I_L \circ x \circ I_R$.

7. Compare q to x.

8. If $x = q$, return **PRESENT**
   else if $q < x$,
   $\quad I \leftarrow I_L$,
   $\quad$ if $type = REGULAR$
   $\quad\quad left\_cnt \leftarrow left\_cnt + 1$; $right\_cnt \leftarrow 0$.
   else if $q > x$,
   $\quad I \leftarrow I_R$,
   $\quad$ if $type = REGULAR$
   $\quad\quad right\_cnt \leftarrow right\_cnt + 1$; $left\_cnt \leftarrow 0$.

9. If I is empty, return **NOT PRESENT**

10. Goto step 2.

Competitive analysis of the algorithm. The algorithm maintains an interval I of the array in which the element q being searched for must lie. It compares q to some element x in the current interval. Depending on the result of the comparison, the algorithm restricts its search in the subinterval of I to the left of x (if $q < x$) or to the right of x (if $q > x$). This procedure is thus guaranteed to find q if indeed it is present in the array.

Recall that we distinguish between two kinds of comparisons made by the algorithm. If the element x compared to is chosen in Steps 3 or 4, such a comparison is called a *boundary* comparison. On the other hand, if the element x compared to is chosen in Step 5 such a comparison is called a *regular* comparison. The following lemma shows that the algorithm makes progress when it performs regular comparisons.

Lemma 4.1 *Each regular comparison performed on group j reduces the length of the restricted interval by a factor of at least 2.*

Proof: Suppose I is the current interval. Let $I = L \circ M \circ R$ where L, M and R are the intervals obtained in Step 5. Suppose x is the element that is chosen to compare with. By choice, x is the element closest to the middle of M. Let $M = M_L \circ x \circ M_R$. Without loss of generality, assume that $|M_L| \le |M_R|$. Hence, $|M_L| \le (|M| - 1)/2$. Further, let $M_R = L' \circ M'$ where M' is the smallest interval containing all the elements of group j in M_R. Note that $M = M_L \circ x \circ L' \circ M'$. By the choice of x, $|M'| \le |M_L| + 1$. We claim that $|M'| \le \frac{1}{2}|M|$. To prove this we consider two cases: (a) $|M_L| < (|M| - 1)/2$; in this case $|M'| \le |M_L| + 1 \le \frac{1}{2}|M|$; and (b) $|M_L| = (|M| - 1)/2$; in this case x is exactly the middle element of M. Thus $|M_R| = (|M| - 1)/2$ and so, $|M'| \le |M_R| < \frac{1}{2}|M|$.

If $q < x$, the restricted interval is a subinterval of M_L. Suppose $q > x$. In this case, the restricted interval is M'. In both cases, the size of the restricted interval drops by a factor of at least 2. ■

Let n_j be the length of the search interval I at the first time that the algorithm considers group j. If m is the last group examined, define n_{m+1} to be 1. Let c_j be the total number of comparisons performed with elements of group j.

Lemma 4.2

$$c_j \leq \left(1 + \frac{1}{t}\right)\left(\log_2\left(\frac{n_j}{n_{j+1}}\right) + 1\right) + t + 1$$

Proof: Let I_j be the search interval at the first time that the algorithm considers elements of group j. We want to bound the number of comparisons made by our algorithm when it considers group j elements. (For the sake of the proof, define I_{m+1} to consist of the single element q, even though q is a member of group m.) We will separately bound:

1. The number of comparisons made during the period when I_{j+1} is part of the restricted interval. (If $j = m$, this is the only case we need to consider. Note also that it is possible that I_{j+1} is not part of the restricted interval of I_j to begin with, and in this case there are no comparisons in this phase.)

2. The number of comparisons made after I_{j+1} is no longer part of the restricted interval.

First, consider the number of comparison steps performed up to the point when I_{j+1} is cut off from the restricted interval. If I_{j+1} is not part of the restricted interval of I_j to begin with, then there are clearly no comparisons performed in this phase, so assume that I_{j+1} belongs to the restricted interval of I_j to begin with. Let e_1 be the length of the restricted interval at the first time that group j is considered, and let e_2 be the minimal length of the restricted interval while it contains I_{j+1}. Clearly, $e_1 \leq |I_j| = n_j$ and $e_2 \geq |I_{j+1}| = n_{j+1}$. Since each regular comparison reduces the length of the restricted interval by a factor of at least 2 by Lemma 4.1, the number of regular comparisons while the restricted interval contains I_{j+1} is compared is at most $\lceil \log_2(e_1/e_2) \rceil \leq \log_2(n_j/n_{j+1}) + 1$. Further, the number of boundary comparisons performed during this time is at most $1/t$ times the number of regular comparisons, since each boundary comparison can be charged to t regular comparisons. Thus the total number of comparisons until I_{j+1} is cut off from the restricted interval is at most most

$$\left(1 + \frac{1}{t}\right) \cdot \left(\log_2\left(\frac{n_j}{n_{j+1}}\right) + 1\right).$$

