Best-First Minimax Search: First Results

Richard E. Korf
David Maxwell Chickering
Computer Science Department
University of California, Los Angeles
Los Angeles, Ca. 90024
(310)206-5383
korf@cs.ucla.edu

Abstract

We present a very simple selective minimax search algorithm for two-player games. It always expands next the frontier node at the end of the principal variation, or current best line of play, which is the node that determines the minimax value of the root. The algorithm requires no information other than a static evaluation function, and its time overhead per node is similar to that of alpha-beta minimax. On random game trees, our algorithm outperforms an efficient implementation of alpha-beta, giving both the same amount of computation. In the game of Othello, using the evaluation function from Bill, the world's best program, best-first minimax also outplays alpha-beta. We present an implementation of the algorithm that reduces its space complexity from exponential to linear in the search depth, at the cost of increased time complexity. Finally, we present a hybrid best-first extension algorithm that combines alpha-beta and best-first minimax, and performs significantly better than either pure algorithm in both domains.

1 Introduction and Overview

In chess, machines such as Deep-Thought[1] are competitive with the very best humans, but generate millions of positions per move. Their human opponents, however, only examine tens of positions, but search much deeper than the machine along some lines of play. Obviously, people are much more selective in their choice of positions to examine. The importance of selective search algorithms was first recognized by Shannon[2].

Most work on two-player search algorithms, however, has focussed on algorithms that make the same decisions as full-width, fixed-depth minimax, but do it more efficiently. This includes alpha-beta pruning[3], fixed and dynamic node ordering[4], SSS*[5], Scout[6], aspiration-windows[7], etc. In contrast, we define a selective search algorithm as one that makes different decisions than full-width, fixed-depth minimax. These include the B* algorithm[8], conspiracy search[9], min/max approximation[10], meta-greedy search[11], and singular extensions[12]. All these algorithms, except for singular extensions, require exponential memory. Furthermore, most of them have large time overheads per node expansion. In addition, B* and meta-greedy search require more information about a position than a single static evaluation. Singular extensions is the only algorithm to be successfully incorporated into a high-performance program[12]. If the best position at the search horizon is significantly better than its alternatives, the algorithm explores that position one ply deeper, and recursively applies the same rule at the next level.

We introduce a very simple selective search algorithm, called best-first minimax. It requires no more information than alpha-beta minimax, and its time overhead per node is roughly the same. Experimentally, the algorithm outperforms alpha-beta on random games and Othello. We also describe an implementation of the algorithm that reduces its space complexity from exponential to linear in the search depth. Finally, we explore best-first extensions, a hybrid combination of alpha-beta and best-first minimax that outperforms both algorithms.

2 Best-First Minimax Search

The basic idea of best-first minimax is to always explore further the current best line of play. Given a partially expanded game tree, with static evaluations of the leaf nodes, we compute the value if all interior MAX nodes as the maximum of the values of their children, and the values of interior MIN nodes as the minimum of their children's values. The minimax value of the root is equal to the static value of at least one leaf node, as well as every node on the path to that leaf node. This path is called the *principal variation*, and we call its leaf node the *principal leaf*. Best-first minimax always expands next the current principal leaf. The primary motivation behind this algorithm is that the principal leaf has the largest affect on the minimax value of the root, although not necessarily the largest affect on the move decision to be made.

Consider the example in figure 1, where squares represent MAX nodes and circles represent MIN nodes. Figure 1A shows the situation after the root node has been expanded. The values of the children are their static evaluations, and the value of the root is 6, the maximum of its children's values. This means that the right child is the principal leaf, and is expanded next, resulting in the situation in figure 1B. The new frontier nodes are statically evaluated at 5 and 2, and hence the value of their MIN parent changes to 2, the minimum of its children's values. This changes the value of the root to 4, the value of its left child. Thus, the left move now appears more promising, the left child of the root is the new principal leaf, and is expanded next, resulting in the situation in figure 1C. The value of the left child of the root changes to the minimum of its children's static values, 1, and the value of the root changes to the maximum of its children's values, 2. At this point, attention shifts back to the right move, and the rightmost grandchild of the root is expanded next, as shown in figure 1D.

