Best-First Minimax Search: Othello Results

Richard E. Korf and David Maxwell Chickering
Computer Science Department
University of California, Los Angeles
Los Angeles, Ca. 90024
korf@cs.ucla.edu

Abstract

We present a very simple selective search algorithm for two-player games. It always expands next the frontier 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. We also 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. In the game of Othello, using the evaluation function from Bill (Lee & Mahajan 1990), best-first minimax outplays alpha-beta at moderate depths. A hybrid best-first extension algorithm, which combines alpha-beta and best-first minimax, performs significantly better than either pure algorithm even at greater depths. Similar results were also obtained for a class of random game trees.

Introduction and Overview

The best chess machines are competitive with the best humans, but generate millions of positions per move. Their human opponents, however, only examine tens of positions, but search much deeper along some lines of play. Obviously, people are more selective in their choice of positions to examine. The importance of selective search was first recognized by (Shannon 1950).

Most work on game-tree search has focussed on algorithms that make the same decisions as full-width, fixed-depth minimax. This includes alpha-beta pruning (Knuth & Moore 1975), fixed and dynamic node ordering (Slagle & Dixon 1969), SSS* (Stockman 1979), Scout (Pearl 1984), aspiration-windows (Kaindl, Shams, & Horacek 1991), etc. We define a selective search algorithm as one that makes different decisions than full-width, fixed-depth minimax. These include B* (Berliner 1979), conspiracy search (McAllester 1988), min/max approximation (Rivest 1987), meta-greedy search (Russell & Wefald 1989), and singular extensions (Anantharaman, Campbell, & Hsu 1990). All of these algorithms, except singular extensions, require exponential memory, and most have large time overheads per node expansion. In addition, B* and meta-greedy search require more information than a single static evaluation function. Singular extensions is the only algorithm to be successfully incorporated into a high-performance program.

We describe a very simple selective search algorithm, called best-first minimax. It requires only a single static evaluator, and its time overhead per node is roughly the same as alpha-beta minimax. We describe an implementation of the algorithm that reduces its space complexity from exponential to linear in the search depth. We also explore best-first extensions, a hybrid combination of alpha-beta and best-first minimax. Experimentally, best-first extensions outperform alpha-beta in the game of Othello, and on a class of random game trees. Earlier reports on this work include (Korf 1992) and (Korf & Chickering 1993).

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, the value of an interior MAX node is the maximum of its children's values, and the value of an interior MIN node is the minimum of its children's values. There exists a path, called the *principal variation*, from the root to a leaf node, in which every node has the same value. This leaf node, whose evaluation determines the minimax value of the root, is called the *principal leaf*. Best-first minimax always expands next the current principal leaf node, since it has the greatest affect on the minimax value of the root.

Consider the example in figure 1, where squares represent MAX nodes and circles represent MIN nodes. Figure 1A shows the situation after the root has been expanded. The values of the children are their static values, and the value of the root is 6, the maximum of its children's values. Thus, 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 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 maximum of its children's values. Thus, the left child of the root is the new principal leaf, and is expanded next, result-

