On the Worst-Possible Analysis of Weighted Comparison-Based Algorithms

By

Dana Richards

Computer Science Report #TR-86-04

March 1986

On the Worst-Possible Analysis of Weighted Comparison-Based Algorithms

Dana Richards

University of Virginia
Charlottesville, VA 22903

1. Introduction

The typical analysis of a comparison-based algorithm is concerned with the total number of comparisons. This is evidenced in the vast literature on sorting and order statistics (e.g. [KNUT73]). We assume that each record has a price or weight associated with its use, perhaps reflecting the length of the key or the difficulty of access. The price is independent of the value of the key field in each record. The cost of a comparison is the sum of the prices of the comparands (though other measures could be imagined). We are concerned with the worst-case total cost of an algorithm, i.e. the most costly path through its decision tree.

There are several types of analyses we could do and they are best understood by analogy with F. K. Hwang's "group testing game" [HWAN84]. There are two players G and H, where G poses a problem instance for which H must a give an algorithm that solves it. The problem which Hwang addressed was to find the set of d counterfeit coins in a set of n coins. The basic operation was a ternary query, i.e. "good, bad, or mixed", of any arbitrary subset. By changing the rules to have queries of two elements only with the responses $>$ and $\leqslant$, we get closer to the model in this paper.

Hwang described three ways of playing such a game. While our initial work predated Hwang's paper we find it convenient to use the same terminology:

- G is almighty, the *A-version*. In this version G will know the algorithm H will choose. G will assign the values to the prices and the keys in such a way as to force H to have the worst total cost. H does not know the prices.
- G is just, the *J-version*. In this case G does not know the algorithm H will adopt before it assigns the values to the prices of the records. However G will still try to maximize the cost for H by assigning the key values after knowing H's algorithm. Again H does not know the prices.

- G is merciful, the *M-version*. In this version G will reveal the price of each record to H initially. H may spend much time choosing its algorithm. Again G will assign key values in an adversarial manner.

The M-version above differs greatly from Hwang's definition (which did not even have a notion of prices). The original definition called for G to be completely random, which is appropriate for a discussion of the average performance H could expect. However we are only concerned with worst-case analysis.

If G is merciful then we are in a position to choose the best possible algorithm, though it may be prohibitively expensive to find it. In [RICH84] we discuss how H can use preprocessing to produce a comparatively short list of algorithms to choose from when the prices become known. We used a dynamic programming approach which does not extend to the case when the prices remain unknown.

In this paper we assume that G is almighty, unless otherwise stated. So we must adopt a pessimistic stance. G is an adversary who will assign the prices and answer our comparisons so that H will get the "worst-possible" cost. In particular, G will not only take H down a costly path of our decision tree but will make the most used key the most expensive and so on. In the next section we present some formalisms and in the last section we concentrate on the problem of finding the maximum.

2. Preliminaries

We expect a comparison-based algorithm to be represented by its *decision tree*. Each internal node compares the keys of two records and we branch to one of its two sons depending on the result. Let the n keys be $K_1, K_2, \ldots, K_n$ with associated prices $p_1, p_2, \ldots, p_n$. (To simplify our exposition we assume that G is constrained to always use the same set of prices and just permutes them as best possible. Our results do not assume any distribution on the prices beyond that they could be skewed.) Each path from the root to a leaf has an *outcome*, which is a vector $\mathbf{c} = (c_1, c_2, \ldots, c_n)$, where c_i is the number of comparisons involving K_i in that path. Each outcome is identified with the (relative) ordering of the keys that defines its path in the decision tree. The *path cost* of that outcome is $\sum_{i=1}^{n} p_i c_i$. The *algorithm cost* is the maximum of the path costs.

To analyze the A-version we sort each outcome. Let $\mathbf{c}' = (c_1', c_2', \ldots, c_n')$ denote the vector with the same elements as $(c_1, c_2, \ldots, c_n)$ sorted in nonincreasing order. We say $\mathbf{c}$ *dominates* $\mathbf{d}$ if $\sum_{i=1}^{j} c_i' \geqslant \sum_{i=1}^{j} d_i'$, for all $1 \leqslant j \leqslant n$, that is $\mathbf{c}'$ "majorizes" $\mathbf{d}'$. Further we say $\mathbf{c}$ *strictly dominates* $\mathbf{d}$ if $\mathbf{c}'$ and $\mathbf{d}'$ are distinct.

