Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

BEST-FIRST MINIMAX SEARCH: INITIAL RESULTS

R. Korf

January 1992
CSD-920021

Best-First Minimax Search: Initial Results

Richard E. Korf*
Computer Science Department
University of California, Los Angeles
Los Angeles, Ca. 90024
(213)206-5383
korf@cs.ucla.edu

January 14, 1992

Abstract

We present a new selective minimax search algorithm for two-player games. The basic idea is to always explore further the move that appears best so far. We describe an implementation of this algorithm that reduces its space complexity from exponential to linear, at the cost of a small constant in time complexity. On a synthetic game, our algorithm outperforms classical alpha-beta minimax search in decision quality for the same number of node generations.

Function: Problem Solving

*This research was supported by an NSF Presidential Young Investigator Award, No. IRI-8552925, and a grant from Rockwell International. Thanks to Valerie Aylett for drawing the graphs and figures.

1 Introduction and Overview

The best chess machines, such as Deep-Thought[1], generate tens of millions of board positions for each move, while their human opponents generate at most tens of positions per move. Since human players often examine deeper lines of play than the machine, they must be much more selective in their choice of positions to search. The value of selective search in two-player games was first recognized by Shannon[2].

In spite of this, however, there has been relatively little progress in discovering useful selective search algorithms. Most of the work in two-player games has focussed on techniques that make the same decisions as a full-width, fixed-depth minimax search, but do it more efficiently. The most important of these techniques is alpha-beta pruning[3], which can almost double the achievable search horizon in practice.

In contrast, the only selective searches we are aware of are the B* algorithm[4], conspiracy search[5], min/max approximation[6], meta-greedy search[7], and singular extensions[8]. All of these algorithms, except for singular extensions, suffer from exponential memory requirements, making them impractical for large searches. 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 just a static evaluation.

Singular extensions is the only selective search algorithm to be successfully incorporated into a high-performance program, the DeepThought machine. The basic idea of singular extensions is that if the best position at the search horizon is significantly better than its alternatives, then explore that position one ply deeper, and recursively apply the same rule at the next level as well. This work can be viewed as carrying this idea even further.

In the following section we describe a new selective search algorithm, called best-first minimax search. The algorithm requires no information other than a static evaluation function. In section 3 we describe an implementation of the algorithm that reduces its space complexity from exponential to linear in the search depth. In section 4 we present experimental results comparing its decision quality to alpha-beta minimax search on a synthetic game. Finally we present our preliminary conclusions in section 5.

2 Best-First Minimax Search

The basic idea of best-first minimax search is to always explore further the move that appears best so far. Given any partially expanded game tree, with static evaluations of the frontier nodes, we can compute the values of all interior nodes, including the root, using the minimax rule. Namely, the value of an interior MAX node is the maximum of the values of its children, and similarly the value of an interior MIN node is the minimum of the values of its children. The value of the root is the same as the value of at least one frontier node, and is based on that node. We call such a frontier node a *minimax node*, of which there may be more than one. The best-first minimax algorithm always expands next a minimax node on the frontier of the current search tree.

Consider the example in figure 1, where square nodes represent MAX's moves and circular nodes represent MIN's moves. 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 11, the maximum of its children's values. This means that the right child is the minimax node, and is expanded next, resulting in the situation in figure 1B. The new frontier nodes are statically evaluated at 9 and 5, and hence the value of their MIN parent changes from its static value of 11 to 5, the minimum of its children's values. This changes the value of the root to 7, the value of its left child. Thus the left move now appears more promising, the left child of the root is the new minimax node, and this node is expanded next, resulting in the situation in figure 1C. The value of the left child of the root changes from its static value of 7 to the minimum of its children's values, 3, and the value of the root changes to the maximum of its children's values, 5. 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, followed by the situation in Figure 1E. In general, best-first minimax will generate an unbalanced tree, exploring certain lines more deeply than others. Whenever some terminating condition is satisfied, the move leading to the current minimax node is made.

