Query Strategies for Priced Information

(Extended Abstract)

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

October, 1999

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, constructing a set of prices for any AND/OR tree that satisfies a very strong type of equilibrium property.

1 Introduction

The potential of *priced information sources* [12, 13] 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

[*]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, Cambridge, MA 02139. Email: `{venkat,amits}@theory.lcs.mit.edu`. Research supported by an IBM Graduate Fellowship and DOD Fellowship, respectively. Most of this work was done while the authors were visiting 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.

electronic commerce [4, 6, 16, 17]. 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. [9]. 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. It 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 Boolean 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 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 performance ratio over all settings σ of the variables.

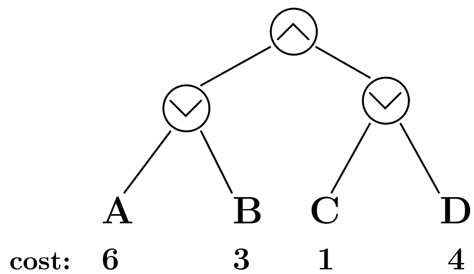

Figure 1: A Boolean function with priced inputs

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 ratio 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* $\gamma_{\mathbf{c}}^{\mathcal{A}}(f) \stackrel{\text{def}}{=} \max_\sigma \mathbf{c}_{\mathcal{A}}(\sigma)/\mathbf{c}(\sigma)$. The best possible competitive ratio for any algorithm, then, is

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

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 rooted (w.l.o.g.) at an AND gate, with each leaf corresponding to a distinct variable, and with each root-to-leaf path 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, 9] — problems for which polynomial-time algorithms are not known. It is not at all obvious that the optimal evaluation algorithm for a Boolean 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 Boolean 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 Boolean trees, and focus on three related sets of issues.

(1) Tractability of optimal algorithms. We show that for every Boolean 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 Boolean 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 Boolean 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 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 a Boolean tree T on n inputs can *simulate* an AND gate of size k if by fixing the values of some $(n-k)$ inputs, 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 [7, 11, 14, 15] 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 [10]. 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 Boolean 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.

Sorting, Searching and Selection. 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. It is possible to formalize this problem in terms of a function f of the type described above, imposing certain constraints on the sets of inputs that are allowed; we omit the details here.

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. 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 preliminary results and open questions. 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 notoriously difficult problem of "sorting nuts and bolts." [8]

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 becomes much more challenging in high dimensions; we have developed one approach that is based on a VC-dimension analysis, and identified a number of interesting open questions. 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. We will report our progress on this problem in the full version of the paper.

2 Tree Functions

We first consider functions computed by Boolean 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 a Boolean 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).

2.1 Efficient algorithm achieving $\gamma(T)$

We first investigate the competitive ratio $\gamma(T)$ for any Boolean tree T (recall the definition of Equation (1)), where the structure of T is fixed, but leaf prices vary. We propose the following simple lower bound on $\gamma(T)$. For any Boolean 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. (Such a k is also the size of the largest minterm in boolean function computed by T. 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.) 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 (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, $\max\{k, \ell\}$ is a lower bound on $\gamma(T)$.[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 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 on (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 $(L_1, c_{L_1}), \ldots, (L_t, c_{L_t})$. 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 upward the recommendation (L_i, c_{L_i}) 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 2.1 *For any Boolean 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.*

[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.

Proof Sketch: 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 more than kc, and yet 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 $(\text{Cost}_{x_j} + c_{L_j}) > k_j c$; on the other hand, since there is a 0-witness of cost c rooted at x_i that has not been found, by induction, the recommendation (L_i, c_{L_i}) from x_i must be such that $(\text{Cost}_{x_i} + c_{L_i}) \leq k_i c$. 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, 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, 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 2.2 *Let k and ℓ be as in Lemma 2.1. Then, $\gamma(T) = \max\{k, \ell\}$, and* WEAKBALANCE *runs in polynomial time and achieves a competitive ratio of $\gamma(T)$.*

Corollary 2.3 *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 $\gamma(T)$.*

Corollary 2.4 *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.*

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 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 boolean trees this computation is actually easy, and this gives a simple polynomial-time $(2\max\{k, l\})$-competitive algorithm for boolean tree evaluation, for any cost vector. 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 our algorithm Balance which, 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 Boolean 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 a Boolean 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 which arise now depending upon whether r_S is an And node or an Or node.

  (1) r_S is an And node: Now, 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, suggesting the bound 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, suggesting the other bound 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.

