A Least-Certainty Heuristic for Selective Search[1]

Paul E. Utgoff utgoff@cs.umass.edu
Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003

Richard P. Cochran cochran@cs.umass.edu
Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003

Abstract: We present a new algorithm for selective search by iterative expansion of leaf nodes. The algorithm reasons with leaf evaluations in a way that leads to high confidence in the choice of move at the root. Performance of the algorithm is measured under a variety of conditions, as compared to minimax with α/β pruning, and to best-first minimax.

Keywords: Selective search, evaluation function error, misordering assumption, certainty, confidence, swing, evaluation goal, swing threshold, LCF, random trees, artificial time, Amazons, Othello.

1 Introduction

It has been recognized for quite some time that some moves in a game are hopelessly suboptimal, and that search effort should not be expended in exploring lines of play that emanate from them. Several methods have been devised to grow the tree selectively, as discussed below. We present a new algorithm of this kind, and investigate its characteristics.

That some positions are obviously worse than others rests on an evaluation mechanism that recognizes apparently debilitating or fatal flaws in the position. It is pointless to investigate the nuances of the poor play that will ensue. Less obvious are those positions in which lesser strengths and weaknesses are evident. Nevertheless, issues of search control remain paramount. It is best to invest effort where it will help the decision process. This calls for a more thorough expansion of lines of play in which the relative advantage of a position is less certain, and a less thorough expansion of lines in which the relative advantage is more certain.

2 Evaluation Function Error

The purpose of search is to obtain information that enhances the decision process at the root. With an evaluation function that is perfectly correlated with the true game value, there is no information to be gleaned from searching. Similarly, with an evaluation function that has no correlation with the game value, search is equally uninformative. Search is useful when it serves to overcome error in an evaluation function that is only imperfectly correlated.

An evaluation function assesses present utility by measuring indicators of future merit. When these indicators predict imperfectly, as will typically be the case, search can reduce the effect of

[1]The correct citation for this article, (C) 2001 Copyright Springer Verlag, is: Utgoff, P. E., & Cochran, R. P. (2001). A least-certainty heuristic for selective search. *Proceedings of the Second International Conference on Computers and Games* (pp. 1-18). Springer Verlag.

prediction errors. For example, one can either estimate the weight of an object (heuristic evaluation), or instead actually weigh it (search). The estimate is imperfectly correlated with the actual weight, and weighing the object obviates the need to use a predicted value.

Because the evaluation function assesses a position imperfectly, it would be best to search those positions for which the evaluation errors are most likely to affect the decision at the root. To the extent that the distribution of errors is known, it is possible to improve the evaluation function itself. The very nature of heuristic evaluation is that the error distribution of the evaluation function cannot be known.

To direct search effort in a useful manner, it is necessary to make at least some weak assumptions about the error distribution. We make three assumptions, all of which are quite common. The first is that the evaluation function is the best available estimator of the game value of a node without searching below it. The second is that the variance in the error of the evaluation function is not so large that it renders comparison of evaluations meaningless. Finally, we assume that the evaluation of a position is on a linear scale, such that the size of the difference in two evaluations is a useful measure of the difference in quality of the two positions.

Minimax search has the strength that its brute force search control is not guided by the evaluations of the states, making the error distribution irrelevant. However, the algorithm pays the price of searching to a uniform depth, wasting search effort on many suboptimal lines of play. Selective search has the weakness that its search control is guided by the state evaluations, making it sensitive to evaluation errors and therefore making it susceptible to being misled. However, the approach can search apparently promising lines to much greater depth by not squandering time on those of less merit. There is a classic tradeoff of being unguided and shallow (risk averse) versus being guided and deep (risk accepting).

Although we do not address evaluation function learning here, it is quite common to use an automatic parameter tuning method such as temporal difference learning (Sutton, 1988) to adjust the coefficients of the functional form of the evaluation function. When doing so, the error distribution changes dynamically over time. One needs a decision procedure that is sensitive to the error distribution under these circumstances too.

3 Related Work

There is much appeal to the notion of expending search effort where it has the highest chance of affecting the decision at the root. Numerous approaches have been developed that grow the game tree by iteratively choosing a leaf, expanding it, and revising the minimax values for the extant tree. An alternative is to retain the depth-first search of minimax, but search to a variable depth, such as singular extensions (Anantharaman, Campbell & Hsu, 1990) and ProbCut (Buro, 1997). The discussion here focuses on iterative tree growing.

Berliner's (Berliner, 1979) B* algorithm attempts to bound the value of a node from below and above by a pair of admissible heuristic functions, one pessimistic and the other optimistic. The algorithm repeatedly selects a leaf node to expand that is expected to drive the pessimistic value of one node above the optimistic value of all the siblings. Palay (Palay, 1982) improved the algorithm by providing it with explicit reasoning about which child of the root to descend. This was possible by assuming a uniform probability distribution of the range of values in the node's admissible interval of values. More recently Berliner & McConnell (Berliner & McConnell, 1996) simplified the procedure for backing up probability distributions to the root.

McAllester (McAllester, 1988) proposed a search algorithm based on conspiracy numbers,

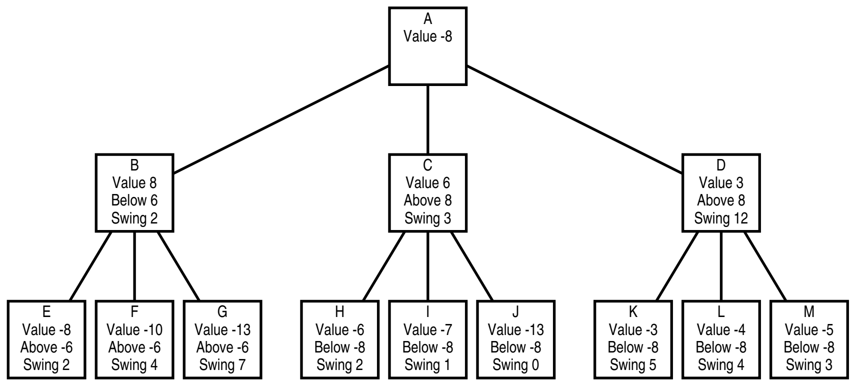

Figure 1. Example NegaMax Search Tree

further illustrated by Schaeffer (Schaeffer, 1990). The algorithm counts the number of leaves whose values would need to change by a given amount to affect the value of the root by a given amount. As the number of such leaves rises, the likelihood of all the values changing as needed drops. When the algorithm becomes sufficiently confident that the best node will remain so, search terminates.

Rivest (Rivest, 1988) suggested guiding the iterative growth process by a penalty-based heuristic. His algorithm expands the leaf node to which the root value is most sensitive. The edge penalty is a function of the derivative of the min or max operator when represented as a generalized mean. The cost associated with a node is the minimum path cost to a leaf below. Starting at the root, the algorithm can follow the path of least total cost to a leaf, which is then expanded. For Connect-4, the algorithm searches fewer nodes than minimax, but with higher total cost due to overhead.