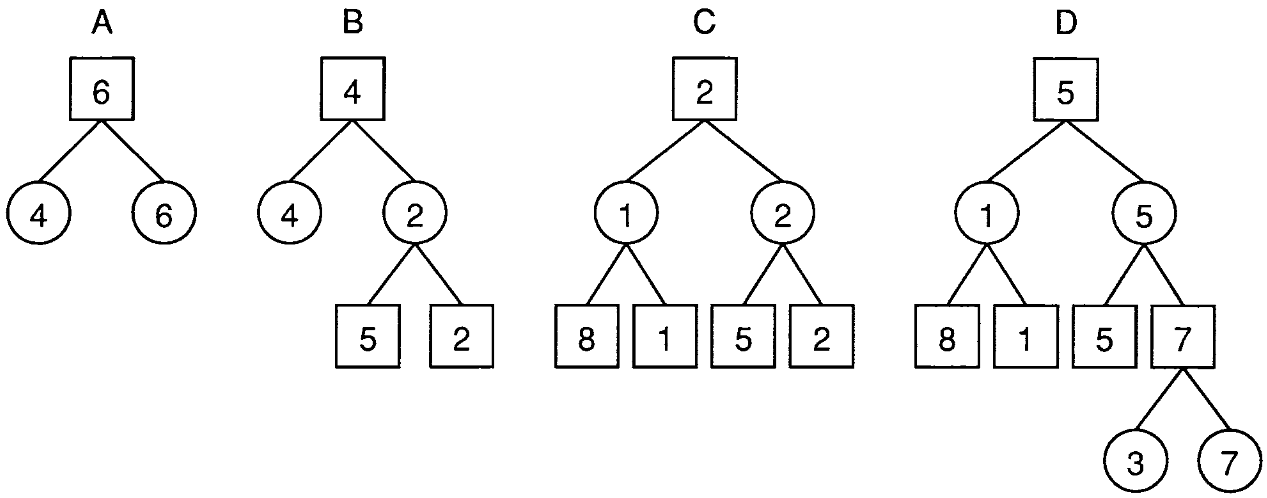

Figure 1: Best-first minimax search example

Best-first minimax always expands next the principal leaf. While this may appear to lead to the exploration of a single path to the exclusion of all others, this does not occur in practice. The expansion of any node tends to make it look worse, thus inhibiting further exploration of the subtree below the node. For example, a MAX leaf node will only be expanded if its static value is the minimum among the children of its MIN parent. Expanding it changes its value to the maximum value of its children, which tends to increase its value, making it unlikely to remain as the minimum among its brothers. Similarly, MIN nodes also tend to appear worse to their MAX parents when expanded, making it less likely that their children will be expanded next. Thus, this tempo effect adds balance to the tree searched by best-first minimax. This effect tends to increase with increasing branching factor. Interestingly, while this oscillation in values with the last player to move is one reason that alpha-beta avoids comparing evaluations at different levels in the tree, it turns out to be advantageous to best-first minimax.

While in principle best-first minimax could make a move at any point in time, we choose to move when the length of the principal variation exceeds a given depth bound, or a winning terminal node is chosen for expansion. This ensures that the chosen move has either been explored to a significant depth, or leads to a win.

The obvious implementation of best-first minimax maintains the entire current search tree in memory. When a node is expanded, its children are evaluated, its value is updated, and the algorithm moves up the tree updating the values of its ancestors, until it either reaches the root, or a node whose value doesn't change. It then moves down the tree to a maximum-valued child of a MAX node, or a minimum-valued child of a MIN node, until it reaches a new frontier node to expand next. The most significant drawback of this implementation is that it requires memory that is exponential in the tree depth.

In spite of its simplicity, we couldn't find this algorithm in the literature. A related algorithm is AO*[13], a best-first search of a single-agent AND-OR tree. The main difference is that in an AND-OR tree, the cost of a node is either the minimum of its children's values (OR node), or the sum of its child values (AND node), as opposed to the alternating minimizing and maximizing levels of a game tree. The closest algorithm to best-first minimax is

Rivest's min/max approximation[10]. Both algorithms strive to expand next the node with the largest affect on the root value. The main difference is that best-first minimax is much simpler than min/max approximation. Note that both AO* and min/max approximation also require exponential memory.

3 Recursive Best-First Minimax Search

Recursive Best-First Minimax Search (RBFMS) is an implementation of best-first minimax that runs in space linear in the maximum search depth. The algorithm is a generalization of Simple Recursive Best-First Search(SRBFS)[14], a linear-space best-first search designed for single-agent problems. Figure 2 shows the behavior of RBFMS on the example of figure 1.