Lemma 1: If G is almighty and $\mathbf{c}$ dominates $\mathbf{d}$ then G can have the path cost of $\mathbf{c}$ be greater than or equal the path cost of $\mathbf{d}$.

Proof: G can assign the most used key of $\mathbf{c}$ the largest price, and so on. Clearly the path cost of $\mathbf{d}$ is maximized if it also happens to have its most used key assigned the highest price, and so on. But even then the result follows immediatedly. □

Hence in analyzing an algorithm H need consider only those outcomes which are not strictly dominated. We call these the *worst-possible outcomes*.

When choosing between two algorithms H can reject the first if every outcome of the second is dominated by an outcome of the first. We say the first algorithm *dominates* the second, and that it *strictly dominates* the second if the worst-possible outcomes are different. This can be easily decided by comparing just their worst-possible outcomes. If H cannot reject all algorithms but one with this criterion then H is in a no-win situation, due to the following simple observation. Let $\mathbf{c}$ and $\mathbf{d}$ be a worst-possible outcome for the first and second algorithm of the remaining algorithms, respectively. If neither $\mathbf{c}$ nor $\mathbf{d}$ dominates the other then the adversary G could come up with assignments of the prices that would favor either the first or the second algorithm. We call an algorithm *optimal* if it does not strictly dominate another algorithm for the same problem; there may be several optimal algorithms.

Finding an optimal algorithm for the sorting problem appears to be difficult. The best known algorithms with respect to our worst-possible case analysis are serializations of $O(\log n)$ depth sorting networks (e.g. [AJTA83]). All other sorting algorithms, that we have seen, are rejected because they have some comparands that are not in $O(\log n)$ comparisons.

3. Max-Finding Algorithms

We turn our attention to the problem of finding the maximum of n keys when G is almighty, since this is a problem we can completely solve. We will characterize the optimal algorithms. The analysis is simplified because there is a unique worst-possible outcome for these algorithms. A balanced knock-out tournament, discussed below, suggests itself. In that case the keys are all equally vulnerable and H is not giving the adversary any advantage. We show that such a tournament is optimal but we must be careful in defining "balanced".

First we observe that the M-version of the max-finding game, where the prices are known to H, is trivial. H will compare the two cheapest keys, then the third cheapest will be compared with the previous winner, and so on. Clearly the best G can do is to have the more expensive comparand win each comparison. (If the problem is somehow constrained so that H must use a tournament with the records assigned to the leaves in a given left-to-right order then the a solution is also known, based on Huffman's algorithm [ZHAN84].) Below we deal with the A-version.

Any full binary tree on n leaves is identified with a "knockout" *tournament*. That is, the n leaves correspond to keys and internal nodes are "matches" which the greater key wins and advances to the next match, i.e. the parent node. (The tournament tree describes an algorithm with a corresponding decision tree. The two trees are otherwise unrelated.) We define a *tourney* to be a tournament based on a binary tree which has height $\lceil \log_2 n \rceil$ and one of the two subtournaments has exactly 2^k participants, $k \geqslant 0$, and both subtournaments are themselves tourneys. A *perfect tourney* is for 2^k participants, that is all the leaves are at depth k.

Lemma 2: Every max-finding algorithm dominates some tournament.

Proof: If the algorithm never involves the loser of a comparison in further comparisons then it already corresponds to a tournament. Otherwise select a path in the decision tree of the algorithm such that a previous loser always loses (or if two losers compete choose arbitrarily). Recall that every path in such a decision tree must have n-1 first-time losers. It is easy to see that the irredundant comparisons done on the path we selected can be represented by a tournament. Clearly the entire algorithm dominates the algorithm defined by this one tournament. □

How is a worst-possible outcome of a tournament calculated? Recall that in this type of analysis the outcomes are sorted to test for domination. So the leading largest terms are the most important. Informally this implies that in the tournament the players at the deepest levels (in both subtrees) should continue to win until the final round. Formally, the cost can be found by labelling each internal node of the tournament with the minimum of the heights of its two subtrees. Combine with these $n-1$ numbers the height of the entire tree; call this set F. Let the sorted values of F be $(f_1', f_2', \cdots, f_n')$. Note an outcome corresponds to an arrangement of wins at the internal nodes.