Proof-number search (Allis, van der Meulen & van den Herik, 1994) of Allis & van den Herik is similar in spirit to conspiracy numbers. Instead of exploring various candidate game values, the algorithm temporarily equates all values below a candidate threshold with a loss, and the rest with a win. With just two possible values, the algorithm searches selectively by following a path that requires the fewest leaf values to change in order to change the decision at the root. This search mechanism is contained within an outer binary search on candidate games values, allowing the algorithm to home in on the game value at log cost in candidate game values. The algorithm depends on a variable branching factor.

Korf & Chickering (Korf & Chickering, 1996) explored best-first minimax search, which always expands the leaf of the principal variation. The algorithm depends on error in the evaluation function, and it also depends on tempo effects to cause its opinion of the principal variation to change as search proceeds. Expansion of a leaf can make it look less attractive because the opponent has had a following opportunity to influence the position in its favor.

These algorithms all endeavor to spend search effort expanding those nodes that are needed to reach the best decision at the root, given the error in the evaluation function. We offer the LCF algorithm, which is based on a different heuristic for selecting the node to expand next. The algorithm is closest in spirit to conspiracy numbers, but discards buckets of supposed target values in favor of a real-valued approach.

```
select_move()
  1. allocate time for this move
  2. update_all_swing()
  3. while (time remains and tree incomplete and swing below threshold) grow_tree()
  4. return best move
grow_tree()
  1. descend path of least non-zero swing to leaf
  2. expand leaf
  3. backup negamax value and swing value
  4. if new or revalued first or second child then update_all_swing()
update_all_swing()
  1. for first child, update_swing_below(first,second→lookahead)
  2. for each non-first child, update_swing_above(non_first,first→lookahead)
update_swing_above(node,val)
  1. if node is a leaf then set node→swing to max(0,val-node→lookahead)
  2. otherwise, for each child of node, update_swing_below(child,-val)
     and set node→swing to sum of swings of children
update_swing_below(node,val)
  1. if node is a leaf then set node→swing to max(0,node→lookahead-val)
  2. otherwise, for each child of node, update_swing_above(child,-val)
     and set node→swing to minimum of swings of children
```

Figure 2. Move selection for LCF

4 A Least-Certainty Heuristic

The purpose of search is to establish a high-confidence best choice at the root. How can this top-level goal be realized as a search strategy? Our approach is patterned as a proof by contradiction, but instead of obtaining an absolute guarantee of contradiction, we accumulate as much evidence as possible. Assume that the actual evaluations of the first and second best children at the root misorder them with respect to the unknown true evaluations. Under this *misordering assumption*, the evaluations of these two children have at least as much error collectively as the difference in their evaluations. Define *swing* of a node to be the minimum amount of change needed in its actual evaluation to alter the decision at the root. Now the search strategy can be oriented to achieving this minimum change at a child of the root.

We associate a larger swing with a lower probability that the original misordering assumption is correct. To the extent that this probability can be driven near zero, one can reject the misordering assumption with increasing confidence. We do not compute confidence levels explicitly, but instead work directly with swing values. As the amount of minimum swing (needed to change the decision) grows, so too does the confidence that the misordering assumption is false. This approach is motivated in much the same way as conspiracy numbers, but we depart by considering the minimum amount of real-valued swing, not the coarser number of leaf evaluations that would need to change by some fixed amount. When considering the swing for each of the evaluations, one should search a line of play in which the swing is least. Our algorithm for doing this is called LCF because is uses best-first search to investigate the line of least certainty (confidence/swing) first.

Consider the example search tree in Figure 1, which assumes that the value of a node is the negative of the maximum (negamax) of its children. Although the figure shows a fixed depth tree, the search tree will typically vary in depth. At any node, the best move is the one that produces the highest-valued child. Node B, with value 8, is currently the highest-valued child of the root. The second best child is node C, with value 6. To change the move selection at the root, node B must come not to have the highest value. One can attempt to accomplish this in a variety of ways.

One can infer a target value for a node based on a value that, if achieved, would change the decision at the root. The only means of changing a value of a node is to expand a leaf at or below the node such that the backed-up move value at the node will be different. So, in this example, one goal for changing the decision at the root would be to drive the value of node C above 8. Another goal would be to drive the value of node B below 6. Yet another goal would be to drive the value of node D above 8. Generally, one can attempt to drive the best value down below the second best value, or one can attempt to drive a non-best value above the best value. There is no need to consider simultaneous goals, such as to drive B down to 6.8 and drive C up to 6.9 because one can achieve the same effect by proceeding one goal at a time, which is necessarily enforced when expanding one leaf at a time.

Given the three goals, one each for each child of the root, which should be pursued first? If these three nodes were leaves, then one could reason that to change the value 3 of node D to its goal of 8 would require a swing of 5 in the valuation. However, to change the value 6 of node C to 8 would require a smaller swing of 2. Similarly, to change the value 8 of node B to 6 would also require only a swing of 2. Assuming that the size of swing is related to the size of the implied error in the evaluations, smaller swings should be easier to achieve than larger swings. Hence, there is more promise in descending node B or C than for node D.

Consider how evaluation goals and swing values are computed when searching deeper than one ply. How are these values managed in the negamax representation? How is a goal to raise an evaluation handled differently from a goal to lower an evaluation?

Suppose that each of node B, node C, and node D have been expanded previously, and that their children evaluate as indicated in the figure. To drive the value of node D up to 8, one would need to drive the value of *every* child below -8. This is because the negamax formulation assigns node values from the point of view of the player on move, and negates the maximum when backing up to the parent. One negates the evaluation subgoals in a corresponding manner. For example, all values need to be driven below -8, so that the maximum will be below -8, which will push the negated backed-up value above 8.

Given that the goal at each of nodes K, L, and M is to drive its value below -8, one can determine the size of the swing in evaluation needed at each. For node K it is $-3-(-8)=5$, for node L it is $-4-(-8)=4$, and for node M, it is $-5-(-8)=3$. Because all these swings need to occur, the total amount of swing needed below node D is now $5+4+3=12$. The same reasoning applies below node C, giving a total swing of $2+1+0=3$. Notice that node J has value -13, which is already below the goal of -8, so a swing of 0 (none) is needed to meet that goal.