Associated with each node on the principal variation is a lower bound Alpha, and an upper bound Beta, similar to the corresponding bounds in alpha-beta minimax. A given node will remain on the principal variation as long as its backed-up minimax value remains within these bounds. The root is bounded by $-\infty$ and ∞, since it is always on the principal variation. Figure 2A shows the situation after the root is expanded, with the right child on the principal variation. It will remain on the principal variation as long as its minimax value is greater than or equal to the maximum value of its brothers (4). The right child is expanded next, resulting in the situation in figure 2B.

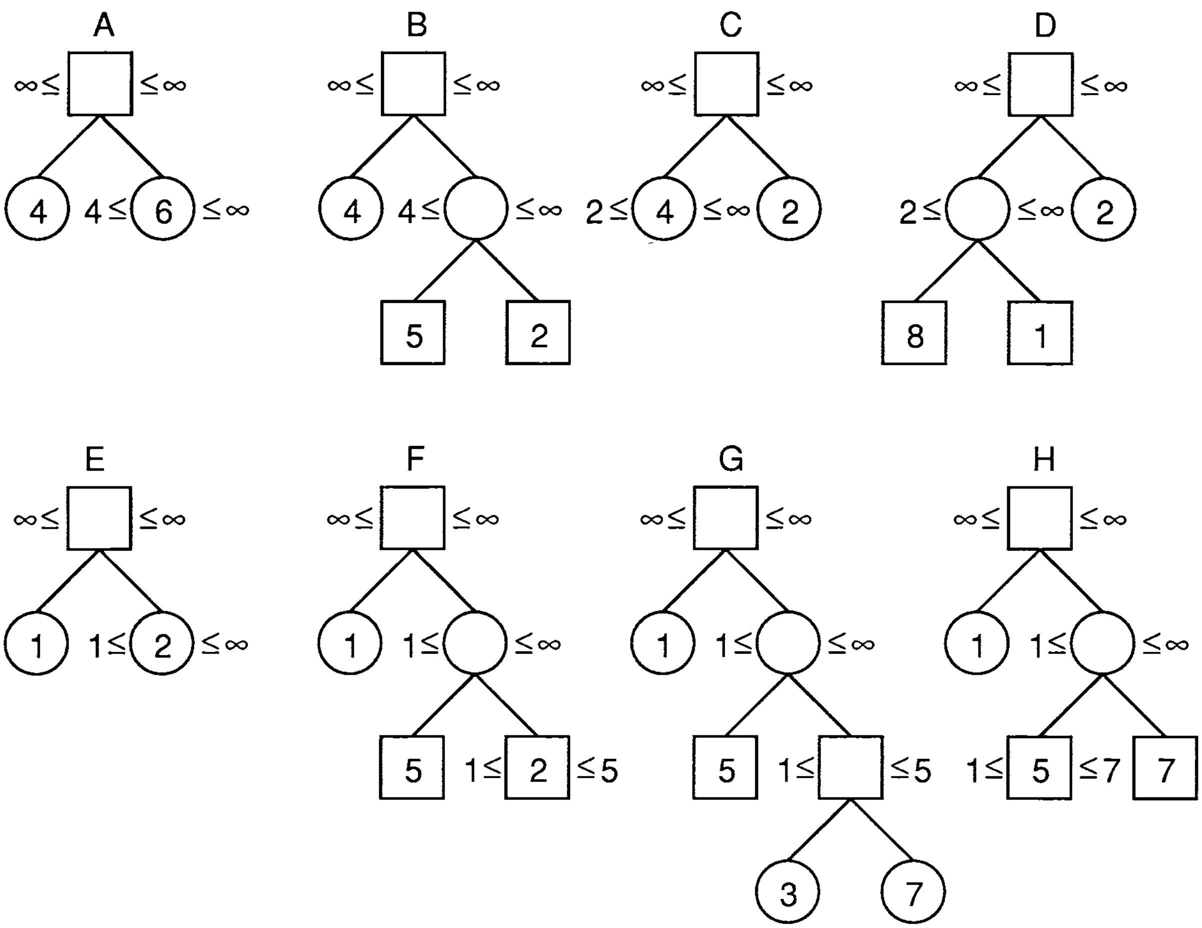

Figure 2: Recursive best-first minimax search example

At this point, the value of the right child becomes the minimum of the values of its children (5 and 2), and since 2 is less than the lower bound of 4, the right child of the root is no longer on the principal variation, and the left child is the new principal leaf. The algorithm returns to the root, freeing memory, but stores with the right child its new minimax value of 2, resulting in the situation in figure 2C.

