ELSEVIER

Artificial Intelligence 84 (1996) 299–337

Artificial Intelligence

Best-first minimax search

Richard E. Korf*, David Maxwell Chickering

Computer Science Department, University of California, Los Angeles, CA 90024, USA

Received September 1994; revised May 1995

Abstract

We describe a very simple selective search algorithm for two-player games, called best-first minimax. It always expands next the node at the end of the expected line of play, which determines the minimax value of the root. It uses the same information as alpha-beta minimax, and takes roughly the same time per node generation. We present an implementation of the algorithm that reduces its space complexity from exponential to linear in the search depth, but at significant time cost. Our actual implementation saves the subtree generated for one move that is still relevant after the player and opponent move, pruning subtrees below moves not chosen by either player. We also show how to efficiently generate a class of incremental random game trees. On uniform random game trees, best-first minimax outperforms alpha-beta, when both algorithms are given the same amount of computation. On random trees with random branching factors, best-first outperforms alpha-beta for shallow depths, but eventually loses at greater depths. We obtain similar results in the game of Othello. Finally, we present a hybrid best-first extension algorithm that combines alpha-beta with best-first minimax, and performs significantly better than alpha-beta in both domains, even at greater depths. In Othello, it beats alpha-beta in two out of three games.

1. Introduction and overview

The best chess machines, such as Deep-Blue [10], are competitive with the best humans, but generate billions of positions per move. Their human opponents, however, search deeper along some lines of play, and much shallower along others. Obviously, people are more selective in their choice of positions to examine. The importance of searching to variable depth was first recognized by Claude Shannon in 1950 [32].

Most work on game-tree search, however, has focussed on algorithms that make the same decisions as full-width, fixed-depth minimax, searching every move to the same depth. These include alpha-beta pruning [12], fixed and dynamic node ordering [33],

* Corresponding author. E-mail: korf@cs.ucla.edu.

SSS* [34], Scout [27], aspiration-windows [8], etc. In contrast, we define a selective search algorithm as one that searches to variable depth, exploring some lines of play more deeply than others [7]. These include B* [3], conspiracy search [21], min/max approximation [29], meta-greedy search [31], and singular extensions [2]. All of these algorithms, except for singular extensions, require exponential memory, and most have large time overheads per node generation. 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 system, the Deep-Thought machine. 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. Unfortunately, applied alone, singular extensions do not result in significantly better play [1].

We describe an extremely simple selective search algorithm, which we call best-first minimax. While SSS* and related algorithms are sometimes referred to as best-first searches, these algorithms always make the same decisions as full-width, fixed-depth minimax, and should not be confused with our algorithm, which searches to variable depths. Best-first minimax requires only a single static evaluator, and its time overhead per node generation is roughly the same as 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. Next, we describe a class of random game trees that serve as an experimental testbed for our algorithm. On random trees, the time complexity of the exponential-space version of best-first minimax seems to grow with the cube of the search depth, while the time complexity of the linear-space version is exponential in depth.

We then explore the decision quality of best-first minimax compared to alpha-beta, in terms of the percentage of time that the two algorithms make optimal decisions in shallow game trees. Next, we consider overall quality of play, by playing the two algorithms against each other, both on random games and in the game of Othello. Our results show that on a large class of random game trees with a uniform branching factor, best-first minimax significantly outperforms alpha-beta. This is also true on random trees with random branching factors, and in the game of Othello, but only with limited search depths. At greater depths, however, alpha-beta outperforms pure best-first minimax in these domains.

Finally, we explore best-first extension, a hybrid combination of alpha-beta and best-first minimax. Best-first extension outplays alpha-beta in both domains, even at large search depths, winning two out of three Othello games. Earlier reports on this work include [13,15,16].

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 heuristic values of the leaf nodes, the value of an interior MAX node is the maximum of its children's values, while the value of an interior MIN node is the minimum of its children's values.

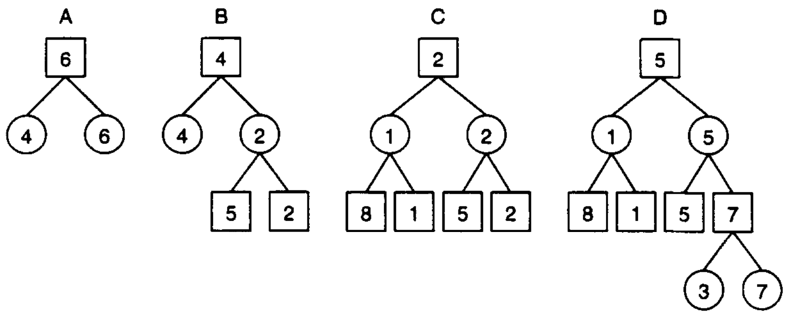

Fig. 1. Best-first minimax search example.