For node B, the goal is to drive its value below 6, which means that the goal for each of nodes E, F, and G is to drive it above -6. For node E, the required swing is $-6-(-8)=2$, for node F it is $-6-(-10)=4$, and for node G it is $-6-(-13)=7$. If any of these goals were achieved, there would be a new maximum value among them, so it is necessary to achieve just one to change the decision at the root. The minimum of the swings is ascribed to the parent node B, giving it a

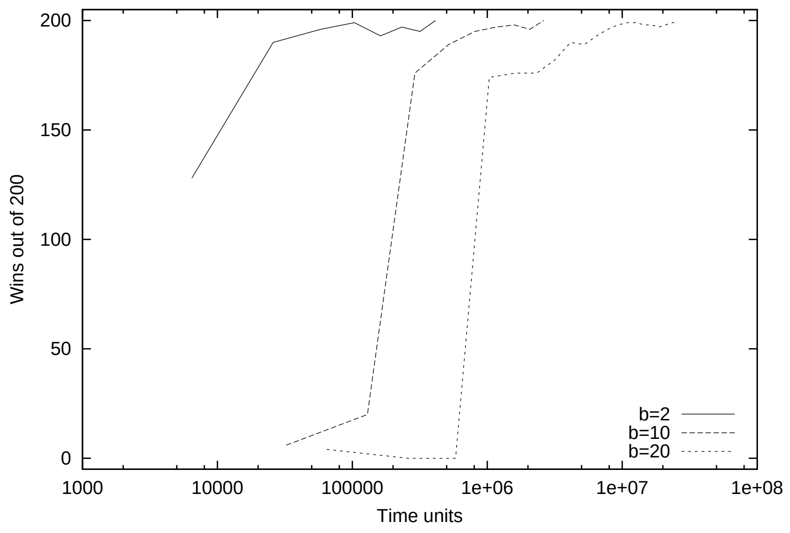

Figure 3. LCF versus AB

swing of 2. Of course it already had a swing of 2. The example would become more interesting after expanding node E, because the swing needed at E would become the sum of the swings of its children.

The LCF algorithm is shown in Figure 2. It is best-first search, with its metric based on total swing required to achieve the goal of changing the decision at the root. One grows the tree by repeatedly starting at the root, and selecting the node with the least non-zero amount of swing required. Note that a swing of 0 is acceptable for children of the root because it indicates two or more top-rated children. Further below however, a 0 means that there is nothing to achieve by searching below the node. Ties by swing value are broken randomly and uniformly. When a leaf is reached, it is expanded, and the normal backing up of negamax values to the root is performed.

The swing values are updated during the backup to the root. If the child selected at the root retains its value, then all the swing values are correct, and nothing more need be done. However, if the value has changed and the child was or is one of the first or second-best values, then all the goals and swing values have been invalidated. In this case, the tree is traversed with the new pair of best and second best values, updating the goals and swing values. From an efficiency standpoint, one would hope for no change in best/second values at the root, but this would be shortsighted since the objective is to determine the best choice at the root.

Finally, how does one know when to stop searching? Of course one needs to stop when the time allocation runs short. However, when the best node is highly likely to remain the best node, the utility of additional searching diminishes. One measure of diminishing utility is the minimum amount of swing needed to change the decision at the root. As this number grows, the chances of achieving it presumably shrink. Depending on the unit of evaluation, one can set a fixed threshold on total swing that, if exceeded, would cause the search to terminate.

5 Experimental Comparison

How does LCF compare to other known algorithms? In this section, we compare three algorithms in three domains. In addition to LCF, we include best-first minimax (BFMM) and minimax

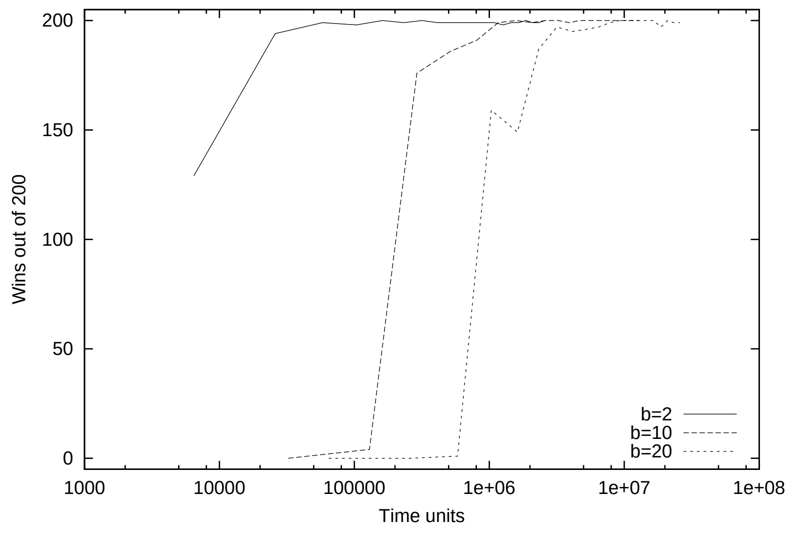

Figure 4. BFMM versus AB

with α/β pruning (AB). The first domain is the class of random-tree games. The second is the game Amazons, and the third is the game Othello.

5.1 Random-Tree Games

The characteristics of a two-person zero-sum perfect-information game can be summarized reasonably well by a small set of parameters. The *branching factor* of the game tree is important because it dictates how quickly the number of states multiplies as a function of search depth. For a fixed amount of search effort, one can search more deeply in a tree that branches less. The length of the game affects the *depth* of the search that is needed to analyze the game tree. For average branching factor b and average depth d, the game tree will contain $O(b^d)$ states. For non-trivial games, one searches a relatively small portion of the game tree when selecting a move. The *total time* available affects how much search effort can be expended during play. More time enables more searching, which improves play in non-pathological games. The *node generation* cost, including node evaluation, also impacts the amount of search that can be accomplished in a fixed amount of time. Finally, the *error distribution* of the evaluation function will mislead search algorithms that are guided by functions of node evaluation.

For our purposes, we vary average branching factor and total time available, holding the others fixed. We follow Korf & Chickering, and Berliner, by using an artificial class of games modeled as random trees. These games can be played at low cost, making a large exploration feasible.

For a random tree, an integer index is associated with each node. For a complete $O(b^d)$ tree, it is straightforward to map each possible state to a unique integer by computing the breadth-order index of the node. The index of each node indexes a random number sequence. This means that a random number is associated with each node, but this indexing method ensures that the mapping of nodes to random numbers is reproducible, no matter what part of the tree is being traversed at any time. The random value associated with each node is the incremental change in the evaluation of that node, called the edge value. The value of the root, which has no edge to a parent, is 0. The heuristic value of a leaf node is the sum of the edge values from that leaf node up to the root.

For our experiments, we follow K&C, using a random number sequence whose period is 2^{32}.