The left child of the root will now remain on the principal variation as long as its value is greater than or equal to 2, the largest value among its brothers. It is expanded, resulting in the situation in figure 2D. Its new value is the minimum of its children's values (8 and 1), and since 1 is less than the lower bound of 2, the left child is no longer on the principal variation, and the right child of the root becomes the new principal leaf. The algorithm returns to the root, and stores the new minimax value of 1 with the left child, resulting in the situation in figure 2E. Now, the right child of the root will remain on the principal variation as long as its minimax value is greater than or equal to 1, the value of its best brother, and is expanded next. The reader is encouraged to follow the rest of the example. The values of interior nodes on the principal variation are not computed until necessary.

RBFMS consists of two recursive, symmetric functions, one for MAX and one for MIN. Each takes three arguments: a node, a lower bound Alpha, and an upper bound Beta. Together they perform a best-first minimax search of the subtree below the node, as long as its backed-up minimax value remains within the Alpha and Beta bounds. Once it exceeds those bounds, the function returns the new minimax value of the node. At any point, the recursion stack contains the current principal variation, plus the immediate brothers of all nodes on this path. Its space complexity is thus $O(bd)$, where b is the maximum branching factor of the tree, and d is the maximum depth.

```
BFMAX (Node, Alpha, Beta)
FOR each Child[i] of Node
    M[i] := Evaluation(Child[i])
   IF M[i] > Beta return M[i]
sort Child[i] and M[i] in decreasing order of M[i]
IF only one child, M[2] := -infinity
WHILE Alpha <= M[1] <= Beta
    M[1] := BFMIN(Child[1], max(Alpha, M[2]), Beta)
   insert Child[1] and M[1] in sorted order
return M[1]

BFMIN (Node, Alpha, Beta)
FOR each Child[i] of Node
    M[i] := Evaluation(Child[i])
   IF M[i] < Alpha return M[i]
sort Child[i] and M[i] in increasing order of M[i]
IF only one child, M[2] := infinity
WHILE Alpha <= M[1] <= Beta
    M[1] := BFMAX(Child[1], Alpha, min(Beta, M[2]))
   insert Child[1] and M[1] in sorted order
return M[1]
```

When a node is expanded, its children are generated and evaluated one at a time. If the value of any child of a MAX node exceeds the upper bound of Beta, the child's value is immediately returned, and the remaining children are not generated. Similarly, if the value of any child of a MIN node is less than the lower bound of Alpha, the child's value is returned without generating the remaining children.

For single-agent problems, there are two linear-space best-first search algorithms, Simple Recursive Best-First Search (SRBFS), and the much more efficient Recursive Best-First Search (RBFS). Surprisingly, the two-player generalizations of these two algorithms behave identically, so we present only the simpler version.

Syntactically, recursive best-first minimax appears very similar to alpha-beta minimax, but behaves quite differently. In particular, alpha-beta makes its move decisions solely on the basis of the static values of nodes at the search horizon, whereas best-first minimax relies on the values of nodes at different levels in the tree.

4 Saving the Tree

The cost of reducing the space complexity of best-first minimax search from exponential to linear is that some nodes are regenerated multiple times. This overhead is significant for deep searches. On the other hand, the time per node generation for RBFMS is significantly less than that of standard best-first search. In the standard implementation, when a new node is generated, the entire board position of its parent node is copied, along with the changes made to it. The recursive algorithm, however, does not copy the board, but only makes incremental changes to a single copy, and undoes them when backtracking.

Our actual implementation retains the recursive control structure of RBFMS, with only incremental changes to a single copy of the board. When backing up the tree, however, the subtree is retained in memory. Thus, when a path is abandoned and then reexplored, the entire subtree need not be regenerated, but only the nodes on the path to the new principal leaf, eliminating most node regenerations and all node reevaluations. While this requires exponential space, it is not a problem in practice, for several reasons.

The main reason is that once a move is made, and the opponent moves, best-first minimax saves only the subtree that is still relevant, pruning the subtrees below all moves it didn't choose, and those moves not chosen by the opponent. This releases the corresponding memory, and the search for the next move then starts from the remaining subtree. While current machines can fill their memories in a matter of minutes, in a two-player game, moves must be made every few minutes, freeing much of the memory.