Lemma 3: Each worst-possible outcome for a tournament is such that the subtournaments had worst-possible outcomes.

Proof: Suppose there were a counterexample. It is easy to see that if the outcome of the offending subtournament was replaced by its worst-possible outcome then the outcome of the entire tournament would dominate the previous outcome. □

Corollary 1: For each worst-possible outcome the winner of each comparison is the comparand with the most previous wins, with ties broken arbitrarily.

Proof: A formal induction proof can be posed using lemma 3 and the definition of dominating outcomes. □

Lemma 4: For any worst-possible outcome **c** of a tournament

$$(c_1', c_2', \ldots, c_n') = (f_1', f_2', \cdots, f_n').$$

Proof: From corollary 1 we see that before a match the number of wins for each player is the height of their respective subtrees. It follows that the corresponding set of comparison counts is F. Note that each internal node was labelled with the lifespan of the loser at that point, while the number of games of the ultimate winner is the height of the tree. Figure 1 illustrates the correspondence. (It is an interesting exercise to show with graph-theoretic arguments that the sum over F is $2(n-1)$, as it must be.) □

In the sequel we will speak of "the" worst-possible outcome of a tournament since it is essentially unique. It is the simple structure of the worst-possible outcome in the preceding proof that permits us to study tournaments. We now want to show that among all tournaments H will prefer tourneys. The next lemma shows, for example, that it does not matter if the tourney for $n=19$ splits elements initially so that we get the best of 16 against the best of 3, 8 against 11, or 4 against 15.

Lemma 5: The worst-possible outcome for every tourney on n keys is the same.

Proof: If $n = 2^k$ there is only the one perfect tourney, otherwise consider the

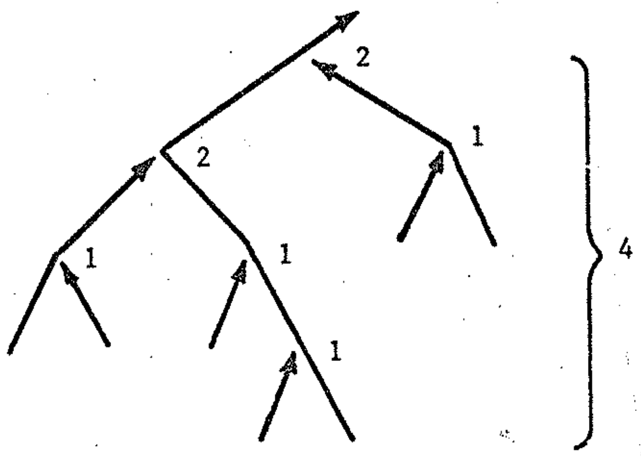

Figure 1

tourney in figure 2. We let n_a be the number of leaves in tree T_a, and so on. In figure 2 $n_a = 2^{\lfloor \log_2 n \rfloor}$, i.e. it is as large as possible. It is easy to verify every tourney is isomorphic to one formed by the following interchange: switch some perfect subtourney $T_{a'}$ of T_a with T_b, when $n_{a'} \geqslant n_b$. Note that this interchange does not effect F. □

The next result is the key lemma, and it is easily verified for small n, $n \leqslant 6$. For example, for $n = 6$ if we end with the best of 4 against the best of 2 the (sorted) outcome is (322111); however 3 against 3 gives the undesired outcome (331111).

Lemma 6: The worst-possible outcome of every tournament that is not a tourney strictly dominates the (unique) outcome of any tourney on the same keys.