The *principal variation* is a path from the root to a leaf node, in which every node has the same value. This leaf node, whose value 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 effect on the minimax value of the root.

Consider the example in Fig. 1, where squares represent MAX nodes and circles represent MIN nodes. Fig. 1A shows the situation after the root has been expanded. The values of the children are their static heuristic 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 Fig. 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, resulting in the situation in Fig. 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 path is the new principal variation, and the rightmost leaf is expanded next, as shown in Fig. 1D.

By always expanding the principal leaf, best-first minimax could explore a single path to the exclusion of all others. This does not occur in practice, however, because of the following *tempo* effect. The static value of a node tends to overestimate its value from the perspective of the last player to move, since each move tends to strengthen the position of the player moving [23]. As a result, the expansion of a node tends to make it look worse from its parent's perspective, thus inhibiting further exploration of the subtree below it. For example, as shown in Fig. 1, a MIN node will only be expanded if its static value is the maximum among its brothers, since its parent is a MAX node. Expanding it changes its value to the minimum of its children's values, which tends to decrease its value, making it less likely to remain as the maximum among its siblings. Similarly, MAX nodes also tend to appear worse to their MIN parents when expanded, making it less likely that their children will be expanded next. While this oscillation in values with the last player to move is one reason that most game-search algorithms avoid comparing nodes at different depths, it tends to balance the tree searched by best-first minimax, and the effect increases with increasing branching factor.

While in principle we could make a move at any time, we choose to move when the length of the principal variation exceeds a given depth bound. This ensures that the chosen move has been explored to some depth.

The simplest implementation of best-first minimax maintains the search tree in memory. When a node is expanded, its children are evaluated, its value is updated, and this value is propagated up the tree until it reaches the root or a node whose value doesn't change. The algorithm 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. Unfortunately, this implementation requires exponential memory, a problem we address below.

Despite its simplicity, best-first minimax has apparently not been explored before. The algorithm is briefly mentioned by Nilsson as a special case of AO*, a best-first search of an AND-OR tree [26]. Harris [6] describes a version of A* applied to minimax trees, in which nodes are evaluated by $f(n) = g(n) + h(n)$, where $g(n)$ is the depth of node n in the tree, and $h(n)$ estimates the number of moves from node n to a winning position. The chess algorithm of Kozdrowicki and Cooper [18] seems related, but is difficult to decipher and behaves differently on their examples. In particular, their algorithm appears to only compare nodes at odd or even depths, hence losing the balancing effect based on tempo described above. Best-first minimax is also related to conspiracy search [21], and only expands nodes in the conspiracy set. It is also related to Rivest's min/max approximation [29]. 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 memory that is exponential in the search depth.

3. Recursive best-first minimax search

Recursive best-first minimax search (RBFMS) is an implementation of best-first minimax that runs in space that is linear, rather than exponential, in the search depth. The algorithm is a generalization of simple recursive best-first search (SRBFS) [14], a linear-space best-first search for single-agent problems. Fig. 2 shows how RBFMS behaves on the example in Fig. 1.

Associated with each node on the principal variation is a lower bound α, and an upper bound β, similar to the bounds in alpha-beta pruning. A node will remain on the principal variation as long as its backed-up minimax value stays within these bounds. The root is bounded by $-\infty$ and ∞. Fig. 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 among its siblings (4). Thus, the lower bound on this node is 4, and the right child is expanded next, as shown in Fig. 2B.

The minimax value of the right child changes to the minimum of its children's values (5 and 2). Since 2 is less than the lower bound of 4, the right branch is no longer the principal variation, and the left child of the root is the new principal leaf. The algorithm returns to the root, freeing memory, but stores with the right child its new minimax