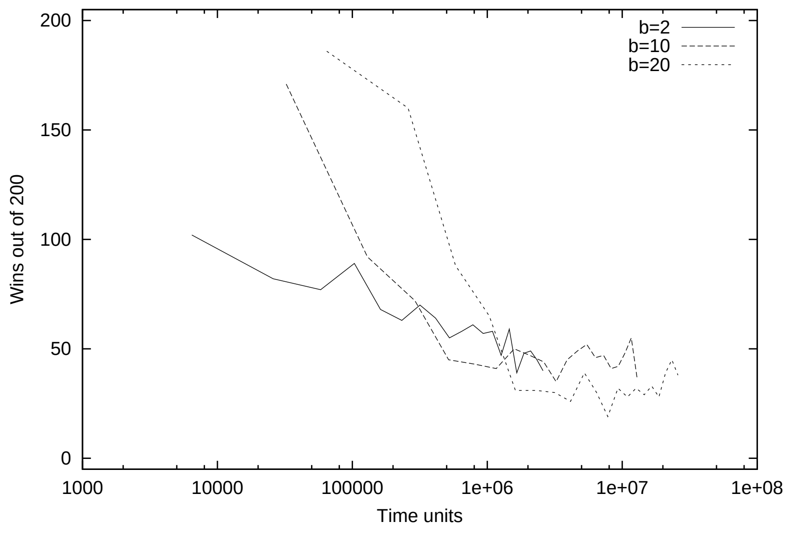

Figure 5. LCF vs BFMM

Every node index is taken as the modulus of this period length, ensuring that it falls within the bounds of the sequence. The random number that is indexed in this way is further modulated to fall within $[-2^{14}, 2^{14}-1]$. The purpose of this indexing scheme is to determine a reproducible random edge value in the evaluation. For any particular branching factor b, game depth d, and random number sequence r, one obtains a particular game. Changing any of these parameters defines a new game. For our experiments, we fixed the depth at 64.

There are two attractive aspects of using random trees. The first is that the computational requirements for node generation and evaluation are only slight. Second, the evaluation function is imperfectly correlated with the true game value. For example, the sum of the edge values from the root to a final game state gives the game value of the final position exactly. Backing up one step, the sum of the edge values from the root to this previous state is well correlated with the game value, with the last unspecified incremental change introducing some variability. This models very well the typical situation in which an evaluation function is better at evaluating positions near the end of the game than near the beginning.

Because node generation is inexpensive, we simulate this cost in artificial time units. This is a departure from previous studies in which time is measured by the number of nodes generated. By using artificial time units, one can charge for a variety of different operations as deemed appropriate. For most of the experiments here, we charge one (1) time unit for traversing a node pointer for any reason, and 100 time units for generating and evaluating a node. The swing cutoff threshold was not used for these experiments because we wanted to minimize confounding time management with heuristic search control. It would help to stop searching when the choice is clear because that time would be available for a subsequent move.

Figure 3 shows the number of wins for LCF when pitted against AB. Artificial time units were charged identically for the AB algorithm, which used static node ordering. Time management for AB was implemented by computing the maximum depth k that could be searched within the time allotment, assuming that $b^{0.75k}$ nodes will be generated. The 200 games played for each branching factor b and time allotment t, consisted of 100 pairs of games. In each pair, the same random

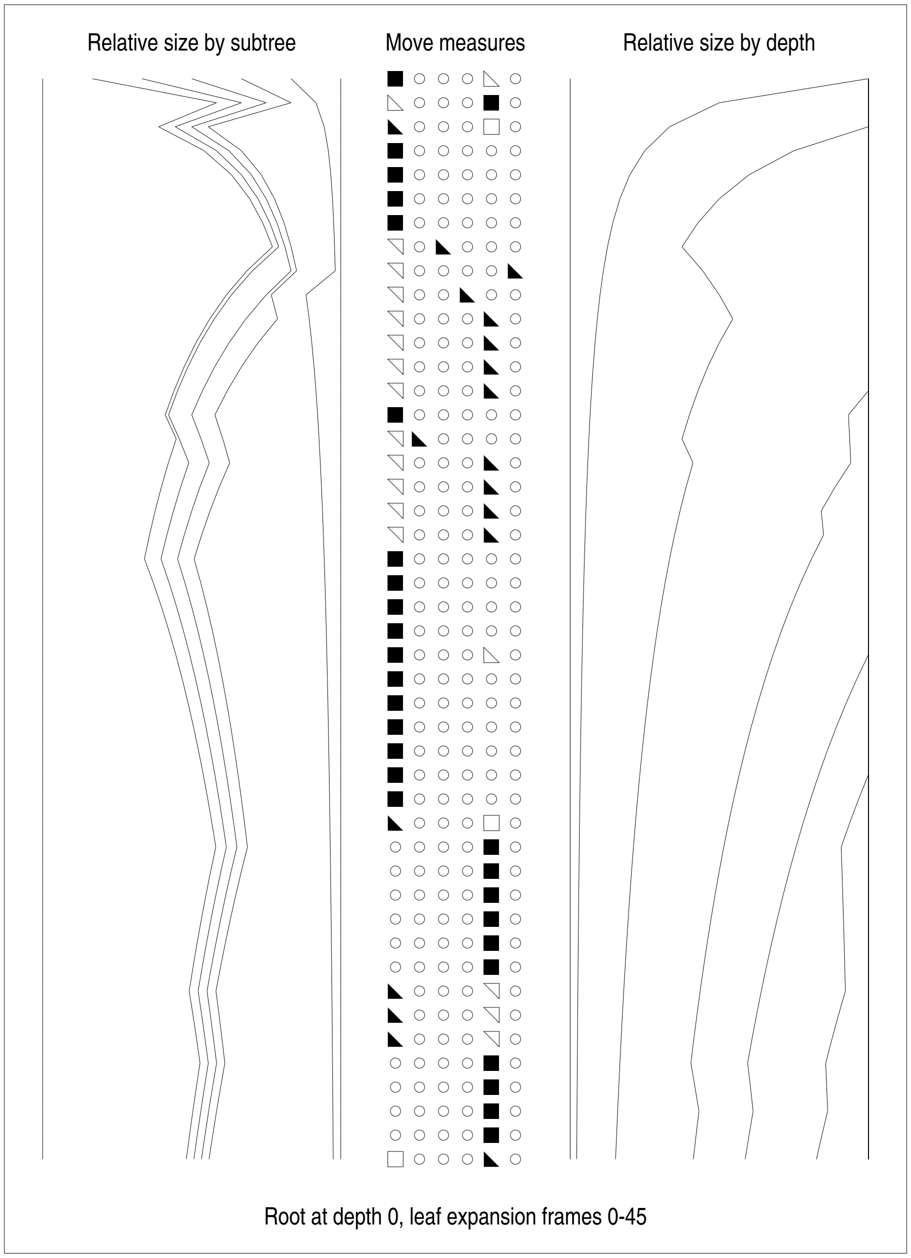

Figure 6. LCF Expansions for Random Tree

number sequence *r* was employed, with each player going first in one of the games. The random number sequence was varied from one pair to the next.

In the figure, one can observe a variety of combinations of branching factor and time allotment. The lines connecting these points help visually with grouping, but are only suggestive of results for intervening time allotments. For average branching factor $b = 2$, LCF won all 200 games for all time allotments. For $b = 10$, LCF lost a majority of the games at the smallest time allotment, but won all 200 games for the remaining allotments. Similarly, for $b = 20$, LCF is at a disadvantage only for the two smallest allotments.