A second reason that memory is not a serious constraint is that all that needs to be stored at a node is the backed-up minimax value of the node, and pointers to its children. The actual board itself, and all other information, is incrementally generated from its parent. A node in our implementation only requires three words of memory, an integer and two pointers. This could be reduced to two words by representing a leaf node as just an integer.

If memory is exhausted in the course of computing a move, there are two options. One is to switch to the linear-space algorithm, thus requiring no more memory than the recursion stack. The other is to prune the least promising parts of the current tree in memory. Since all nodes off the principal variation have their backed-up minimax values stored at all times, pruning is simply a matter of recursively freeing the memory in a given subtree.

Since best-first minimax spends most of its effort on the expected line of play, it can save much of the information collected for one move, and apply it to subsequent moves, particularly if the opponent makes the expected move. This saving of the tree between moves improves the performance of best-first minimax considerably. In contrast, the standard depth-first implementation of alpha-beta minimax doesn't save any of the tree from one move to the next. It essentially computes each new move from scratch. Even when alpha-beta is modified to save the relevant subtree from one move to the next, since it searches every move to the same depth, relatively little of the subtree computed during one move is still relevant after the player's move and the opponent's response. In particular, it can be shown that even in the best case, when the minimal tree is searched and the opponent makes the expected move, only $1/b$ of the tree that is generated by alpha-beta in making one move will still be relevant after the move and the opponent's move, where b is the branching factor of the tree.

5 Experimental Results

The test of a selective search algorithm is how well it plays. We played best-first minimax against alpha-beta minimax, both in Othello[16, 17] and on random game trees[18], giving both algorithms the same amount of computation.

In a uniform random game tree with branching factor b and depth d, each edge is independently assigned a random cost. The static value of a node is the sum of the edge costs from the root to the node. In order not to favor MAX or MIN, the edge-cost distribution is symmetric around zero. Figure 3 shows a sample uniform binary random game tree with edge costs chosen from -9 to 9. Since real games do not have uniform branching factors, in our random games the number of children of a given node is a random variable uniformly distributed between one and the maximum branching factor B. Our edge-cost distribution is uniform from -2^{15} to 2^{15}.

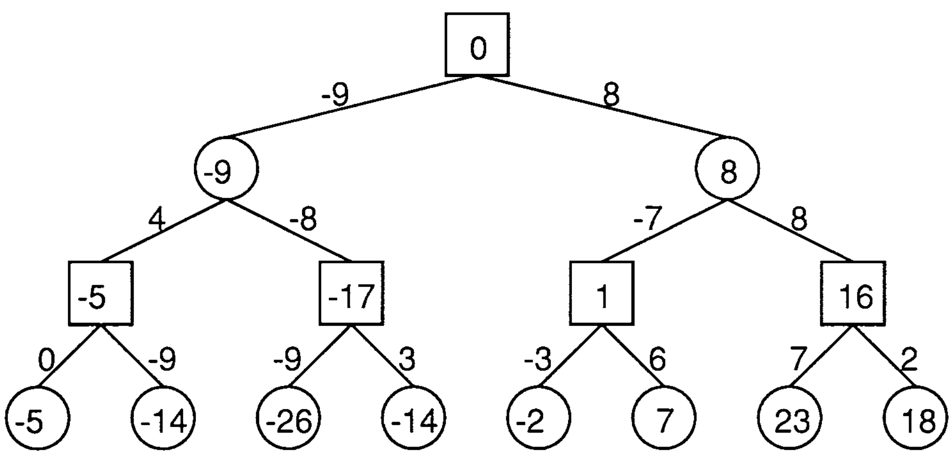

Figure 3: Sample uniform random game tree with b=2, d=3, and edge costs from [-9,9]

In generating a particular random tree, it is important that the same cost always be assigned to the same edge of the tree. The simplest way to do this is to assign a unique integer to each edge of the tree, and to use that integer as the seed for the random number generator at that point[8]. Unfortunately, this produces more highly

AB depth	1	2	3	4	5	6
BF depth	1	4	8	12	15	19
BF wins	50%	66%	73%	72%	57%	61%

Table 1: Winning percentage of pure best-first minimax vs. alpha-beta on Othello

correlated values than assigning successive random numbers to the tree edges in a breadth-first fashion, the scheme we use. Doing this efficiently, however, requires a more complex calculation, which space restrictions prevent us from detailing here.