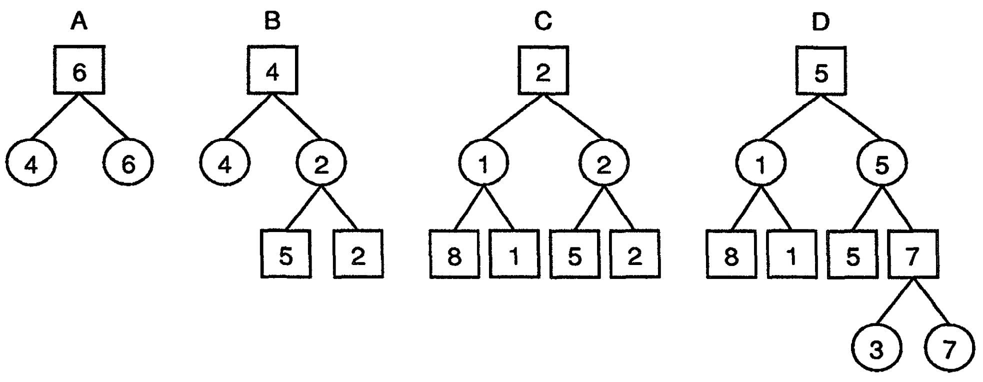

Figure 1: Best-first minimax search example

ing in the situation in figure 1C. The value of the left child of the root changes to the minimum of its children's values, 1, and the value of the root changes to the maximum of its children's values, 2. At this point, the rightmost grandchild is the new principal leaf, and is expanded next, as shown in figure 1D.

By always expanding the principal leaf, best-first minimax may appear to suffer from the exploration of a single path to the exclusion of all others. This does not occur in practice, however. The reason is that the expansion of a node tends to make it look worse, thus inhibiting further exploration of the subtree below it. For example, a MAX node will only be expanded if its static value is the minimum among its brothers, since its parent is a MIN node. Expanding it changes its value to the maximum of its children, which tends to increase its value, making it less likely to remain as the minimum among its siblings. 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. This *tempo* effect adds balance to the tree searched by best-first minimax, and increases with increasing branching factor. Surprisingly, while this oscillation in values with the last player to move is the reason that alpha-beta avoids comparing nodes 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 been explored to a significant depth, or leads to a win.

The simplest implementation of best-first minimax maintains the current 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 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 principal leaf. A drawback of this implementation is that it requires exponential memory, a problem that we address below.

Despite its simplicity, best-first minimax has apparently not been explored before. The algorithm is mentioned as a special case of AO*, a best-first search of an AND-OR tree, in (Nilsson 1969). The chess algorithm of (Kozdrowicki & Cooper 1973) seems related, but behaves differently on their examples. Best-first minimax is also related to conspiracy search (McAllester 1988), and only expands nodes in the conspiracy set. It is also related to Rivest's min/max approximation (Rivest 1987). Both algorithms strive to expand next the node with the largest affect on the root value, but best-first minimax is much simpler. All four related algorithms above require exponential memory.

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 search depth. The algorithm is a generalization of Simple Recursive Best-First Search (SRBFS) (Korf, 1993), 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 bounds in alpha-beta pruning. A node will remain on the principal variation as long as its minimax value stays within these bounds. The root is bounded by $-\infty$ and ∞. 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 siblings (4). The right child is expanded next, as shown in figure 2B.

The value of the right child changes to the minimum of its children's values (5 and 2), and since 2 is less than the lower bound of 4, the right child is no longer on the principal variation, and the left child of the root is the new principal leaf. The algorithm returns to the