The most straightforward implementation of this algorithm is to maintain the current search tree in memory. When a minimax node is expanded, its children are evaluated, 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

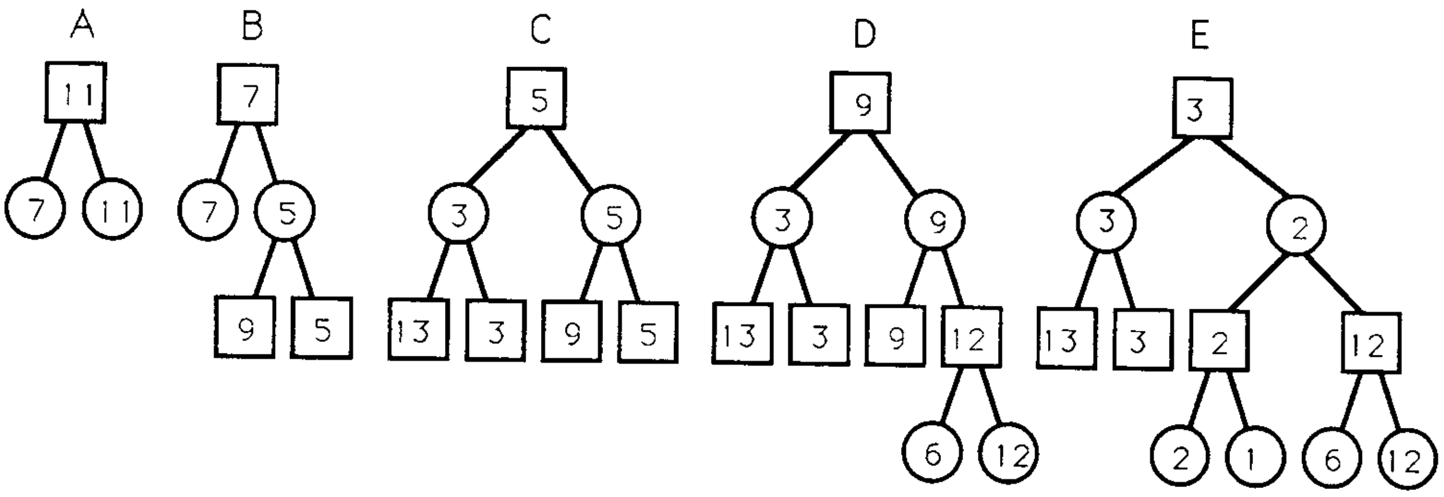

Figure 1: Best-First Minimax Search

a MAX node, or a minimum-valued child of a MIN node, until it reaches a new minimax node to expand next. The most significant drawback of this naive implementation, however, is that it requires memory that is exponential in the tree depth to store all the nodes, making it severely space-bound in practice.

3 Recursive Best-First Minimax Search

Recursive Best-First Minimax Search (RBFMS) is an implementation of best-first minimax search that runs in space linear in the maximum search depth, but expands new nodes in the same order as the naive implementation. The algorithm is a generalization of Recursive-Best First Search (RBFS)[10, 9], a linear-space best-first algorithm designed for single-agent problems, where the value of a node is always the minimum of the value of its children. Figure 2 shows the step-by-step behavior of RBFMS on the example of figure 1.

Associated with each node on the path from the root to the minimax node, called the *principal variation*, is a lower bound called alpha and an upper bound called beta. The node in question 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 principle variation as long as its minimax value is greater than or equal to the maximum value of its brothers (7), as indicated by the bounds on the right child. Thus, the right child is expanded next, resulting in the situation in figure 2B.