Now, I_{j+1} can be cut off from the restricted interval only by means of a comparison with a group j element immediately to the left or right of I_{j+1} (or I_{j+1} could have started off being outside the the restricted interval on the right or left side). Without loss of generality, suppose that just after I_{j+1} has been cut off from the restricted interval, the search interval is of the form $I_{j+1} \circ x_r \circ I'$. (Since I_{j+1} does not contain any elements of group j, it is no longer part of the restricted interval.) Since the search gets narrowed down to I_{j+1} later, it follows that for all group j elements x' compared to from this point on, $q < x'$. But there can be at most $t+1$ such comparisons. If within t more regular comparisons the search has not already been narrowed down to I_{j+1}, then element x_r will be picked in the next iteration in Step 3 and compared with q. That will narrow down the search interval to I_{j+1} in at most $t+1$ steps.

Adding the two bounds, we get the bound in the statement of the lemma. ∎

Theorem 4.3 *If we let $r = 1 + 1/\log_2^{1/3} n$ and $t = \log_2^{1/3} n$, the competitive ratio of the algorithm is bounded by $\log_2 n + O(\log_2^{2/3} n)$.*

Proof: Recall group m is the last group examined by the algorithm. Then the cost of the algorithm is at most

$$\begin{aligned}\sum_{j=1}^{m} r^j \cdot c_j &\leq \sum_{j=1}^{m} r^j \left(\left(1+\frac{1}{t}\right)\log_2\left(\frac{n_j}{n_{j+1}}\right) + t + \frac{1}{t} + 2\right) \\ &= \left(1+\frac{1}{t}\right)\sum_{j=1}^{m} r^j \cdot \log_2\left(\frac{n_j}{n_{j+1}}\right) + (t + \frac{1}{t} + 2)\sum_{j=1}^{m} r^j \\ &\leq \left(1+\frac{1}{t}\right) r^m \log_2 n + (t+\frac{1}{t}+2)\frac{r^{m+1}}{r-1}\end{aligned}$$

The optimal proof has cost at least r^{m-1}. Hence the competitive ratio of the algorithm is bounded by

$$\left(1+\frac{1}{t}\right) r \log_2 n + (t + \frac{1}{t} + 2)\frac{r^2}{r-1}.$$

Setting $r = 1 + 1/\log_2^{1/3} n$ and $t = \log_2^{1/3} n$, it is straightforward to check that we get the claimed bound on the competitive ratio. ∎

4.1.1 Improved algorithm

We can improve the competitive ratio by modifying the above algorithm slightly. The idea is to change the way in which boundary comparisons are performed. From the proof of Lemma 4.2, the number of boundary comparisons made with group j elements in the algorithm described above is roughly $t + \frac{1}{t}\log_2\left(\frac{n_j}{n_{j+1}}\right)$ (roughly t comparisons after I_{j+1} goes outside the restricted interval, and about $\frac{1}{t}\log_2\left(\frac{n_j}{n_{j+1}}\right)$ prior to that). The improvement comes from balancing the two terms in this expression. The modified algorithm does not use the parameter t. Instead, it keeps track of the total number of regular comparisons performed so far for the current group. A boundary comparison is performed roughly every time the total number of regular comparisons equals a perfect square.

Modified Algorithm. The improved algorithm is the same as Algorithm Search, except for the following modifications:

1. The conditions in Steps 3 and 4 are replaced by "If *left_cnt* is a perfect square" and "If *right_cnt* is a perfect square," respectively.

2. Except for those in Steps 1 and 2, all other "*left_cnt* $\leftarrow 0$" and "*right_cnt* $\leftarrow 0$" commands are discarded.

These two modifications have the effect of counting the number of left and right regular comparisons for each group, and performing the respective boundary comparisons whenever the number of corresponding regular comparisons has reached a perfect square.