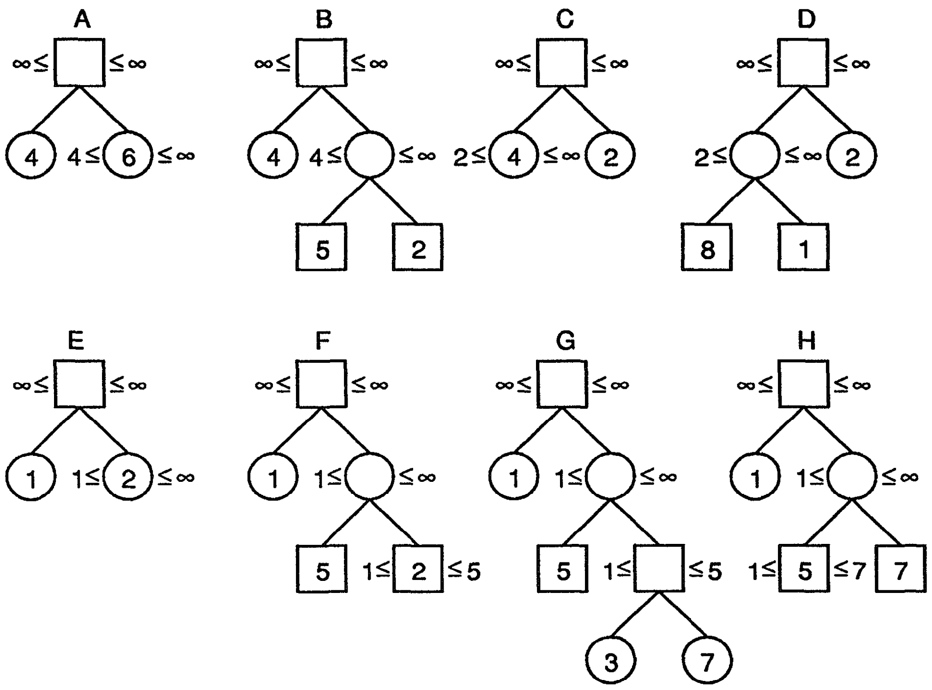

Figure 2: Recursive best-first minimax search example

root, freeing memory, but stores with the right child its new minimax value of 2, as shown in figure 2C. This method of backing up values and freeing memory is similar to that of (Chakrabarti et al. 1989).

The left child of the root will remain on the principal variation as long as its value is greater than or equal to 2, the largest value among its siblings. It is expanded, as shown 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, as shown 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 sibling, and is expanded next. The reader is encouraged to complete the example. Note that the values of interior nodes on the principal variation are not computed until necessary.

RBFMS consists of two recursive and entirely 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 backed-up minimax value of the node. At any point, the recursion stack contains the current principal variation, plus the siblings of all nodes on this path. Its space complexity is thus $O(bd)$, where b is the branching factor of the tree, and d is the maximum depth.

The children of a node are generated and evaluated one at a time. If the value of any child of a MAX node exceeds Beta, or the value of any child of a MIN node is less than Alpha, that child's value is immediately returned, without generating the remaining children.

```
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
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
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]
```

Syntactically, recursive best-first minimax appears very similar to alpha-beta, but behaves quite differently. Alpha-beta makes its move decisions based on the values of nodes all at the same depth, while best-first minimax relies on node values at different levels.[1]

Saving the Tree

RBFMS reduces the space complexity of best-first minimax by generating some nodes more than once. This overhead is significant for deep searches. On the other hand, the time per node generation for RBFMS is less than for standard best-first minimax. In the standard implementation, when a new node is generated, the state of its parent is copied, along with any changes to it. The recursive algorithm does not copy the state, but rather makes only incremental changes to a single copy, and undoes them when backtracking.

Our actual implementation uses the recursive control structure of RBFMS. When backing up the tree, however, the subtree is retained in memory. Thus, when a path is abandoned and then reexplored, the entire subtree is not regenerated, While this requires exponential space, it is not a major problem, for several reasons.

The first is that once a move is made, and the opponent moves, we only save the remaining relevant subtree, and prune the subtrees below moves that weren't chosen by either player, releasing the corresponding memory. While current machines will exhaust their memories in minutes, in a two-player game, moves are made every few minutes, freeing much of the memory.

The second reason that memory is not a serious constraint is that only the backed-up minimax value of a node, and pointers to its children must be saved. The actual game state, and alpha and beta bounds, are incrementally generated from the parent. Thus, a node only requires a few words of memory.

If memory is exhausted while computing a move, however, there are two options. One is to complete the current move search using the linear-space algorithm, thus requiring no more memory than for the recursion stack. The other is to prune the least promising parts of the current search tree. 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 time on the expected line of play, it can save much of the tree computed for one move, and apply it to subsequent moves, particularly if the opponent moves as expected. Saving the tree between moves improves the performance considerably. In contrast, the standard depth-first implementation of alpha-beta doesn't save the tree from one move to the next, but only a subset of the nodes in a transposition table. Even if alpha-beta is modified to save the tree, 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 and opponent's moves. In the best case, when alpha-beta searches the minimal tree and the opponent moves as expected, only $1/b$ of the tree that is generated in computing one move is still relevant after the player's and opponent's moves, where b is the branching factor.