At this point, the value of the right child becomes the minimum of the values of its children (9 and 5), and since 5 is less than the lower alpha bound of 7, the right child of the root is no longer on the principal variation, and the left child of the root is the new minimax node. In order to save space, the algorithm returns to the root from the right child, automatically freeing the space for the children of the right child, and storing with the right child its minimax value of 5, 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 5, 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 (13 and 3), and since 3 is less than the lower alpha bound of 5, the left child is no longer on the principal variation, and the right child becomes the new minimax node. Again to save memory, the algorithm returns to the root node, and stores the new minimax value of 3 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 3, the value of its best brother, and would be expanded next. The reader is encouraged to follow the rest of the example in the figure. The reason for the empty nodes in the figure is that for efficiency, the values of interior nodes on the principal variation are not updated until necessary.

The simplest version of RBFMS, corresponding to simple recursive best-first search (SRBFS)[10, 9], consists of two entirely symmetric functions, one for MAX and one for MIN. Each purely recursive function takes three arguments, a node, a lower bound called alpha, and an upper bound called beta. Together they perform a best-first minimax search of the subtree below the node, as long as its minimax value remains within the alpha and beta bounds. Once it exceeds the bounds, the function returns the new minimax value of the node. At any given point, the recursion stack contains the current principal variation, plus the immediate brothers of all nodes on this path. As a result, its space complexity is $O(bd)$, where b is the branching factor of the tree, and d is the maximum search depth. The cost of this reduction in memory from exponential to linear is that some nodes are expanded more