Furthermore, we pick the "grouping" parameter r differently. Specifically, we set $r = 1 + \frac{2}{\sqrt{\log_2 n}}$.

Improved Competitive Analysis. Define $x_j = \log_2(n_j/n_{j+1})$. Note that

$$\sum_{j=1}^{m} x_j = \log_2 n \tag{8}$$

We now state and prove the analog of Lemma 4.2 for the modified algorithm.

Lemma 4.4 *Let c_j be the number of comparisons performed by the modified algorithm with group j elements. Then*

$$c_j \leq x_j + 4\sqrt{x_j} + 7 \ . \tag{9}$$

Proof: We first make the following observations:

1. For group j, if k regular comparisons are made, at most $2\lceil\sqrt{k}\rfloor$ boundary comparisons could be performed for the group.
2. For group j, suppose that I_{j+1} is cut off from the restricted interval after k regular comparisons have been made. After this point, all regular comparisons with group j elements will point in the same direction. Clearly within at most $2\lceil\sqrt{k}\rceil + 1$ such comparisons, the count of total number of regular comparisons will reach a perfect square, and a boundary comparison will be performed that will narrow the search interval to I_{j+1}.

These observations, using arguments similar to those used in the proof of Lemma 4.2, yield:

$$\begin{aligned} c_j &\leq \left(\lceil x_j\rceil + 2\sqrt{\lceil x_j\rceil}\right) + 2\sqrt{\lceil x_j\rceil} + 2 \\ &\leq x_j + 4\sqrt{x_j} + 7, \end{aligned}$$

where in the last step we used the facts that $\lceil x_j\rceil \leq x_j + 1$ and $\sqrt{\lceil x_j\rceil} \leq \sqrt{x_j} + 1$. ■

By Lemma 4.4, the total cost that the algorithm incurs is at most:

$$\sum_{j=1}^{m} r^j \left(x_j + 4\sqrt{x_j} + 7\right) \tag{10}$$

We bound the cost of the algorithm by computing the maximum of the quantity (10) subject to Constraint (8). Using standard arguments from the theory of Lagrange multipliers, we see that at the global maximum of Expression 10, the partial derivatives with respect to each x_j must be equal to some constant c. Thus, we have that for every j:

$$r^j \left(1 + \frac{2}{\sqrt{x_j}}\right) = c$$

and thus,

$$x_j = \frac{4}{\left(\frac{c}{r^j} - 1\right)^2}$$

Note that $x_m \geq 0$ implies that $c > r^m$. Thus, we can write $c = r^{m+\delta}$ for some value of $\delta > 0$. Recall that $r = 1 + \frac{2}{\sqrt{\log_2 n}}$.

Now, using these maximizing values for x_j, let us bound the total cost that the algorithm can incur:

$$\begin{aligned}
\sum_{j=1}^{m} r^j \cdot c_j &\leq \sum_{j=1}^{m} r^j \left(\frac{4}{(r^{m-j+\delta} - 1)^2} + 4\frac{2}{(r^{m-j+\delta} - 1)} + 6 \right) \\
&\leq r^m \sum_{i=0}^{m-1} \frac{1}{r^i} \frac{4}{(r^{i+\delta} - 1)^2} + 4r^m \sum_{i=0}^{m-1} \frac{1}{r^i} \frac{2}{(r^{i+\delta} - 1)} + 7\frac{1}{r-1}
\end{aligned}$$

We bound the contribution of each term to the competitive ratio separately. Recall that the cost of the minimal witness is at least r^{m-1}.

The contribution of the first term is at most:

$$\begin{aligned}
\frac{1}{r^{m-1}} \cdot r^m \sum_{i=0}^{m-1} \frac{1}{r^i} \frac{4}{(r^{i+\delta} - 1)^2} &\leq r \cdot \sum_{t=0}^{m-1} \frac{4}{(r^{i+\delta} - 1)^2} \\
&= r \log_2 n \\
&= \log_2 n + 2\sqrt{\log_2 n}
\end{aligned}$$

The contribution of the second term is at most:

$$\frac{1}{r^{m-1}} \cdot 4r^m \sum_{i=0}^{m-1} \frac{1}{r^i} \frac{2}{(r^{i+\delta} - 1)} \leq 4r \sum_{i=0}^{\sqrt{\log_2 n}} \frac{1}{r^i} \frac{2}{(r^{i+\delta} - 1)} + 4r \sum_{i > \sqrt{\log_2 n}} \frac{1}{r^i} \frac{2}{(r^{i+\delta} - 1)}$$