Proof: Suppose n is the least number of leaves leading to a counterexample. From the preceding discussion we see $n > 6$. Wlog assume this tournament is as shown in figure 3, where T_b and T_c are perfect tourneys, $n_a \neq n_b, n_c \neq n_d$, and $n_b > n_c$. Recall by lemma 2 and the minimality of n that the subtournaments must be tourneys. By lemma 5 we can assume $n_b > n_a$ and $n_c > n_d$. Now, as in figure 2, we cut a perfect subtourney $T_{b'}$ out of T_b and switch it with T_d, where the heights of

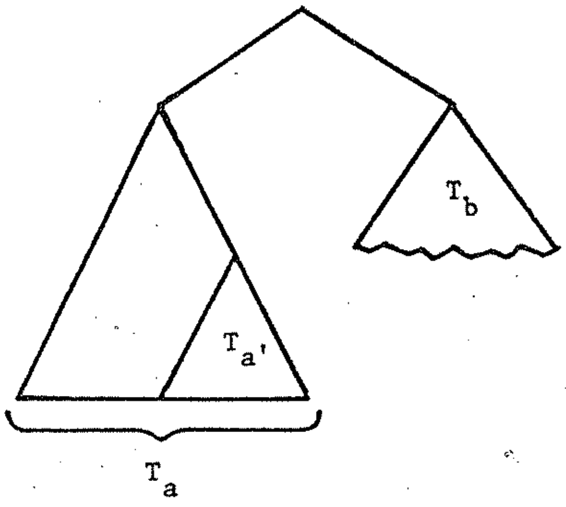

Figure 2

T_b' and T_d are the same. Note F is unchanged. Now the left subtree is not a tourney. The left subtree with lemma 3 contradicts the minimality of n. □

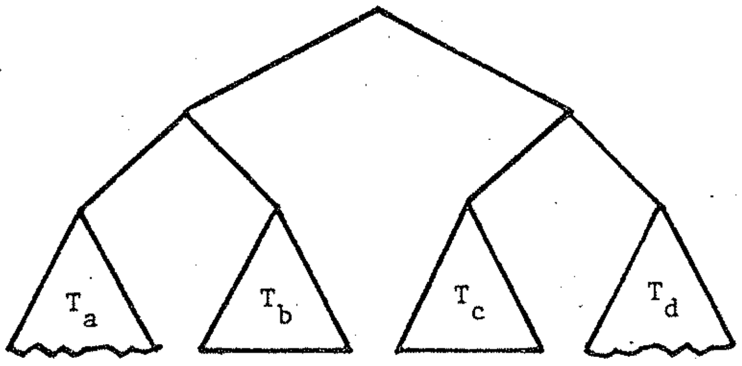

Figure 3

Theorem 1: Every max-finding algorithm dominates any tourney in the worst-possible case.

Proof: This follows from lemmas 2 and 6. □

Finally we remark that the J-version, where the prices are assigned randomly, can be easily solved. H will use a "balanced tournament" [HWAN77] which has every leaf at depth $\lceil \log_2 n \rceil$ or $\lceil \log_2 n \rceil - 1$. (It is not necessarily a tourney.) The reasoning is that any tournament with greater height would dominate this. Further among tournaments of the same height this one has the fewest nodes at depth $\lceil \log_2 n \rceil$ and therefore minimizes the chances of having expensive records assigned to them. It is interesting that the same type of tournament has been conjectured to give the best average performance for stochastic tournaments; this has been shown for a few probability distributions [HWAN77,MAUR75]. It would be interesting to investigate any connection between those studies and J-version problems.

4. References

[AJTA83] M. Ajtai, J. Komlos and E. Szemeredi, Sorting in c log n Parallel Steps, *Combinatorica*, **3**, 1983, pp. 1-19.

[HWAN77] F. K. Hwang, Several Problems on Knockout Tournaments, *Proc 8th S.E. Conf Combinatorics, Graph Theory, and Computing*, 1977, pp. 363-380.

[HWAN84] F. K. Hwang, Three Versions of a Group Testing Games, *SIAM J Alg Disc Meth*, **5**, 1984, pp. 145-153.

[KNUT73] D. Knuth, *The Art of Computer Programming: Sorting and Searching*, Addison-Wesley, 1973.

[MAUR75] W. Maurer, On the Most Effective Tournament Plans with Fewer Games than Competitors, *Ann Statistics*, **3**, 1975, pp. 717-727.

[RICH84] D. Richards, Sorting with Expensive Comparands, *Intl J Computer Math*, **16**, 1984, pp. 23-45.

[ZHAN84] C. Zhang, Optimal Alphabetic Binary Tree for a Nonregular Cost Function, *Disc Appl Math*, **8**, 1984, pp. 307-312.