One virtue of the random game tree is that it allows us to control the branching factor and depth of the game. More importantly, however, the random game is simple enough to allow other researchers to reproduce our results. Conversely, real game programs are complex, and good evaluation functions are hard to find. We obtained the evaluation function for Bill[17], the world's best Othello player, from Kai-Fu Lee, one of its authors.

The efficiency of alpha-beta is greatly affected by the order in which nodes are searched. The simplest way to achieve good performance, called fixed ordering[4], is to fully expand each node, statically evaluate each child, sort the children, and then search the children of MAX nodes in decreasing order of their values, and the children of MIN nodes in increasing order. We use fixed ordering on newly generated nodes until alpha-beta reaches one level above the search horizon. At that point, since there is no advantage to ordering the nodes on the horizon, the children are evaluated one at a time, allowing additional pruning. To ensure a fair comparison to best-first minimax, our alpha-beta implementation saves the relevant subtree from one move to the next. This allows us to order previously generated nodes by their backed-up values rather than their static values, further improving the node ordering and performance of alpha-beta. In addition, the previously generated nodes need not be reevaluated.

Each game was played twice, with each algorithm alternately moving first, to eliminate the effect of a particular initial state favoring the first or second player to move. An Othello game is won by the player with the most discs when no further moves are possible. A random game ends when a terminal position is reached, 100 moves in our experiments, and returns the value of the final position. Given a pair of random games, and the corresponding terminal values reached, the winner is the algorithm that played MAX when the larger terminal value was obtained. Each random game tournament consisted of 100 pairs of games, and each Othello tournament consisted of 244 pairs of games. Different random games were generated from different random seeds, from 1 to 100, while different Othello games were generated by making all possible first four moves, and starting the game with the fifth move. A small number of both Othello and random games were tied, but these are ignored in the results presented below.

When the alpha-beta search horizon is sufficient to reach the end of the game, both algorithms use alpha-beta to complete the game, since alpha-beta is optimal then the static values are exact. In Othello, the disc differential is used as the exact value in searching the endgame, while the sum of the edge values becomes the exact value at the end of a random game.

Since best-first searches deeper than alpha-beta in the same amount of time, it reaches the end of the game before alpha-beta does. Since disc differentials are not comparable to the values returned by Bill's evaluation function, Othello endgame positions are evaluated at $-\infty$ if MAX has lost, ∞ if MAX has won, and $-\infty + 1$ in the case of a tie. If the principal leaf node is a terminal node, and a won position for best-first, it stops searching and makes a move. If alpha-beta then makes the expected response, the principal leaf won't change, and best-first will make its next move without further search. Conversely, if the principal leaf is a lost or tied position, best-first will continue to search until it finds a won position, or runs out of time. While this endgame play is not ideal and could be improved, it is the most faithful to the original algorithm.

For each alpha-beta horizon, we experimentally determined what depth limit caused best-first to take most nearly the same amount of time as alpha-beta. This is done by running a series of tournaments, and incrementing the search horizon of the algorithm that took less time in the last tournament. Node evaluation is the dominant cost for both algorithms in both games, and running time is roughly proportional to the number of node evaluations.

Table 1 shows the results of the Othello experiments. The top line shows the different alpha-beta search depths, and the second line shows the best-first search depth that took most nearly the same amount of time as the corresponding alpha-beta depth. The third line shows the percentage of games that were won by best-first minimax, excluding tie games. Each data point is an average of 244 pair of games, or 488 total games.

Both algorithms are identical at depth one. At greater depths, best-first searches much deeper than alpha-beta, and wins most of the time. The winning percentage increases to 73%, and then begins to drop off at greater depths.

Table 2 shows the corresponding results for random game trees of various random branching factors. The branch-

AB depth	1	2	3	4	5	6	7	8	9	10
B=2, BF depth	1	3	6	9	13	17	22	26	33	38
B=2, BF wins	50%	59%	74%	80%	76%	85%	83%	76%	82%	80%
B=3, BF depth	1	3	7	10	14	19	26	33	42	52
B=3, BF wins	50%	65%	89%	88%	80%	88%	84%	76%	77%	65%
B=4, BF depth	1	3	7	11	16	21	30	38	51	64
B=4, BF wins	50%	68%	91%	85%	85%	86%	81%	63%	70%	54%
B=5, BF depth	1	3	7	11	17	23	33	43	59	76
B=5, BF wins	50%	66%	92%	92%	83%	81%	81%	65%	45%	40%
B=10, BF depth	1	3	8	13	22	31	48			
B=10, BF wins	50%	65%	97%	80%	76%	60%	46%			
B=20, BF depth	1	3	9	16	28					
B=20, BF wins	50%	66%	97%	86%	54%					

Table 2: Winning percentage of pure best-first minimax vs. alpha-beta on random games

ing factor of an individual node is a random variable uniformly distributed from one to B, for an average branching factor of (B+1)/2. Each data point is the average of 100 pairs of games, or 200 total games.