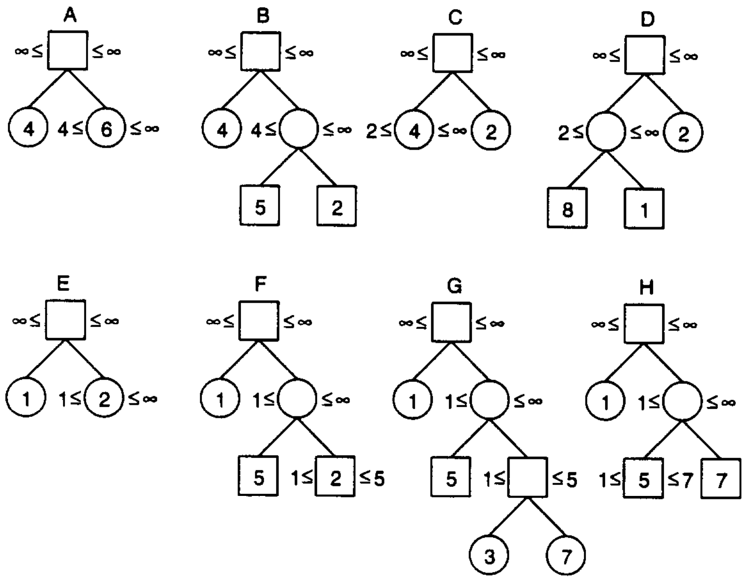

Fig. 2. Recursive best-first minimax search example.

value of 2, the minimum of its children's values, as shown in Fig. 2C. This backing up of values and freeing of memory is similar to that of [4].

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 Fig. 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 Fig. 2E. 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 largest sibling, and is expanded next, leading to the situation shown in Fig. 2F.

The rightmost branch is now the principal variation, and the rightmost grandchild is the principal leaf. It will remain on the principal variation as long as its minimax value stays between 1 and 5. If it exceeds 5, its brother becomes the new principal leaf, and if it drops below 1, its uncle, the left child of the root, will be the new principal leaf. Expanding this node, as shown in Fig. 2G, changes its value to the maximum of its children's values (3 and 7). Since 7 is greater than 5, its brother becomes the new principal leaf. The algorithm returns to the parent, storing the new minimax value (7) with the child, as shown in Fig. 2H. The node labelled 5 in Fig. 2H will remain on the principal variation as long as its value stays between 1, the lower bound on its parent, and 7, the smallest value among its brothers.

RBFMS consists of two recursive and entirely symmetric functions, one for MAX and one for MIN, shown in Fig. 3.[1] Each takes three arguments: a node, a lower bound α,

[1] Alternatively, a negamax version could be written as a single function.

```
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]
```

Fig. 3. Pseudo-code for recursive best-first minimax search.

and an upper bound β. 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 α and β bounds. Once it falls outside those bounds, the function returns the new backed-up minimax value of the node.

The lower bound on a MAX node is the same as the lower bound of its parent, and the upper bound on a MAX node is the minimum of the upper bound of its parent, and the smallest value among its immediate siblings. Similarly, the upper bound on a MIN node is the same as the upper bound of its parent, and the lower bound on a MIN node is the maximum of the lower bound of its parent and the largest value among its immediate siblings.

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

At any point, the recursion stack contains the current principal variation, plus the siblings of all nodes on this path, along with their current minimax values. Thus, the space complexity is $O(bd)$, where b is the branching factor of the tree, and d is the maximum depth. The minimax values of interior nodes on the principal variation are not computed until necessary.

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 at the same depth, while best-first minimax relies on node values at different levels. RBFMS also appears similar to Slagle and Dixon's dynamic ordering algorithm [33], but dynamic ordering makes the same decisions as full-width fixed-depth minimax, only more efficiently.

3.1. Correctness of RBFMS

We now consider the correctness of recursive best-first minimax search. In particular, we prove that RBFMS correctly implements a best-first minimax search. The first step is to define more precisely what we mean by a best-first search of a minimax tree.

Given a node, we define its static value as the value returned by applying the heuristic evaluation function to the node. Given a partially-expanded game tree, a leaf node in the tree is one with no children included in the tree, and an interior node is one with at least one child included in the tree. The minimax value of a leaf node is its static value, while the minimax value of an interior node is computed recursively as the minimum or maximum of the minimax values of its children, depending on whether the node is a MIN or MAX node, respectively. The minimax value of any subtree is defined as the minimax value of its root node.

For simplicity, assume that all static leaf values are unique. Then, there exists a unique path from the root to a frontier node, on which every node has the same minimax value. This path is the principal variation, and the leaf node at the end of it is the principal leaf. In a best-first minimax search, the next ungenerated brother of the principal leaf is generated next. If all the siblings of the principal leaf have been generated, then the first child of the principal leaf is generated next. A best-first search order is the resulting sequence of node generations. For an algorithm to implement a best-first search, each time a node is generated, all nodes that precede it in the best-first order must have already been generated at least once. This allows the same node to be generated more than once in a best-first search.