Othello Results

The test of a selective search algorithm is how well it plays. We played best-first minimax against alpha-beta in the game of Othello, giving both algorithms the same amount of computation, and the same evaluation function from the program Bill (Lee & Mahajan 1990), one of the world's best Othello players.

The efficiency of alpha-beta is greatly affected by the order in which nodes are searched. The simplest ordering scheme, called fixed ordering (Slagle & Dixon 1969), fully expands each node, statically evaluates each child, sorts the children by their values, and then searches the children of MAX nodes in decreasing order, and the children of MIN nodes in increasing order. We use fixed ordering on newly generated nodes until one level above the search horizon. At that point, since there is no advantage to further ordering, 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.

Each tournament consisted of 244 pairs of games. Different games were generated by making all possible first four moves, and starting the game with the fifth move. Each game was played twice, with each algorithm 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 at the end. About 3% of the games were tied, and are ignored in the results presented below.

When alpha-beta can search to the end of the game, both algorithms use alpha-beta to complete the game, since alpha-beta is optimal when the static values are exact. In Othello, the disc differential is the exact value at the end of the game. Since best-first minimax searches deeper than alpha-beta in the same amount of time, however, it reaches the endgame before alpha-beta does. Since disc differentials are not comparable to the values returned by Bill's heuristic function, best-first minimax evaluates endgame positions at $-\infty$ if MAX has lost, ∞ if MAX has won, and $-\infty + 1$ for ties. If the principal leaf is a winning terminal node for best-first, it stops searching and makes a move. If alpha-beta makes the expected response, the principal leaf doesn't change, and best-first minimax will make

[1]While Recursive Best-First Search (RBFS) is more efficient than Simple Recursive Best-First Search (SRBFS) for single-agent problems (Korf 1993), the minimax generalizations of these two algorithms behave identically.

AB depth	1	2	3	4	5	6	7
BF depth	1	4	8	12	15	19	23
BF wins	50%	67%	78%	69%	57%	58%	51%

Table 1: Pure best-first vs. alpha-beta on Othello

its next move without further search. Conversely, if the principal leaf is a loss or tie, best-first minimax will continue to search until it finds a win, or runs out of time. While this endgame play is not ideal, it is the most natural extension of best-first minimax.

For each alpha-beta search horizon, we experimentally determined what depth limit caused best-first minimax to take most nearly the same amount of time. This was 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, and running time is roughly proportional to the number of node evaluations.

Table 1 shows the results of these experiments. The top line shows the 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 ties. Each data point is an average of 244 pairs of games, or 488 total games.

Both algorithms are identical at depth one. At greater depths, best-first searches deeper than alpha-beta, and wins most of the time. Its winning percentage increases to 78%, but then begins to drop off as the gap between the alpha-beta and best-first horizons becomes very large. At greater depths, we believe that best-first will lose to alpha-beta.

Best-First Extensions

One explanation for this performance degradation is that while best-first minimax evaluates every child of the root, it may not generate some grandchildren, depending on the static values of the children. In particular, if the evaluation function grossly underestimates the value of a node, it may never be expanded. For example, this might occur in a piece trade that begins with a sacrifice. At some point, it makes more sense to consider all grandchildren of the root, rather than nodes 23 moves down the principal variation.

To correct this, we implemented a hybrid algorithm, called best-first extension, that combines the uniform 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 minimax to a 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 evaluation, before exploring the most promising moves much deeper. This is similar to the idea of principal variation lookahead extensions (Anantharaman 1990).