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 Boolean tree T. Hence all the functions above have a *compact* (of complexity polynomial in the number of leaves and the sum of the costs) representation as a table of values and this representation can be computed efficiently: It is clear that the operations of Equations (2) and (4) can be performed efficiently. For Equations (3) and (5), it is not difficult to see that by representing all functions as a table of values, it is possible to calculate them in time polynomial in the sum of the costs of the leaves.

Later, in the specification of our algorithm, we will also be referring to the inverse $(f_0^T)^{-1}$ 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 $\min\{x : f(x) = y\}$. In words, $f^{-1}(y)$ is the minimum element in the inverses image of y under f. 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 2.5 *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 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 entire 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 at least $f_0^{S_j}(c)$, ensure (using part (2) of the inductive hypothesis) that 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). Hence 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 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 $i \in [1..t]$. 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. 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 which in turn will imply a poor competitive ratio. 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 any Boolean 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 2.5, note that this immediately implies that BALANCE achieves the optimum competitive ratio possible for

Algorithm BALANCE:

Input: A Boolean 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 $\text{Cost}_x = 0$ 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 $(f_0^{x_q})^{-1}(c_{L_q} + \text{Cost}_{x_q})$.
   (ii) If x is an OR node, choose the child x_q with the minimum value of $(f_1^{x_q})^{-1}(c_{L_q} + \text{Cost}_{x_q})$.
   (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 $\text{Cost}_y = \text{Cost}_y + c_L$.

endWhile

Output the value of the tree T.

Figure 2: The BALANCE Algorithm.

any deterministic algorithm; indeed any deterministic algorithm has a competitive ratio of at least $\max\Big[\max_x\{f_0^T(x)/x\},\ \max_x\{f_1^T(x)/x\}\Big]$, and BALANCE achieves this competitive ratio.

Theorem 2.6 *If* BALANCE *when running on* $(T, \mathbf{c})$ *spends an amount which is greater than* $f_0^T(c)$ *(respectively* $f_1^T(c)$*), then there exists* **no** 0*-witness (respectively* 1*-witness) for* T *which has cost at most* c*. Or, equivalently, 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 $x_1, x_2, \ldots, x_t$ with subtree T_i rooted at x_i for $1 \leq i \leq t$.

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 a k, $1 \leq k \leq t$, 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_i. 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.

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. 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). 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 2.7 *For any boolean tree* T *and cost vector* $\mathbf{c}$*,* BALANCE *achieves a competitive ratio of* $\gamma_{\mathbf{c}}(T)$*.*

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 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 Boolean trees, and as a result we need to run two algorithms in parallel (balancing the costs they incur) one of which uses the function f_1 and the other f_0 for the balancing criterion; this incurs a factor 2 loss in the competitive ratio of the algorithm. We next specify the lower bound functions for general threshold gates. The details of the proof on how and why modified Balance works for threshold trees are similar to those given for Boolean trees and are omitted in this version.

Lower Bound Function for Threshold Gates: Suppose a threshold tree T 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]

$$f_1^T(x) = \max_I \Big[\max_{\substack{x_1,\ldots,x_p: \\ \sum_j x_j = x}} \Big\{ f_1^{S_{i_1}}(x_1) + \cdots + f_1^{S_{i_p}}(x_p) + \textstyle\sum_{i \notin I} f_1^{S_i}(\max x_j) \Big\} \Big] \tag{6}$$

Observe that this equation is equivalent to:

$$f_1^T(x) = \max_I \Big[\max_{\substack{x_1,\ldots,x_{p-1}: \\ \sum_j x_j \le x}} \Big\{ f_1^{S_{i_1}}(x_1) + \cdots + f_1^{S_{i_{p-1}}}(x_{p-1}) + \textstyle\sum_{i \notin I} f_1^{S_i}(x - \sum_j x_j) \Big\} \Big] \tag{7}$$

The latter equation gives insight into the lower bound argument, while the former corresponds to the argument for optimality of the modified Balance algorithm. The equation for f_0^S is 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]

Theorem 2.8 *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 twice $\gamma_{\mathbf{c}}(T)$.*

2.4 Game Trees

We can in fact generalize Balance to competitively evaluate *game trees* (also called MIN/MAX trees). A game 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.