If there are ties among static node values, then there may be more than one principal variation, more than one principal leaf, and more than one best-first search order for the tree. The order in which nodes are generated by these different best-first search orders may be radically different after the first tie is broken. In that case, a best-first search is one that conforms to at least one of these best-first search orders. We assume that all subtrees are infinitely deep, meaning that every node has at least one child.

Theorem 1. *Given a lower bound α, an upper bound β, and a node n whose static heuristic value is greater than or equal to α and less than or equal to β, recursive best-first minimax search executes a best-first minimax search of the subtree below node n, as long as its minimax value lies within the bounds. When the minimax value of node n falls outside the bounds, this new minimax value is returned as the value of the search.*

Proof. Since the algorithm consists of two mutually recursive and entirely symmetric subroutines, we only provide the proof for one of the subroutines, BFMIN, and omit the completely symmetric argument for BFMAX. The proof is by induction on the depth of recursion.

Basis step: The basis step is that the theorem is true of a call to BFMIN that does not generate any recursive calls. Consider the call $BFMIN(n,\alpha,\beta)$, and assume that the static value of node n is greater than or equal to α, and less than or equal to β. Thus, a best-first minimax search of node n must generate at least one of its children.

BFMIN generates each of the children of node n one at a time, unless a child with static value less than α is generated. In that case, let x be the first node generated with static value less than α. Thus, its static value is the minimum among the children of node n generated so far, and hence the current minimax value of node n. Since the minimax value of node n is less than the lower bound of α, this value is returned as the value of the function without further node generations, as required by the theorem.

If all the children of node n are generated, and none of their static values are less than α, then they are sorted in increasing order of their static values. Since by assumption there are no recursive calls in the base case, the condition of the WHILE loop must be violated in this case. Since the static values of all the children are greater than or equal to α, $M[1]$, the smallest static value, must be greater than β. Since $M[1]$ is the smallest static value among the children of node n, it is the minimax value of n, and hence the minimax value of node n is greater than β. This value is returned as the value of the function, as required by the theorem. Thus, when there are no recursive calls, BFMIN performs a best-first search below node n until its minimax value falls outside the α or β bounds, and returns this value as the value of the function. An entirely symmetric argument applies to BFMAX.

Induction step: Assume that the theorem is true for all calls to BFMAX or BFMIN in which the depth of recursion is no greater than k. We need to show that the theorem is true for all calls in which the maximum recursion depth is $k+1$. Thus, there must be at least one recursive call from inside the WHILE loop. Again, consider BFMIN. Before entering the loop for the first time, the children of node n are sorted in increasing order of their static values. Thus, $M[1]$, the static value of $Child[1]$, is the smallest value among the children of node n, and hence the current minimax value of node n, since n is a MIN node.

Our goal is to perform a best-first minimax search of the subtree below node n, as long as its minimax value remains within the α and β bounds. After generating all the children, a best-first minimax search below node n is a best-first minimax search below $Child[1]$, as long as its minimax value, $M[1]$, is the minimum among the minimax values of the children of node n, and is within the α and β bounds. The first condition will be true as long as $M[1] \leqslant M[2]$, the minimax value of its next larger brother, $Child[2]$. This condition is satisfied by sorting the children, and assigning ∞ to $M[2]$ if there is only one child. The WHILE test guarantees that $\alpha \leqslant M[1] \leqslant \beta$. Since $M[1] \leqslant M[2]$, and $M[1] \leqslant \beta$, $M[1] \leqslant \min(\beta, M[2])$. Since $M[1]$ is initially equal to the static value of $Child[1]$, the static value of $Child[1]$ is within the α and β bounds of the recursive call. Since by assumption the maximum depth of recursion below the parent call is $k+1$, the maximum depth below any recursive call is at most k. Thus, the induction hypothesis applies to the first recursive call, and the call $BFMAX(Child[1], \alpha, \min(\beta, M[2]))$ will perform a best-first minimax search below $Child[1]$, as long as its minimax value is greater than or equal to α, and less than or equal to both β and $M[2]$. When the minimax value of $Child[1]$ falls outside these bounds, the value returned will be the new minimax value of $Child[1]$, or $M[1]$.

This can happen in one of two ways. If the new value of $M[1]$ is less than α, it will remain the smallest value among the children of node n after being inserted in the sorted order. Thus, the next test at the top of the WHILE loop will fail, and the call on the parent node will return. Alternatively, if the new value of $M[1]$ is greater than β or $M[2]$, then it must be greater than the previous value of $M[1]$, which was less than or equal to both β and $M[2]$. Therefore, as long as the WHILE loop continues to execute, the sequence of values stored for each child node forms a strictly increasing sequence for that node. Since the start of the sequence is the static value of the node, the stored values are always greater than or equal to the static values.