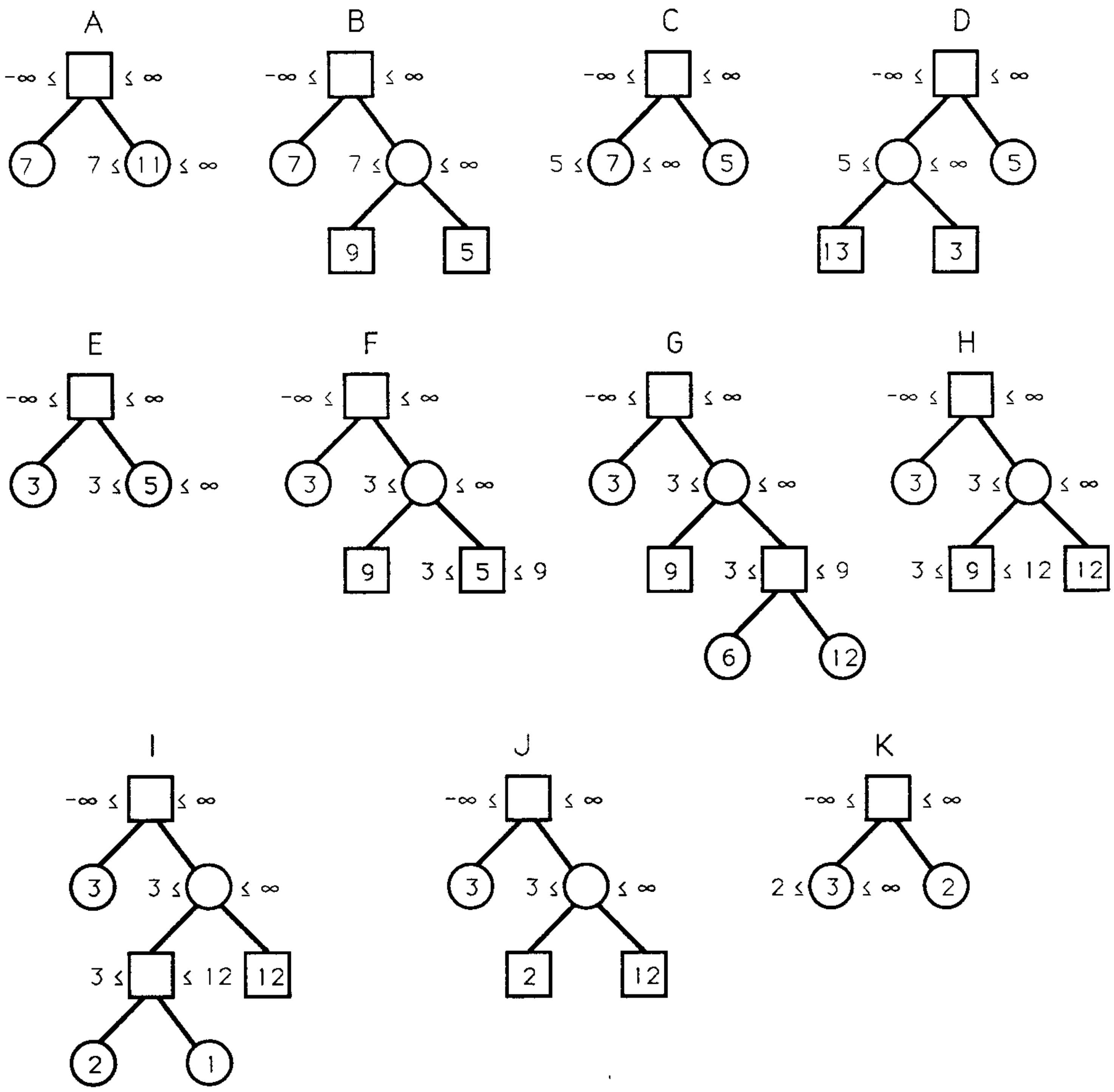

Figure 2: Recursive Best-First Minimax Search

than once. As we will see in the following section, this time overhead was a small constant factor in our experiments.

This simple version of the algorithm is less efficient than it can be, however. In particular, if the minimax value stored with a node is different than its static evaluation, then it must have been expanded before, and the stored value is the maximum of its children's last values if it is a MAX node, and the minimum of its children's last values if it is a MIN node. Thus, if the static values of any of the children of such a MAX node are greater than their parent's value, then they inherit their parent's value, since they must have been expanded before and their parent's value is a more accurate estimate of their minimax value than their static value. Similarly, if the static values of any of the children of such a MIN node are less than their parent's value, then they inherit their parent's value, since they must have been expanded before and their parent's value is a more accurate estimate of their minimax value than their static value. This refinement of the basic algorithm requires an additional parameter that is the stored minimax value of the parent node, but improves the efficiency of the algorithm. Below we give a pseudo-code description of the full recursive best-first minimax algorithm.

```
BFMAX (Node, Value, Alpha, Beta)
expand (Node)
FOR each Child[i] of Node,
   IF Value <> Static(Node), M[i] := min(Value, Static(Child[i]))
   ELSE M[i] := Static(Child[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], M[1], max(alpha, M[2]), beta)
   insert Child[1] and M[1] in sorted order
return M[1]

BFMIN (Node, Value, Alpha, Beta)
expand (Node)
FOR each Child[i] of Node,
   IF Value <> Static(Node), M[i] := max(Value, Static(Child[i]))
   ELSE M[i] := Static(Child[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], M[1], alpha, min(beta, M[2]))
   insert Child[1] and M[1] in sorted order
return M[1]
```

4 Experimental Results

The real test of a selective search algorithm is how well it plays. Since the standard algorithm is full-width fixed-depth minimax search with alpha-beta pruning, we compared the performance of best-first minimax search to alpha-beta. The obvious approach is to implement both algorithms for a standard game, such as chess, and play them against each other, giving each the same amount of computation per move, to see which algorithm wins the most games. We did not adopt this approach for several reasons. One is that any such an experiment would involve a large number of additional parameters, any one of which could have a large impact on the results. For example, a particular evaluation function must be chosen, some sort of quiescence search must be added to both algorithms, etc. The overriding consideration, however, was that we wanted an experiment that was simple enough that its results could be easily reproduced by other investigators, and hence lay a foundation for a scientific study of selective search in two-player games. Given all the necessary components of a performance chess program, and the fact that the most important ones like the evaluation function are often secret, experiments done with such programs are generally not reproducible by other researchers.