K&C's best-first minimax (BFMM) was implemented by modifying the LCF code to expand

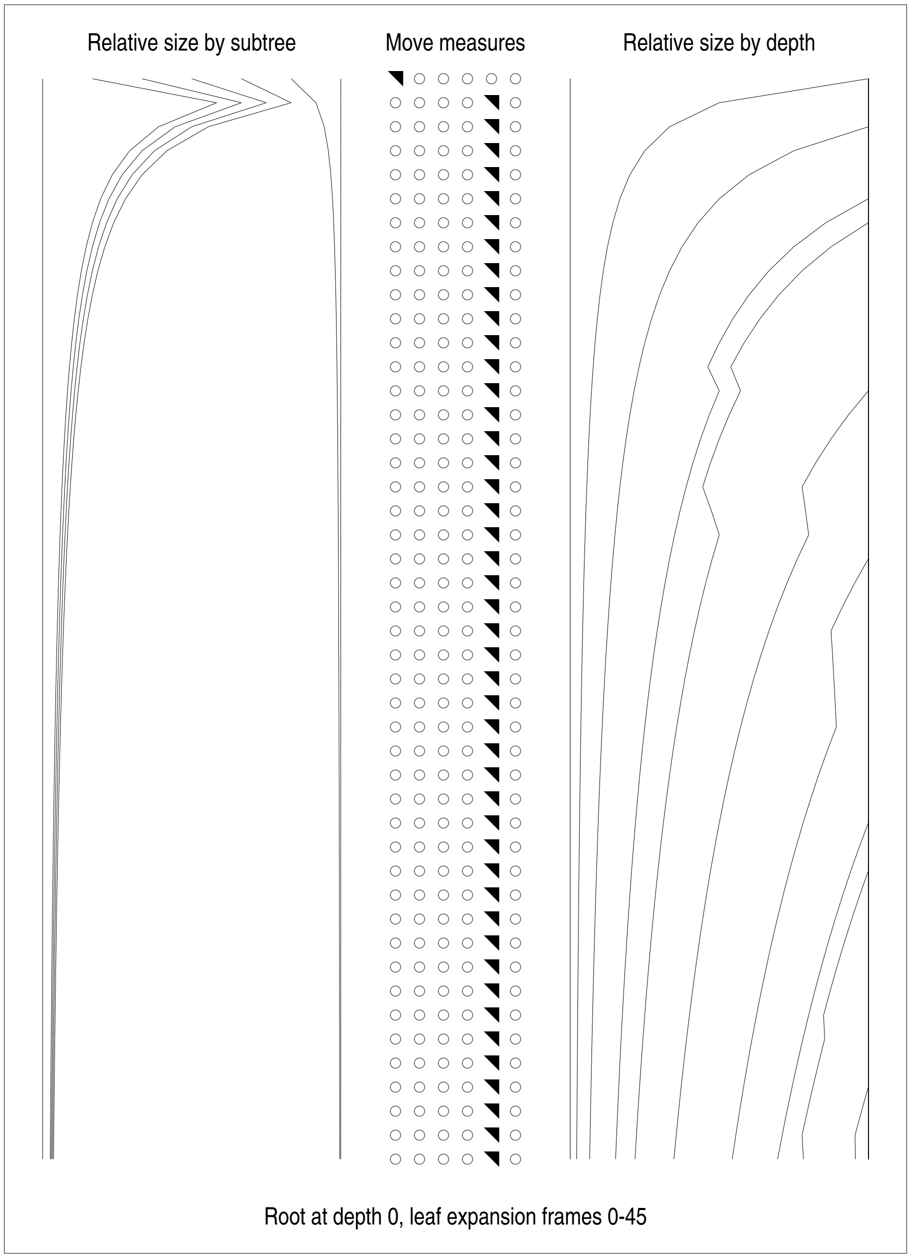

Figure 7. BFMM Expansions for Random Tree

the principal leaf, and to skip computation of swing values. Figure 4 shows how BFMM fared in the identical experimental setup that was used for the LCF comparison. One can observe the same pattern, that the leaf expander is at a disadvantage for the smaller time allotments, and at an advantage for the larger allotments.

Which of LCF and BFMM is stronger under various conditions? Figure 5 shows a comparison following the same experimental design as before. In this context, as the time allotment is increased, the LCF algorithm becomes weaker than the BFMM algorithm. We note that for smaller allotments, LCF is favored. We ran a version of LCF that did not charge for the overhead of maintaining the swing values, and saw the same fundamental pattern. The explanation rests somewhere

Table 1. Amazons Tournaments Results

5 mins		10 mins		20 mins		30 mins		Total	
LCF	0	LCF	0	LCF	0	LCF	2	LCF	2
BFMM	2	BFMM	2	BFMM	2	BFMM	0	BFMM	6
LCF	2	LCF	0	LCF	1	LCF	0	LCF	3
AB	0	AB	2	AB	1	AB	2	AB	5
BFMM	1	BFMM	1	BFMM	0	BFMM	1	BFMM	3
AB	1	AB	1	AB	2	AB	1	AB	5

within the actual node selection strategy.

Figure 6 shows the search activity of the LCF algorithm for the first move of a particular random tree game. Figure 7 shows the search activity of the BFMM algorithm in the same setting. LCF expands 74 leaves in the time that BFMM expands 88 leaves. This difference is most likely due to the overhead cost of LCF in computing swing values. An attractive property of BFMM is that it has no extra measures to compute for the game tree.

Each figure consists of three columns of information. The middle column depicts information regarding the children of the root. A lower triangle indicates a node with lowest swing value, and an upper triangle indicates a node with highest move value. A box identifies a node that has both lowest swing value and highest move value. A circle denotes a node that is suboptimal in both senses. If the symbol is filled (solid black), then that is the child of the root that was selected for descent to a leaf to be expanded.

Notice that LCF explores several moves, principally child$_0$ and child$_4$. BFMM spends most of its time below child$_4$, though at expansion 48 (not shown), it switches to child$_0$ and sticks with it to the end of the search at expansion 88. LCF also settled fundamentally on child$_0$ at expansion 44.

The lefthand column of the figure shows the proportion of nodes in each subtree of the root after each expansion. The values are connected to improve readability. At each expansion (row in the figure), observe the distance between each of the lines. It is evident that for LCF the growth is mostly below child$_0$ and child$_4$, whereas for BFMM the growth is mostly below child$_4$.

The righthand column shows after each expansion (row) the proportion of nodes at each depth of the tree. Initially, all nodes are at depth 1, just below the root. Subsequently, there are expansions at depth 2, and later at various depths. The proportion of nodes at depth 0 is the leftmost area (between lines) in the column. The BFMM algorithm tree becomes narrower and deeper than the LCF tree during this particular search.