We consider these two parts separately. Observe that for $i > \sqrt{\log_2 n}$, by the definition of $r = 1 + 2/\sqrt{\log_2 n}$ we have that $r^i > 3$, and hence:

$$2/(r^{i+\delta} - 1) < 1$$

Thus,

$$\begin{aligned}
4r \sum_{i > \sqrt{\log_2 n}} \frac{1}{r^i} \frac{2}{(r^{i+\delta} - 1)} &\leq 4r \sum_{i > \sqrt{\log_2 n}} \frac{1}{r^i} \\
&\leq \frac{4r}{1 - 1/r} \\
&= O(\sqrt{\log_2 n})
\end{aligned}$$

On the other hand, we first observe:

$$r^{i+\delta} - 1 \geq \frac{2(i+\delta)}{\sqrt{\log_2 n}}$$

Using this, we bound the contribution of the second term for $i < \sqrt{\log_2 n}$:

$$\begin{aligned}
4r \sum_{i=0}^{\sqrt{\log_2 n}} \frac{1}{r^i} \frac{2}{(r^{i+\delta}-1)} &\leq 4r \sum_{i=0}^{\sqrt{\log_2 n}} \frac{2}{(r^{i+\delta}-1)} \\
&\leq 4r \sum_{i=0}^{\sqrt{\log_2 n}} \frac{\sqrt{\log_2 n}}{i+\delta} \\
&= 4r\sqrt{\log_2 n} \cdot O(\log\log n) \\
&= O(\sqrt{\log_2 n}\log\log n)
\end{aligned}$$

Finally, the contribution of the last term is at most

$$\frac{7}{r-1} = O(\sqrt{\log_2 n})$$

Adding these bounds together, we conclude that our improved algorithm achieves a competitive ratio of $\log_2 n + O(\sqrt{\log_2 n}\log\log n)$.

Theorem 4.5 *There is an algorithm for searching in a sorted array of n elements that achieves a competitive ratio of $\log_2 n + O(\sqrt{\log n}\log\log n)$ for any cost vector.*

4.2 Optimal search for a given cost vector

We now present a dynamic programming algorithm to compute the optimal algorithm for searching a sorted array of priced elements. Straightforward dynamic programming would entail considering all $O(n^2)$ subintervals, and computing the best competitive ratio possible for each subinterval. This, however, fails, as can be seen from the following illustration. Suppose on some particular subinterval I of interval J, the adversary could force any algorithm to pay total cost at least 2 to find an element of cost 1, or pay total cost at least 60 to find an element of cost 20. A strict competitive ratio analysis would lead us to believe that the adversary should always force the algorithm to pay at least 60 to find an element of cost 20. However, if on the larger interval J, it was the case that the adversary could force any algorithm to pay cost at least 2 before reducing the search problem to I, then clearly when the search focuses on I, the adversary should force the algorithm to pay 2 more and find the element of cost 1, as this would lead to an overall competitive ratio of 4 (as opposed to $(60+2)/20$).

This suggests the following algorithm, which does work: For every subinterval I, and every x, we will first compute a lower bound $f(I,x)$ for the competitive ratio that any deterministic algorithm can achieve on I, *given that the algorithm has already spent* x. For any element $a \in I$, let c_a denote the cost of examining a. For any singleton interval $I = \{a\}$, clearly $f(\{a\},x) = (x+c_a)/c_a$ is an exact bound on the competitive ratio. Also, for an empty interval I, we let $f(I,x) = 0$ for all x. Now for all larger intervals I, we define:

$$f(I=[a,b],x) = \min_{i \in I}\Big[\max\{ \begin{array}{l} f([a,(i-1)], x+c_i), \\ (x+c_i)/c_i, \\ f([(i+1),b], x+c_i)\}\Big] \end{array} \qquad (11)$$

A simple inductive argument shows that this gives the desired lower bound, as the algorithm has choice over which i to examine, and the adversary can choose to either respond that the element being searched for is smaller than, equal to, or greater than element i. Furthermore, we can efficiently pre-compute a table of these lower bounds for every subinterval and every value for x up to the sum of all costs. This then yields an optimal algorithm for performing the binary search, as the optimal first move for interval I having already spent x is determined by the minimizing choice of i in the computation of $f(I, x)$.