AB depth	1	2	3	4	5	6	7	8
BF depth	1	4	7	10	14	18	21	24
BF wins	50%	67%	83%	81%	67%	72%	67%	68%

Table 2: Best-first extension vs. alpha-beta on Othello

Best-first extension has two parameters: the depth of the initial alpha-beta search, and the depth of the subsequent best-first search. In our 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 was whatever depth took most nearly the same total amount of time, including the initial 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 2 shows the results for Othello, in the same format as table 1. At alpha-beta depths greater than two, best-first extension performs significantly better than both alpha-beta and pure best-first minimax. At increasing depths the results appear to stabilize, with best-first extension defeating alpha-beta about two out of three games.

Random Game Tree Results

As a separate test of our results, we also experimented with a class of random game trees (Fuller, Gaschnig, & Gillogly 1973). In a uniform random game tree with branching factor b and depth d, each edge is independently assigned a random cost. The static heuristic evaluation of a node is the sum of the edge costs from the root to the node. Since real games do not have uniform branching factors, we let the number of children of any node be a random variable uniformly distributed from one to a maximum branching factor B. In order not to favor MAX or MIN, the edge-cost distribution is symmetric around zero. Our edge-cost distribution was uniform from -2^{15} to 2^{15}.

Different random games were generated from different random seeds. Each game was played twice, with each algorithm moving first. A random game ends when a terminal position is reached, 100 moves in our experiments, and returns the static value of the final position as the outcome. 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.

In random games with maximum branching factors ranging from 2 to 20, we obtained results similar to those for Othello (Korf & Chickering 1993). In particular, pure best-first outplayed alpha beta at shallow depths, but tended to lose at greater depths, while best-first extension outplayed alpha-beta at all depths.

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. One 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 Othello, best-first minimax outplays alpha-beta, giving both algorithms the same amount of computation and evaluation function, up to a given search depth, but starts to lose beyond that 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, defeating alpha-beta roughly two out of three games. While memory 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. Finally, we performed the same experiments on a class of random games, with similar results.

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 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.

Acknowledgements

Thanks to Kai-Fu Lee for the sources to Bill, to Judea Pearl, Joe Pemberton, and Weixiong Zhang for many helpful discussions, and to Hermann Kaindl for comments on an earlier draft. This work was supported by NSF Grant No. IRI-9119825, and a grant from Rockwell International.

References

Anantharaman, T.S., A statistical study of selective min-max search in computer chess, Ph.D. Thesis, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa. 1990.

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

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

Chakrabarti, P.P., S. Ghose, A. Acharya, and S.C. de Sarkar, Heuristic search in restricted memory, *Artificial Intelligence*, Vol. 41, No. 2, 1989, pp. 197-221.

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

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, 1991, pp. 1225-1235.

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

Korf, R.E., Best-first minimax search: Initial results, Technical Report, CSD-920021, Computer Science Dept., University of California, Los Angeles, 1992.

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

Korf, R.E., and D.M. Chickering, Best-first minimax search: First results, *Proceedings of the AAAI Fall Symposium on Games: Planning and Learning*, Raleigh, NC, Oct. 1993, pp. 39-47.

Kozdrowicki, E.W., and D.W. Cooper, COKO III: The Cooper-Koz chess program, *C.A.C.M.*, Vol. 16, No. 7, 1973, pp. 411-427.

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

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

Nilsson, N.J., Searching problem-solving and game-playing trees for minimal cost solutions, in *Information Processing 68, Proceedings of the IFIP Congress 1968*, A.J.H. Morrell (Ed.), North-Holland, Amsterdam, 1969, pp. 1556-1562.

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

Rivest, R.L., Game tree searching by min/max approximation, *Artificial Intelligence*, Vol. 34, No. 1, 1987, pp. 77-96.

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

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

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.

Stockman, G., A minimax algorithm better than Alpha-Beta? *Artificial Intelligence*, Vol. 12, No. 2, 1979, pp. 179-196.