Again, we see that best-first significantly outperforms alpha-beta, up to a point. The performance of best-first search tends to degrade when the gap between the alpha-beta and best-first horizons becomes very large, and in some cases alpha-beta outperforms best-first search. Note that the entire game is only 100 moves long.

6 Best-First Extension: A Hybrid Alpha-Beta/Best-First Algorithm

In both the Othello and random game experiments, the relative performance of best-first minimax degrades when the difference between the alpha-beta and best-first search horizons becomes excessive. One explanation is that while best-first minimax evaluates every child of the root, it may not generate some grandchildren of the root at all, depending on the static values of the children. In particular, if the evaluation function grossly underestimates the value of a child of the root from the perspective of the parent, it may never be expanded. For example, this would occur in a piece trade whose first move is a sacrifice. At some point, it makes more sense to consider grandchildren of the root instead of a node 30 or more moves down the principal variation.

To correct this, we implemented a hybrid algorithm, called best-first extension, that combines the coverage of alpha-beta with the penetration of best-first minimax. Best-first extension performs alpha-beta to a shallow search horizon, and then executes best-first search to a much greater depth, starting with the tree, backed-up values, and principal variation generated by the alpha-beta search. This guarantees that every move will be explored to a minimum depth, regardless of its static evaluation, before exploring the most promising moves much deeper. This is similar to the approach taken by singular extension search[12].

Best-first extension has two parameters: the depth of the initial alpha-beta search, and the depth of the following best-first search. In our initial experiments, the alpha-beta horizon of the initial search was set to one less than the horizon of its pure alpha-beta opponent, and the best-first horizon is whatever depth takes the same total amount of time, including the alpha-beta search, as the pure alpha-beta opponent. Even in this case, most of the time is spent on the best-first extension. Table 3 shows the results for Othello, in the same format as table 1, and Table 4 shows the results for the random game trees. Against alpha-beta depths 1 and 2, best-first extension is identical to pure best-first minimax, since it always evaluates all nodes at depth 1. At greater depths, however, the results for best-first extension are significantly better than for pure best-first minimax. In both games, best-first extension outperforms alpha-beta in every tournament.

AB depth	1	2	3	4	5	6	7
BF depth	1	4	7	10	14	18	21
BF wins	50%	66%	83%	79%	64%	72%	69%

Table 3: Winning percentage of best-first extension vs. alpha-beta on Othello

AB depth	1	2	3	4	5	6	7	8	9	10
B=2, BF depth	1	3	6	7	10	12	15	17	21	24
B=2, BF wins	50%	59%	68%	81%	75%	82%	79%	78%	80%	78%
B=3, BF depth	1	3	6	8	11	14	19	22	29	35
B=3, BF wins	50%	65%	91%	86%	85%	78%	90%	82%	88%	81%
B=4, BF depth	1	3	7	9	12	16	23	27	38	45
B=4, BF wins	50%	68%	98%	88%	90%	91%	84%	81%	87%	84%
B=5, BF depth	1	3	7	9	14	18	26	32	46	56
B=5, BF wins	50%	66%	95%	89%	89%	89%	83%	85%	87%	75%
B=10, BF depth	1	3	8	11	19	25	41			
B=10, BF wins	50%	65%	99%	95%	92%	84%	82%			
B=20, BF depth	1	3	9	13	25					
B=20, BF wins	50%	66%	98%	95%	88%					

Table 4: Winning percentage of best-first extension vs. alpha-beta on random games

7 Conclusions and Further Work

We presented a very simple selective search algorithm, best-first minimax. It always expands next the frontier node at the end of the current principal variation, which is the node that determines the minimax value of the root. An important advantage of the algorithm is that it can save most of the results from one move computation, and apply them to subsequent moves. In experiments on random games and on Othello, best-first minimax outplays alpha-beta, giving both algorithms the same amount of computation, up to a given search depth. We also presented a hybrid combination of best-first minimax and alpha-beta, which guarantees that every move is searched to a minimum depth. This best-first extension outperforms both algorithms in both games, defeating alpha-beta in every tournament. While memory capacity was not a limiting factor in our experiments, we also showed how to reduce the space complexity of the algorithm from exponential to linear in the search depth, but at significant cost in nodes generated for deep searches.