5 A competitive algorithm for finding the maximum

As discussed in the introduction, one can study several fundamental algorithmic problems like sorting, searching and selection in a framework where the comparisons have varying costs. We studied one such problem, namely binary search, in the previous section. It turns out that problems like sorting and median finding become extremely challenging in this "priced" setting. In this section, we consider the problem of competitively finding the maximum of n elements when the comparisons have varying costs. Our result here is stated in the following theorem.

Theorem 5.1 *Let $n \geq 2$. There is an efficient algorithm with competitive ratio $(2n-3)$ for finding the maximum of n elements, for* any *set of costs for the comparisons between pairs of elements.*

Proof: We give the following strategy which we will prove has competitive ratio $(2n-3)$ for every cost vector. Let $S = \{x_1, x_2, \ldots, x_n\}$ be a set of $n \geq 2$ distinct elements where the goal is to find the maximum element in S.

1. Initially $T = S$ (T is the set of all potential maxima, i.e. those elements that have not yet been ruled out from being the maximum);
2. While T has more than one element:
   (a) For each element of T, determine the cheapest comparison (breaking ties arbitrarily) involving that element, which has not been performed so far. (Note that this comparison could be with an element in $S \setminus T$; even though such elements cannot themselves be the maximum, comparisons with them can still be useful in ruling out elements in T from being the maximum.)
   This gives a *multiset* $\mathcal{M}$ of $|T|$ comparisons, one for each element of T.
   (b) Perform all comparisons in the multiset $\mathcal{M}$ chosen in Step (a) above, *except the most expensive one* among them (ties are broken arbitrarily).
   (c) Remove from T all those elements which ended up being the smaller element in their comparison in Step (b) (and thus cannot be the maximum element in S).
3. Output the unique element still left in T as the maximum.

(Note that if T has only two elements a, b and both the elements have the same comparison, namely the one that compares a and b, as their cheapest comparison, then Step 2(b) *will* indeed perform this comparison. This is due to the fact that we treat the comparisons chosen in Step 2(a) as a *multiset.*)

It is clear that the above algorithm always terminates and outputs the correct maximum. We now analyze the performance of the algorithm. Let x_k be the maximum element in S. Note that a cheapest "witness" W to x_k being the maximum is a rooted directed tree (with x_k at the root) with edges directed away from the root (an out-going edge from x_j to x_ℓ means that $x_j > x_\ell$). Each x_i, $i \neq k$, has in-degree exactly 1 in W. Denote by $\mathcal{C}_i$ the comparison corresponding to the unique edge going into x_i in W. We prove that the while loop in the above strategy is executed at most $(2n-3)$ times and in each iteration the algorithm spends an amount which is at most the cost of W. Together these will imply that the competitive ratio of the algorithm is at most $(2n-3)$.

At every stage of the algorithm define the "out-degree" of an element x in T to be the number of comparisons involving x that have not been (explicitly) performed yet.[8] In each iteration we perform comparisons involving all elements of T except one, and hence the sum of the largest and second-largest out-degrees in T goes down by at least 1 after each iteration. (This is because of the simple fact that if we have m numbers $a_1, \ldots, a_m$ and we decrease all but one of them by 1, then the sum of the largest and second-largest a_i's goes down by at least 1.) Since this sum is $2(n-1)$ initially, after at most $(2n-3)$ iterations of the while loop, T will have only one element, and we would have thus found the maximum. Thus there are at most $(2n-3)$ iterations of the while loop.

Now consider a fixed iteration of the while loop that begins with a specific set T of potential maxima. For each $x_i \in T$, let $\mathcal{C}'_i$ be the cheapest comparison involving x_i that has not been performed yet and which is chosen in Step (a). All comparisons involving elements of T made in previous iterations must have been with smaller elements (otherwise the element would not be in T in the first place). Hence for each i such that $x_i \in T \setminus \{x_k\}$, the comparison $\mathcal{C}_i$ (from the witness W) has not been performed yet. Therefore the cheapest comparison $\mathcal{C}'_i$ chosen for x_i has a cost at most that of $\mathcal{C}_i$. Now we use the fact that we make all comparisons in the set $\{\mathcal{C}'_i : x_i \in T\}$ *except the most expensive one*, and hence the total cost of comparisons performed in this iteration is at most

$$\sum_{i \,:\, x_i \in T \setminus \{x_k\}} \mathrm{cost}(\mathcal{C}'_i) \leq \sum_{i \,:\, x_i \in T \setminus \{x_k\}} \mathrm{cost}(\mathcal{C}_i) \leq \mathrm{cost}(W) \ .$$