Instead, we chose a very simple synthetic game, originally called an N-game after its inventor[11]. We are given a uniform tree with branching factor b and depth d, and each edge of the tree is assigned a cost independently chosen from an identical distribution function. The static value of a node is the sum of the edge costs from the root to the node. In order not to favor either MAX or MIN, the distribution is chosen to be symmetric around zero, and in order to avoid ties among node values, the range of the distribution is chosen as large as possible. In our experiments, we used a uniform distribution from -2^{15} to 2^{15}. Figure 3 shows a sample tree where the edge cost distribution is from -9 to 9.

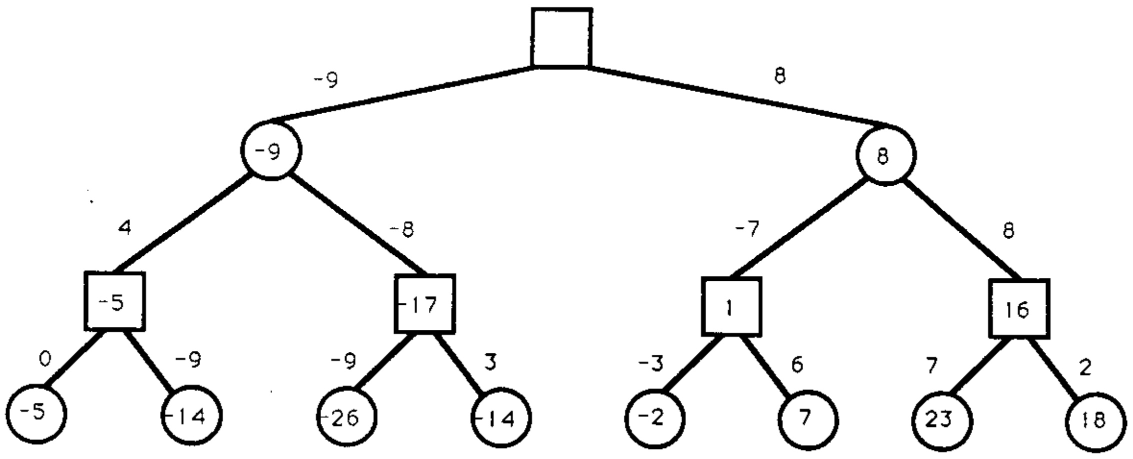

Figure 3: Sample Game Tree with B=2, D=3, and edge costs from [-9,9]

In order to compare the two different algorithms on an identical set of game trees, the entire trees must be stored in memory. A uniform tree of branching factor *b* is easily stored in an array A where the children of A[i] are located at A[bi+1], A[bi+2] ... A[bi+b]. We ran various branching factors from 2 through 10, inclusive, plus 20 and 40. For a given branching factor, the maximum depth was determined by the largest game tree we could store on a workstation with eight megabytes of memory. The maximum depths ranged from 20 on a binary tree to 4 on a tree with branching factor 40. In order to reduce the effects of the random number generator, we generated 10,000 different game trees of each size, and averaged the results.

To judge performance, we looked at *decision quality*, or the percentage of time that an algorithm made the correct *first* move in the game. A correct first move is defined as the one that would be made by a minimax search of the entire tree. To evaluate decision quality as a function of search effort, we ran alpha-beta to different search horizons, up to but not including the terminal depth *d*, and recorded the numbers of nodes generated and the percentage of correct first moves made. Finding a corresponding stopping criteria for best-first minimax search is not so obvious. We chose as a stopping condition the first time that the algorithm chose to expand a node at a particular depth, and ran the algorithm to successively greater stopping depths, from 1 through *d* inclusive.