Since pure best-first minimax performs best against relatively shallow alpha-beta searches, it is likely to be most valuable in games with large branching factors, and/or expensive evaluation functions. These are the games, such as Go, in which computers have been least successful against humans. Current research is focussed on implementing singular extensions in an attempt to improve our alpha-beta opponent, and implementations on other games such as chess.

8 Acknowledgements

Thanks to Kai-Fu Lee for providing the sources to Bill, to Stuart Russell for providing an initial Othello evaluation function, to Milos Ercegovac and Peter Montgomery for technical assistance in efficiently generating random trees, to Joe Pemberton and Weixiong Zhang for discussions concerning this research, and to Valerie Aylett for drawing the figures. This research was supported by NSF Grant No. IRI-9119825, and a grant from Rockwell International.

References

[1] Hsu, F.-H., T. Anantharaman, M. Campbell, and A. Nowatzyk, A grandmaster chess machine, *Scientific American*, Vol. 263, No. 4, Oct. 1990, pp. 44-50.

[2] Shannon, C.E., Programming a computer for playing chess, *Philosophical Magazine*, Vol. 41, 1950, pp. 256-275.

[3] Knuth, D.E., and R.E. Moore, An analysis of Alpha-Beta pruning, *Artificial Intelligence*, Vol. 6, No. 4, 1975, pp. 293-326.

[4] Slagle, J.R., and Dixon, J.K., Experiments with some programs that search game trees, J.A.C.M., Vol. 16, No. 2, 1969, pp. 189-207.

[5] Stockman, G., A minimax algorithm better than Alpha-Beta? *Artificial Intelligence*, Vol. 12, No. 2, 1979, pp. 179-96.

[6] Pearl, J. *Heuristics*, Addison-Wesley, Reading, Mass, 1984.

[7] Kaindl, H., R. Shams, and H. Horacek, Minimax search algorithms with and without aspiration windows, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 13, No. 12 December 1991, pp. 1225-1235.

[8] Berliner, H.J., The B* tree search algorithm: a best-first proof procedure, *Artificial Intelligence*, Vol. 12, 1979, pp. 23-40.

[9] McAllester, D.A., Conspiracy numbers for min-max search, *Artificial Intelligence*, Vol. 35, No. 3, 1988, pp. 287-310.

[10] Rivest, R.L., Game tree searching by min/max approximation, *Artificial Intelligence*, Vol. 34, No. 1, 1986, pp. 77-96.

[11] Russell, S., and E. Wefald, On optimal game-tree search using rational meta-reasoning, *Proceedings of the Eleventh International Conference on Artificial Intelligence (IJCAI-89)*, Detroit, Michigan, August 1989, pp. 334-340.

[12] Anantharaman, T., M.S. Campbell, and F.-H. Hsu, Singular extensions: Adding selectivity to brute-force searching, *Artificial Intelligence*, Vol. 43, No. 1, April, 1990, pp. 99-109.

[13] Nilsson, N.J., *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, New York, 1971.

[14] Korf, R.E., Linear-space best-first search, *Artificial Intelligence*, Vol. 62, No. 1, July 1993, pp. 41-78.

[15] Korf, R.E., Best-first minimax search: Initial results, Computer Science Department Technical Report, No. CSD-920021, University of California, Los Angeles, Ca. 90024, January, 1992.

[16] Rosenbloom, P.S., A world-championship-level Othello program, *Artificial Intelligence*, Vol. 19, 1982, pp. 279-320.

[17] Lee, K.-F. and S. Mahajan, The development of a world-class Othello program, *Artificial Intelligence*, Vol 43, No. 1, 1990, pp. 21-36.

[18] Fuller, S.H., J.G. Gaschnig, and J.J. Gillogly, An analysis of the alpha-beta pruning algorithm, Dept. of Computer Science Technical Report, Carnegie-Mellon University, Pittsburgh, Pa., 1973.