5.2 Amazons

Amazons is played on a 10x10 board. Each player has four amazons, which move like chess queens. To take a turn, the player-on-move selects one of his/her amazons, moves it like a chess queen to an empty square, and from its resting point throws a stone as though it were a second chess queen move. Where the stone lands is permanently blocked. The last player to move wins. The game has elements of piece movement and territory capture.

Three version of an Amazons program were implemented, an LCF version, a BFMM version, and an AB version. The LCF version generates the children of a node in a particular way, keeping up to the ten best. The BFMM version is identical except for its search method as described above.

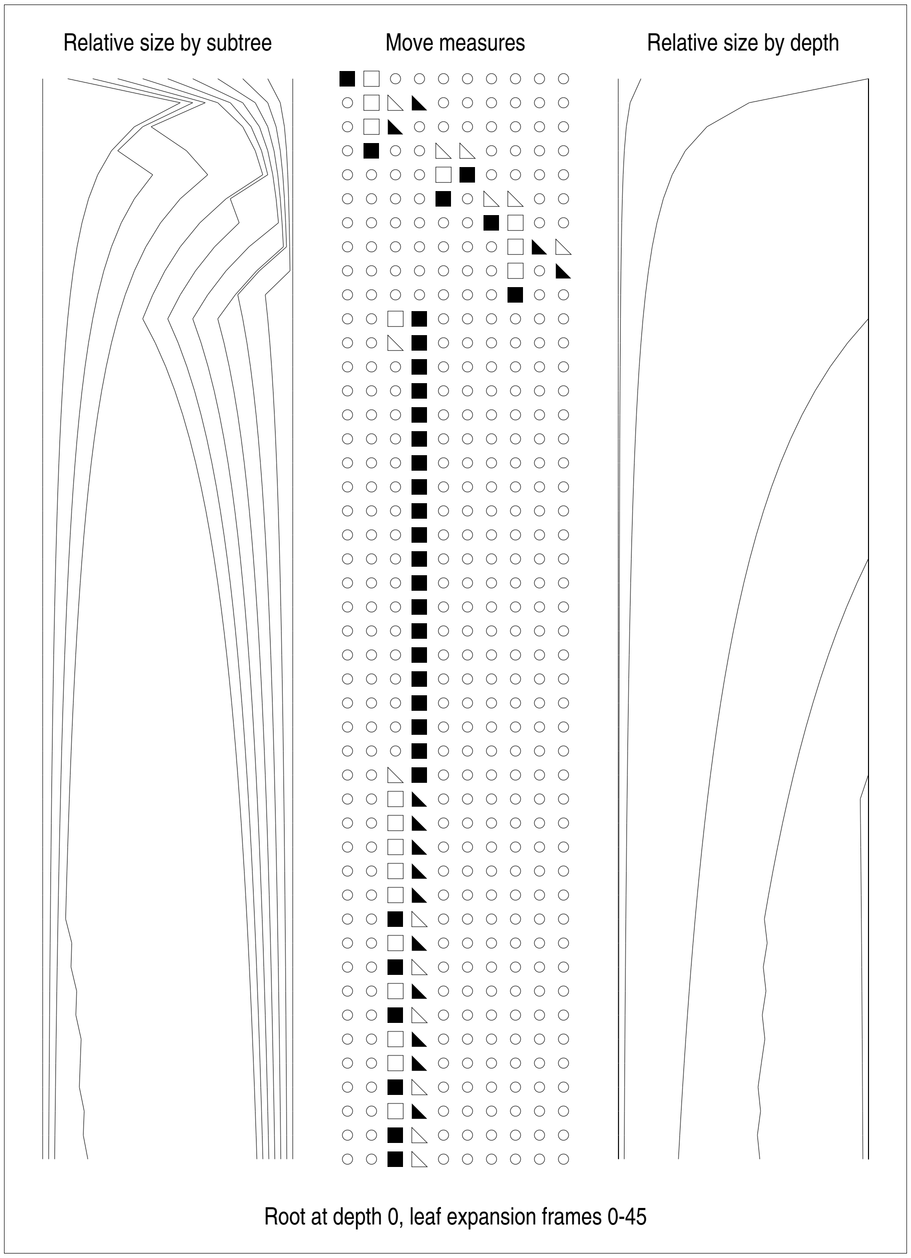

Figure 8. LCF Expansions for Amazons

Similarly, the AB version differs in just its search method and time management policy. It uses iterative-deepening negamax search with static node ordering. The three programs were pitted against one another in four round-robin tournaments of various time allocations. The results are summarized in Table 1. AB won a total of ten (10) games, BFMM nine (9) games, and LCF five (5) games. Again, LCF is weaker than BFMM. Remarkably however, AB is not dominated by LCF or BFMM, as it was in the Random-Trees case. Variable branching factor, an issue discussed by Korf & Chickering, is not an issue here because it was uniformly ten during the decisive portion of the game.

Figure 8 shows the search behavior of LCF during the first move of a game of Amazons.