The number of nodes generated by alpha-beta is greatly effected by the order in which nodes are searched. The simplest way to achieve good node ordering is that instead of generating and searching the children of a node one at a time (depth-first generation), we generate all the children of a node (depth-first expansion), statically evaluate each one, and then search the children of a MAX node in decreasing order of their static values, and the children of a MIN node in increasing order of their static values. Thus, our version of alpha-beta generates all children of a node if it generates any child. While at the search horizon it would be more efficient for alpha-beta to generate the children one at a time, the same optimization can be applied to best-first minimax search as well, with similar results. Since this would complicate both algorithms without significantly effecting the results, both algorithms were run using depth-first expansion.

Figure 4 shows the results for trees with branching factors 2, 3, 4, 10, and 40. Each set of lines starting from a common point corresponds to a different branching factor. The horizontal axis is the number of nodes generated on a logarithmic scale, and the vertical axis is the percentage of time that the first move selected by each algorithm was correct. The solid line shows the performance of full-width, fixed-depth search with alpha-beta pruning, and the broken lines show the performance of best-first minimax search.

The fine broken line reflects the number of nodes that would be generated by the simple, exponential memory version of the algorithm, which generates each node at most once, while the coarse broken line shows the nodes generated by the linear-space version of the algorithm, including all node regenerations. Both algorithms make the same decisions. In fact, since recursive best-first search can determine when a node is being generated for the first time, as opposed to when it is being regenerated, thus both sets of data were collected by running RBFMS. The difference in node generations between these two lines represents the node regeneration overhead of RBFMS. This overhead is about 25% for $b = 2$, drops to 18% for $b = 10$, 10% for $b = 20$, and only 5% for $b = 40$.

The reason that the best-first minimax curves don't go out as far as the alpha-beta curves is that the best-first algorithm terminates when a terminal node is chosen for expansion, and this requires fewer node generations than a complete alpha-beta search to the level just above the terminal nodes.

The data show that as the search horizon increases, increasing the number of node generations, the decision quality of both algorithms improves, as