[5] In Equations (6) and (7), the first max operators are taken over choices of $I = \{i_1, i_2, \ldots, i_p\} \subseteq [t]$. In Equation (6), the second max operator is taken only over choices of $x_1, \ldots, x_p$ such that there can 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. Similarly, in Equation (7), the second max operator is taken only over choices of $x_1, \ldots, x_{p-1}$ such that: (A) there can exist 1-witnesses in $S_{i_1}, \ldots, S_{i_{(p-1)}}$ of cost at most $x_1, \ldots, x_{p-1}$, respectively; (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.

[6] For our algorithm, it is important that these functions f_0^T and f_1^T can be computed in polynomial time; this turns out to be true.

For a MIN/MAX tree T we use a pair of witnesses, an L-witness and a U-witness, that prove matching lower and upper bounds respectively on the value of the tree. One can then define appropriate lower bound functions f_L^T, f_U^T similar to the functions f_1^T, f_0^T (for Boolean trees) respectively, and run two copies of BALANCE simultaneously (balancing the cost they incur), one trying to prove a lower bound (on the value of T) and using f_L^T for balancing, and the other trying to prove a matching upper bound (and using f_U^T for balancing), till these two bounds match.

Theorem 2.9 *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).

3 Ultra-uniform Prices

Given a Boolean 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. (Clearly 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 a Boolean tree T with n leaves, one can find an ultra-uniform price vector for T in polynomial (in n) time.*

Proof: The idea is to ensure that the cost of all 0-witnesses of T is the same, say c_0, and similarly that the cost of all 1-witnesses of T is the same, say c_1 (the costs c_0, c_1 need not be equal).

We first claim that any setting of prices satisfying the above property is in fact 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 these prices are indeed ultra-uniform.

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 a Boolean tree T, then it holds for all subtrees of T as well, and this actually shows that such a price vector is unique up to scaling. This 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 $C_1[v]$ which 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 (which we assume for definiteness to be an AND node) with children $u_1, u_2, \ldots, u_k$ (which are OR nodes). Our goal is to construct an ultra-uniform price vector for T_u, the subtree of T rooted at u, from the ultra-uniform price vectors $\vec{P}_i$ of the T_{u_i}'s. Since u is an AND node, a 0-witness of T_u is simply a 0-witness of one of the T_{u_i}'s. Hence 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 $1 \le i < j \le k$ 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 thus an ultra-uniform price vector for T exists and can be found in polynomial time. ∎

4 Searching with Prices

4.1 A near-optimal algorithm

We outline 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 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 c. 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. Let group j consist of all elements with cost 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 *effective 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 effective interval M. Such comparisons are called *regular* comparisons and each such comparison is guaranteed to halve the size of the effective interval. This certainly makes progress as long as the element q lies within the effective interval. However, if q does not belong to the current group j, at some point, q could fall outside the effective 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 c regular comparisons of q with group j elements, we perform an *extra* comparison by querying one of the extreme group j elements. This checks if q lies outside the effective 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.

Algorithm Search

1. $I \leftarrow [1 \ldots n]$, $j \leftarrow 0$, $\mathit{left\_cnt} \leftarrow 0$, $\mathit{right\_cnt} \leftarrow 0$.
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 $\mathit{left\_cnt} = c$,
   $\mathit{left\_cnt} \leftarrow 0$.
   Let x be the leftmost element of group j in I.
   $\mathit{type} \leftarrow \mathit{EXTRA}$. Jump to Step 6.
4. If $\mathit{right\_cnt} = c$,
   $\mathit{right\_cnt} \leftarrow 0$.
   Let x be the rightmost element of group j in I.
   $\mathit{type} \leftarrow \mathit{EXTRA}$. Jump to Step 6.
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 *effective interval*.

   Let x be the element in group j that is closest to the middle of M, breaking ties arbitrarily.
   $\mathit{type} \leftarrow \mathit{REGULAR}$.
6. Let $I = I_L \circ x \circ I_R$.
7. Compare q to x.
8. If $x = q$, return **PRESENT**
   else if $q < x$,
   $I \leftarrow I_L$,
   if $\mathit{type} = \mathit{REGULAR}$
   $\mathit{left\_cnt} \leftarrow \mathit{left\_cnt} + 1$; $\mathit{right\_cnt} \leftarrow 0$.
   else if $q > x$,

$I \leftarrow I_R$,
if *type* $=$ *REGULAR*
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 an *extra* 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 in performing regular comparisons.