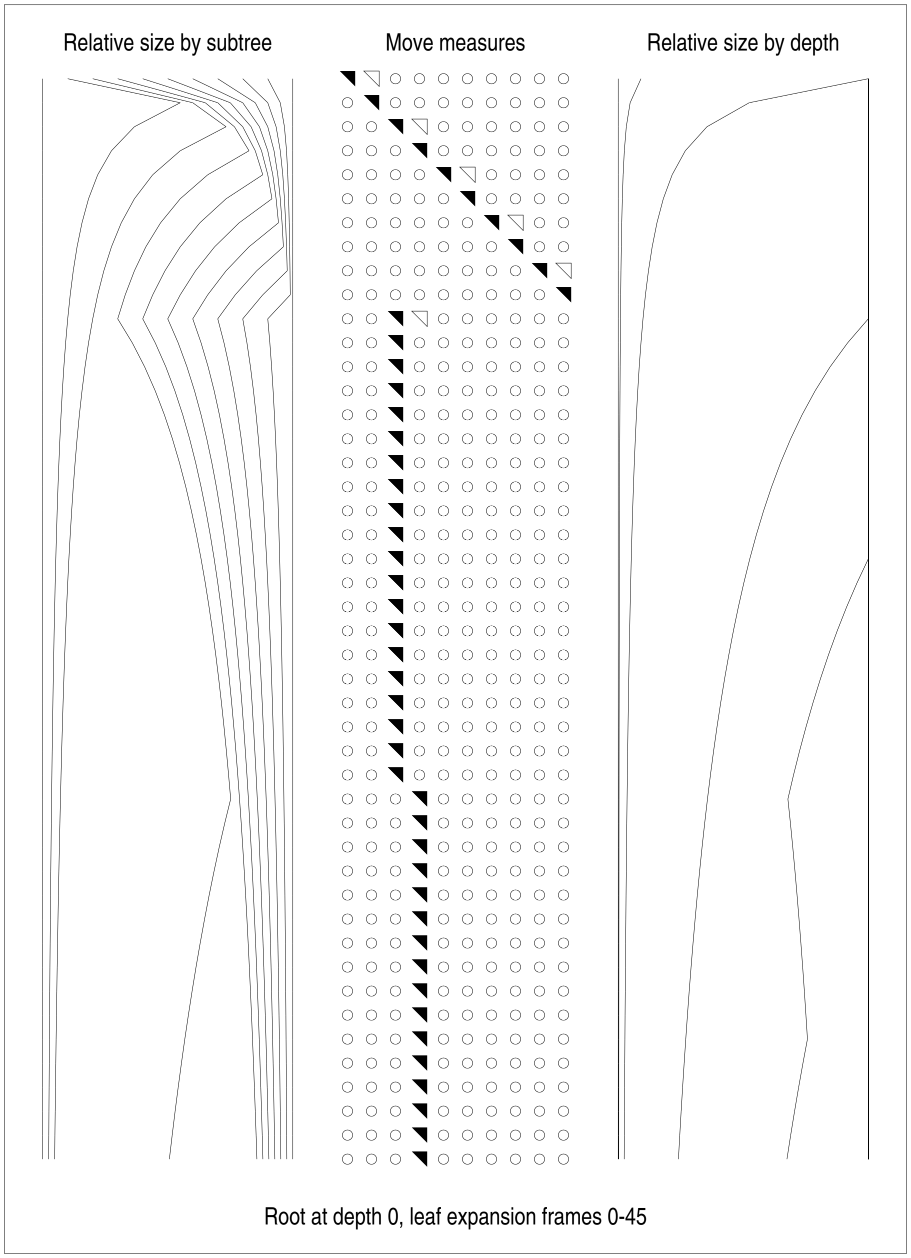

Figure 9. BFMM Expansions for Amazons

Table 2. Othello LCF versus BFMM Results

1 min		10 mins		30 mins		60 mins		Total	
LCF	5	LCF	8	LCF	3	LCF	5	LCF	21
BFMM	5	BFMM	2	BFMM	7	BFMM	4	BFMM	18

Similarly, Figure 9 shows the same information for BFMM. LCF expanded 250 leaves, and BFMM expanded 245 leaves, which is virtually identical. LCF explores principally child$_2$ and child$_3$ throughout this search, settling on child$_3$. BFMM explores a large variety throughout its 245 expansions, settling on child$_0$. These behaviors have ‘swapped’ in some sense from random trees,

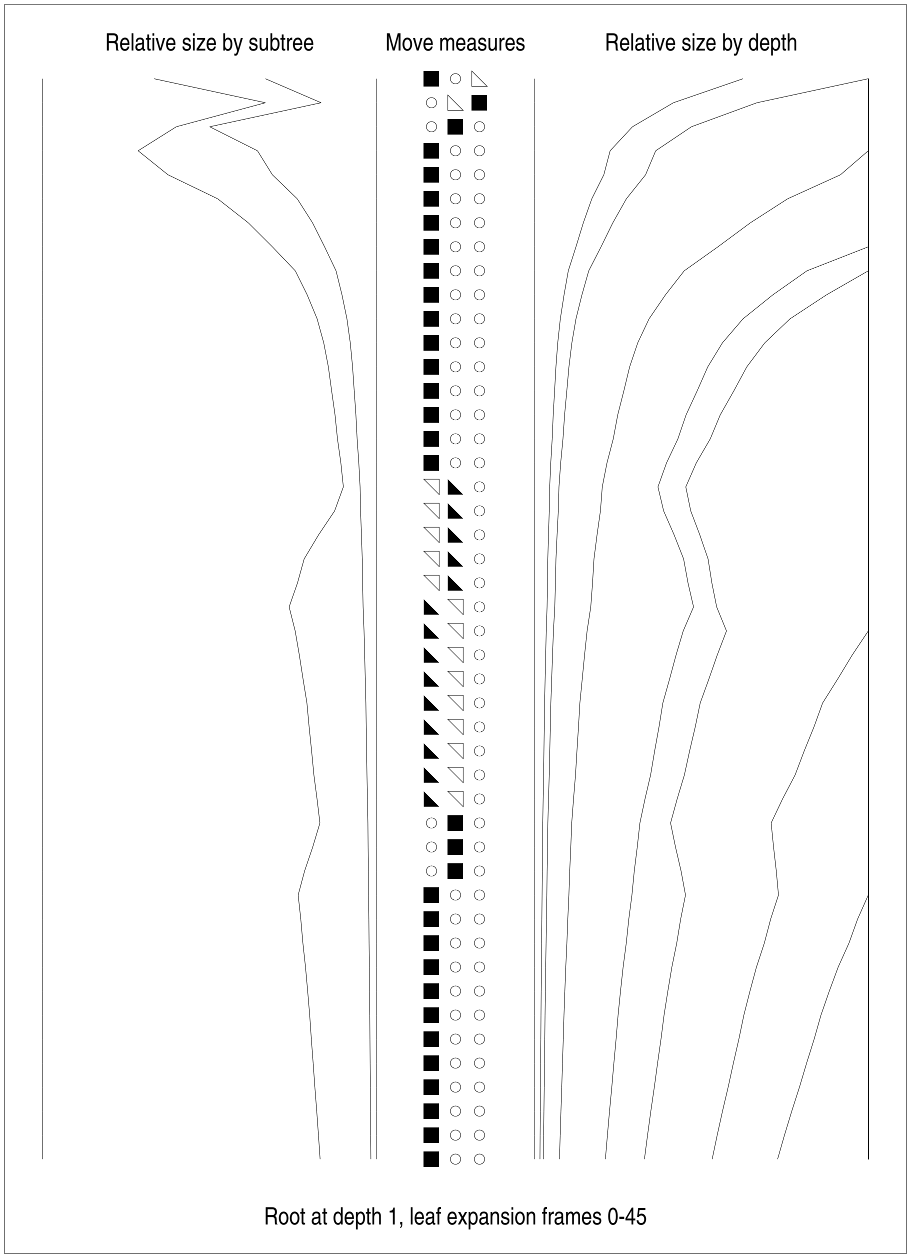

Figure 10. LCF Expansions for Othello

where LCF was the more varied of the two.