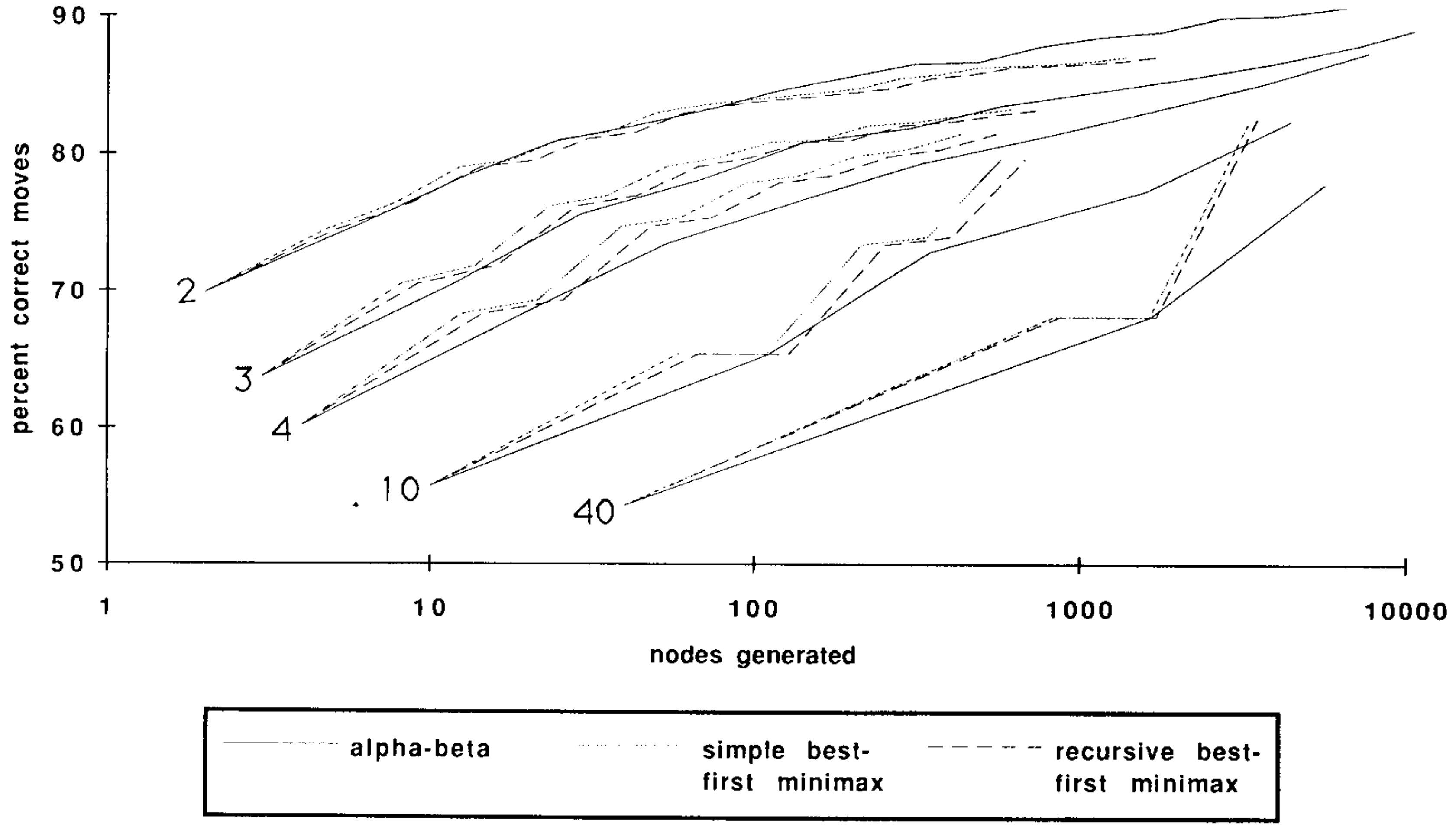

Figure 4: Decision Quality vs. Nodes Generated for Different Branching Factors

expected. Alpha-beta smoothly improves its decision quality, but the best-first minimax curves exhibit a staircase phenomenon, where increasing the search depth from an odd level to an even level doesn't increase decision quality nearly as much as going from an even level to an odd level.

With a branching factor of 2, alpha-beta outperforms best-first minimax search, at least beyond a given amount of computation. With $b = 3$, the results are mixed. For larger branching factors, however, best-first minimax consistently outperforms alpha-beta, especially if we run best-first minimax only to odd depths. Furthermore, the difference seems to increase with increasing branching factor. The lines for the branching factors not shown, 5,6,7,8,9, and 20, look similar to the lines for 4 and 10.

For this simple game, the constant factor overhead per node expansion is 27% greater for RBFMS than for alpha-beta. This is due to the fact that generating and evaluating a node is very cheap in this trivial game, and hence the difference in the bookkeeping functions between the two algorithms becomes significant. In a real game such as chess, node generation and evaluation will predominate, and the two algorithms will run at almost the same speed per node generation.

5 Conclusions

We have presented a very simple best-first minimax search algorithm. It always expands next the frontier node that appears best so far. We showed how to reduce the space complexity of the algorithm from exponential to linear in the search depth, at a cost of a small constant in nodes generated. In experiments on a simple synthetic game, with branching factors greater than 3, best-first minimax made better decisions than alpha-beta, for a given number of node generations. We believe that this represents a promising new selective search algorithm for two-player games, and the next step is to test it on a real game such as chess or Othello.

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

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

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

[7] 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.

[8] 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.

[9] Korf, R.E., Linear-space best-first search, submitted to *Artificial Intelligence*, August 1991.

[10] Korf, R.E., Linear-space best-first search: Summary of results, submitted to AAAI-92, San Jose, Ca., July, 1992.

[11] Nau, D.S., An investigation of the causes of pathology in games, *Artificial Intelligence*, Vol. 19, 1982, pp. 257-278.