Together with the bound on the number of iterations of the while loop, this completes the proof that the competitive ratio of our algorithm is at most $(2n-3)$. ■

Remarks on recent related work: Recently, Hartline *et al* [7] have shown that the above analysis is tight in the sense that there are examples where the above algorithm has a competitive ratio $2n - O(1)$. They also prove that modifying the algorithm so it considers, at any iteration, *only* those comparisons that *cannot* be inferred by transitivity, improves its competitive ratio to $(n-1)$. Finally, they also prove a lower bound of $(n-2)$ on the competitive ratio for *every* deterministic algorithm, and therefore the competitive ratio of $(n-1)$ achieved by their algorithm is essentially the best possible. The task of finding a good algorithm to competitively find the maximum for a *fixed* cost vector appears to be significantly harder, and is an interesting direction for future work.

[8] Some of these comparisons could be unnecessary since we might already know their result by transitivity, but the algorithm will be $O(n)$-competitive even when it performs such redundant comparisons. Hence we will not be concerned about eliminating such obviously useless comparisons. However, doing so leads to roughly a factor 2 improvement in the competitive ratio [7] (see the remarks following the proof of Theorem 5.1).

Acknowledgments

We thank Ravi Kumar for useful discussions and for suggesting the generalization to threshold trees. We thank several students at the University of Washington including Jason Hartline, Ed Hong, Alex Mohr and Emily Rocke for helpful comments on a previous write-up of this work.

References

[1] A. Blum, P. Chalasani, D. Coppersmith, W. Pulleyblank, P. Raghavan and M. Sudan, "The minimum latency problem," *Proceedings of the 26th ACM Symposium on the Theory of Computing*, pp. 163–171, 1994.

[2] A. Borodin, R. El-Yaniv, *On-Line Computation and Competitive Analysis*, Cambridge University Press, 1998.

[3] B. Bollobas, *Extremal Graph Theory*, Academic Press, 1978.

[4] Clickshare Service Corp., www.clickshare.com.

[5] O. Etzioni, S. Hanks, T. Jiang, R.M. Karp, O. Madani, O. Waarts, "Efficient information gathering on the Internet," *Proc. IEEE FOCS*, 1996.

[6] H. Garcia-Molina, S. Ketchpel, N. Shivakumar, "Safeguarding and Charging for Information on the Internet," *Proc. Intl. Conf. on Data Engineering*, 1998.

[7] J. Hartline, E. Hong, A. Mohr, E. Rocke and K. Yasuhara. Personal Communication, November 2000.

[8] R. Heiman, A. Wigderson, "Randomized vs. Deterministic Decision Tree Complexity for Read–Once Boolean Functions," *Complexity Theory*, to appear.

[9] J. Komlós, Y. Ma, E. Szemerédi, "Matching nuts and bolts in $O(n \log n)$ time," *Proc. ACM-SIAM SODA*, 1996.

[10] E. Koutsoupias, C. Papadimitriou, M. Yannakakis, "Searching a fixed graph," *Proc. Intl. Conf. on Automata, Languages, and Programming* 1996.

[11] D. Kreps, *A Course in Micro-Economic Theory*, Princeton University Press, 1990.

[12] R. Motwani, P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.

[13] Pricing Economic Access to Knowledge (PEAK) Home Page, http://www.lib.umich.edu/libhome/peak/papers.html.

[14] S. Sairamesh, C. Nikolaou, D. F. Ferguson and Y. Yemini. Economic Framework for Pricing and Charging in Digital Libraries. D-Lib Magazine, February 1996.

[15] M. Saks, A. Wigderson, "Probabilistic Boolean decision trees and the complexity of evaluating game trees," *Proc. IEEE FOCS*, 1986.

[16] M. Snir, "Lower bounds on probabilistic linear decision trees," *Theoretical Computer Science*, Vol. 38, pp. 69-82, 1985.

[17] D. Tygar, "NetBill: An Internet Commerce System Optimized for Network-Delivered Systems," *IEEE Personal Communications*, Vol. 2, pp. 20-25, 1995.

[18] "What's the Value of Digital Information?", panel at *ICEE Conf. on Electronic Commerce: Foundations for the Future*, 1999.

[19] Y. Zhang, "On the optimality of randomized alpha-beta search," *SIAM Journal on Computing*, Vol. 24, pp. 138-147, 1995.