Lemma 4.1 *For all regular comparisons performed on group j the length of the effective interval goes down 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 $|M_L| \leq |M_R|$. Hence, $|M_L| \leq (|M|-1)/2$. 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, $|M'| \leq |M_L| + 1$. We claim that $|M'| \leq \frac{1}{2}|M|$. If $|M_L| < (|M|-1)/2$, $|M'| \leq |M_L| + 1 \leq \frac{1}{2}|M|$. If $|M_L| = (|M|-1)/2$, x is exactly the middle element of M. Thus $|M_R| = (|M|-1)/2$ and $|M'| \leq |M_R| < \frac{1}{2}|M|$.

If $q < x$, the effective interval is a subinterval of M_L. Suppose $q > x$. In this case, the effective interval is M'. In both cases, the size of the effective 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

$$c_j \leq \left(1 + \frac{1}{c}\right) \log_2 \left(\frac{n_j}{n_{j+1}}\right) + c + 2$$

Proof: Let I_j be the search interval at the first time that the algorithm considers elements of group j. I_{j+1} must have been *created* by comparisons to the elements immediately to the left and right of I_{j+1} (say x_l and x_r respectively). Suppose that x_l was compared to before

x_r. We will bound separately, the number of comparisons of group j performed up to the comparison with x_l and the number after the comparison with x_l.

Consider the number of comparison steps performed up to the point that x_l was compared with. Throughout this time, I_{j+1} is part of the effective interval. Let e_1 be the length of the effective interval at the first time that group j is considered and e_2 be the length of the effective interval just before x_l is compared with. $e_1 \leq |I_j| = n_j$ and $e_2 \geq |I_{j+1}| = n_{j+1}$. Since each regular comparison reduces the length of the effective interval by at least 2, the number of regular comparisons before x_l is compared is at most $\log_2(e_1/e_2) \leq \log_2(n_j/n_{j+1})$. Further, the number of extra comparisons performed during this time is at most $1/c$ times the number of regular comparisons, since each extra comparison can be charged to c regular comparisons. Thus the total number of comparisons including the comparison to x_l is at most

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

After the comparison with x_l, 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 effective 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 $c + 1$ such comparisons. If within c more 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 $c + 1$ steps.
Adding the two bounds, we get the bound in the statement of the lemma. ∎

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

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

$$\begin{aligned}
\sum_{j=0}^{m} r^j \cdot c_j &\leq \sum_{j=0}^{m} r^j \left(\left(1 + \frac{1}{c}\right) \log_2 \left(\frac{n_j}{n_{j+1}}\right) + c + 2\right) \\
&= \left(1 + \frac{1}{c}\right) \sum_{j=0}^{m} r^j \cdot \log_2 \left(\frac{n_j}{n_{j+1}}\right) + (c+2) \sum_{j=0}^{m} r^j \\
&\leq \left(1 + \frac{1}{c}\right) r^m \log_2 n + (c+2) \frac{r^{m+1}}{r-1}
\end{aligned}$$

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

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

Setting $r = 1 + 1/\log_2^{1/3} n$ and $c = \log_2^{1/3} n$, we get the desired bound. ∎

We can improve the competitive ratio by modifying the algorithm slightly. The idea is to change the way in which extra comparisons are performed. Note that in the algorithm described above, the number of extra comparisons for group j is of the form $c + \frac{1}{c}\log_2\left(\frac{n_j}{n_{j+1}}\right)$. The improvement comes from balancing the two terms in this expression. The modified algorithm does not use the parameter c. We keep track of the total number of regular comparisons performed so far for the current group. An extra comparison is performed every time the total number of regular comparisons equals a perfect square. As before, let c_j be the total number of comparisons performed with elements of group j. We can prove that

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

Setting $r = 1 + 2/\sqrt{\log_2 n}$, we can prove that the competitive ratio of the algorithm is bounded by $\log_2 n + O(\sqrt{\log n}\log\log n)$. We omit the details in this extended abstract.

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, we let $f(I, x) = 0$ for all x. Now for all larger intervals I, we define:

$$f(I = [a \ldots b], x) = \min_{i \in I}\Big[\max\{\begin{array}{l} f([a \ldots (i-1)], x + c_i), \\ (x + c_i)/c_i, \\ f([(i+1) \ldots b], x + c_i)\}\Big] \end{array} \tag{8}$$

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 $f(I, x)$.

Acknowledgments

We thank Ravi Kumar for useful discussions and for suggesting the generalization to threshold trees.

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*, 1994, 163–171.

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

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

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

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

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

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

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

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

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

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

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

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