5.3 Othello

For the game of Othello, we also prepared LCF, BFMM, and AB versions that were identical except for the method of search. At a time allocation of one minute total per player for all moves of the game (very quick game), the LCF version won two of ten games against the AB version, and at all large time allocations lost all games. We did not pit BFMM against AB, but Korf & Chickering report their BFMM Othello program doing poorly against their AB Othello program. Regarding our LCF version against our BFMM version, we ran a large number of games at various time

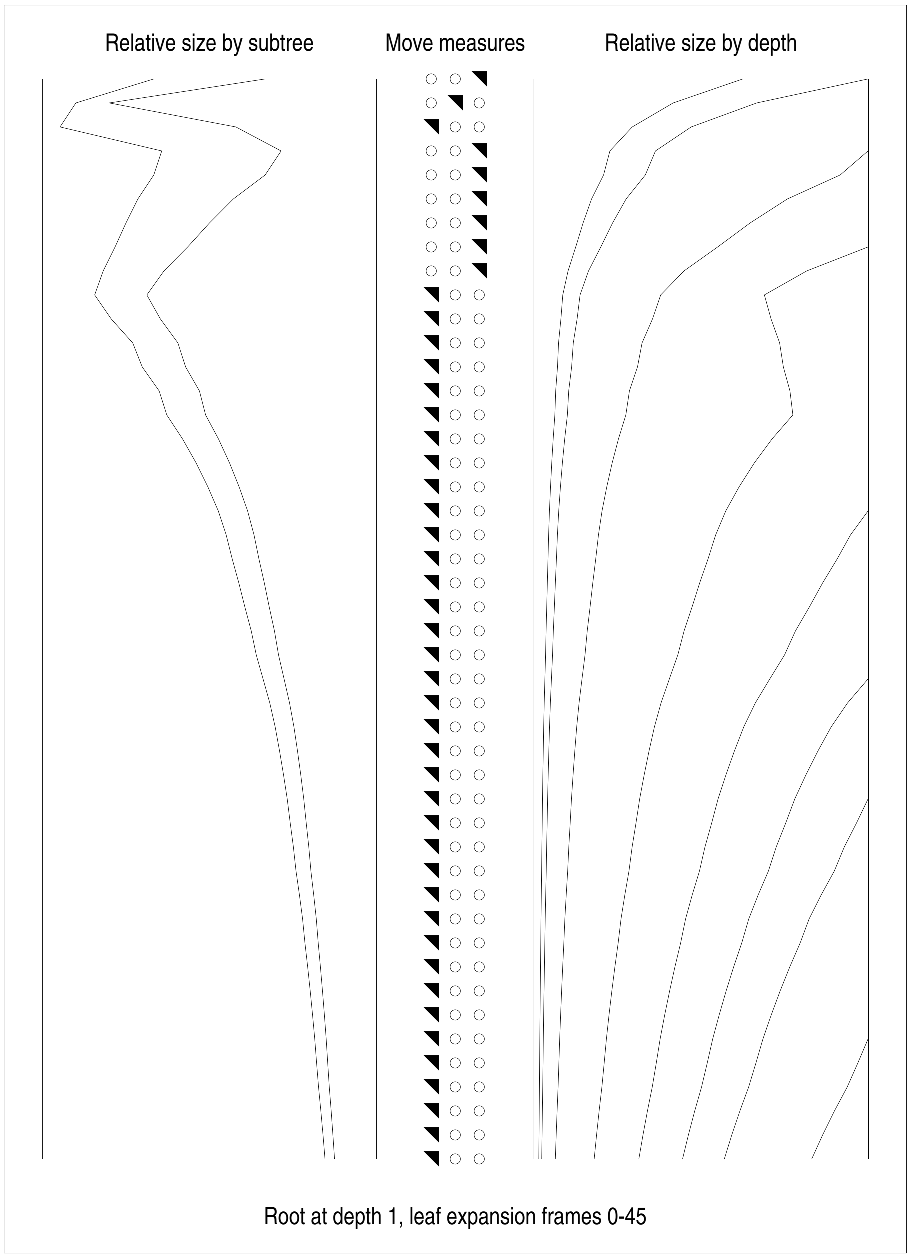

Figure 11. BFMM Expansions for Othello

allocations, as summarized in Table 2. The programs appear to be of fundamentally equal strength in this setting. A tenth game at the 60 minute time allocation was a draw, and is not included in the table.

Figure 10 shows the search behavior of LCF during the second move of a game of Othello, as does Figure 11 for BFMM. LCF explores the two openings that are well-regarded and shuns the third. BFMM latches onto one of the openings, exploring it at great depth. Examination of traces of this kind shows that both algorithms conduct searches that are quite narrow and deep.

6 Discussion

It is disappointing that LCF or BFMM do not produce play that is stronger than AB. (It appears that superior performance for random tree games is a special case.) It is naturally appealing to want to search those parts of the tree that are most likely to be informative. However, evaluation functions are imperfect, encoding a variety of strengths and weaknesses that collectively bias position assessment. To use such a function to guide search has a circular dependency. Imperfect value assessment implies imperfect relevance assessment. Searching those nodes that appear to bear on (are relevant to) the choice at the root is subject to the blind spots of the evaluation function, yet that is exactly what the search was intended to overcome.

7 Summary

We have presented a new heuristic for guiding best-first adversary search. The misordering assumption provides a basis for achieving goals in real-valued node evaluation. The notion of swing, and its relation to certainty, provides a new search heuristic. This approach was motivated by the work on conspiracy numbers, with the goal of eliminating the need to compute certainty with respect to an assortment of proposed game values. Although this goal has been achieved, the only evidence we have so far is that a player using the LCF algorithm in a real game (Amazons, Othello) will be weaker than a player using a minimax variant.

Acknowledgments

This material is based on work supported by the National Science Foundation under Grant IRI-9711239. Rich Korf shared his code for efficient indexing of a random number sequence. Gang Ding, David Stracuzzi, and Margaret Connell provided helpful comments.

Allis, L. V., van der Meulen, M., & van den Herik, H. J. (1994). Proof-number search. *Artificial Intelligence, 66*, 91-124.

Anantharaman, T., Campbell, M.S., & Hsu, F-h (1990). Singular extensions: Adding selectivity to brute-force searching. *Artificial Intelligence, 43*, 99-109.

Berliner, H. (1979). The B* tree search algorithm: A best-first proof procedure. *Artificial Intelligence, 12*, 23-40.

Berliner, H., & McConnell, C. (1996). B* probability based search. *Artificial Intelligence, 86*, 97-156.

Buro, M. (1997). Experiment with multi-probcut and a new high-quality evaluation function for Othello. *Proceedings of the Workshop on Game-Tree Search*. Princeton: NEC Research Institute.

Korf, R. E., & Chickering, D. M. (1996). Best-first minimax search. *Artificial Intelligence, 84*, 299-337.

McAllester, D. A. (1988). Conspiracy numbers for min-max search. *Artificial Intelligence, 35*, 287-310.

Palay, A. J. (1982). The B* tree search algorithm - new results. *Artificial Intelligence, 19*, 145-163.

Rivest, R. (1988). Game tree searching by min/max approximation. *Artificial Intelligence, 34*, 77-96.

Schaeffer, J. (1990). Conspiracy numbers. *Artificial Intelligence, 43*, 67-84.

Sutton, R. S. (1988). Learning to predict by the method of temporal differences. *Machine Learning, 3*, 9-44.