Once a recursive call returns, $Child[1]$ and its new minimax value of $M[1]$ will be reinserted into the sorted order of minimax values. Thus the new $Child[1]$ will be the new child of n with the lowest minimax value, and the new $Child[2]$ will be the child of n with the second lowest minimax value, if it exists. If the minimax value of the new best child, $M[1]$, is within the α and β bounds of the parent call, a new recursive call will be made on this node, $BFMAX(Child[1], \alpha, \min(\beta, M[2]))$. Since the static values of all the children are greater than or equal to α, and the static values are always less than or equal to the stored values, at the top of the WHILE loop, and the value of $M[1]$ is less than or equal to both β and $M[2]$, the static value of $Child[1]$ is within the α and β bounds of this recursive call. Furthermore, since it is a recursive call, its maximum recursion depth is less than or equal to k. Thus, the induction hypothesis applies to this recursive call, and it will correctly continue the best-first minimax search below node n, as will all subsequent recursive calls within the WHILE loop, by induction on the number of such recursive calls.

The WHILE loop will continue until the smallest minimax value among the children of node n falls outside the α or β bounds. Since n is a MIN node, the smallest minimax value among its children is also the minimax value of node n, and hence the minimax value of node n must fall outside the α or β bounds. At that point, BFMIN returns this new minimax value to its parent, as required by the theorem.

Thus, by induction on the depth of recursion, $BFMIN(n, \alpha, \beta)$ performs a best-first minimax search below node n, until its minimax value is less than α or greater than β, at which point it returns this new minimax value. A completely symmetric argument applies to BFMAX, which completes the proof of our theorem. □

Given this theorem, a call to BFMAX or BFMIN on the root of a tree, with an α bound of $-\infty$ and a β bound of ∞, performs a best-first minimax search of the tree. For simplicity, we assumed that every branch of the tree is infinitely deep, and hence the root call would never terminate. In practice, however, there are two conditions that would terminate the search. The first is if the principal leaf were a terminal node and had no children. The second is if we artificially limit the search depth in order to make a move within a particular time interval. In that case, we terminate the search when the principal leaf reaches this cutoff depth.

3.2. Can we improve the efficiency of RBFMS?

The reader familiar with [14] may notice that RBFMS is a generalization of simple recursive best-first search (SRBFS) to two-player games.[2] As such, it suffers from the same inefficiency that plagues SRBFS in the single-agent setting. In particular, when searching parts of the tree that have already been explored, it continues to search best-first, regenerating the same nodes multiple times before generating any new nodes.

The solution to this problem in the single-agent case is to traverse old territory in a depth-first order, while still exploring new ground in a best-first fashion. This more efficient algorithm, called recursive best-first search (RBFS), works as follows. In the single-agent setting, all nodes are MIN nodes. If a node's stored value is different from its static value, then it must have been expanded before. Its stored value is the minimum of the last stored values of its children. When it is expanded again, if the static values of any of its children are less than the stored value of the parent, the child is assigned the stored value of the parent, since this is a more accurate estimate of the last backed-up value of the child. This inheritance of parent's stored values reduces the asymptotic complexity of SRBFS, while maintaining the best-first search order for new node generations.

The obvious thing to do to try to improve the efficiency of RBFMS is to apply this same modification to BFMIN, and the symmetric modification to BFMAX. In order for inheritance of stored values to be valid, however, whenever one child of a node is generated, all children must be generated. This full-expansion version of the algorithm is obtained by deleting the fourth lines of the pseudo-code descriptions in Fig. 3, "`IF M[i] > Beta return M[i]`" and "`IF M[i] < Alpha return M[i]`". Surprisingly and unfortunately however, even with full expansion, inheritance of stored values has no effect, and the minimax generalization of RBFS behaves identically to the minimax generalization of SRBFS. The reason is that in the two-player case, a node never inherits its parent's stored value, but always receives its static value instead.

Consider the example tree fragment in Fig. 4, and whether node z could inherit the stored value of its parent, node x. In order for a node to inherit its parent's stored value, there must be a first time that this happens, so without loss of generality, let z be the first node to potentially inherit its parent's value. Thus, node x did not inherit the value of its parent node w, and the initial value of node x is its static value. In order for node z to inherit the stored value of node x, the stored value of node x must be different from its static value. That can only happen by expanding node x, returning to node w and storing a new value for node x, expanding at least one of its brothers y, and then returning to expand node x again. The first time that node x is fully expanded, its minimax value becomes the minimum of its children's values. Thus, the first backed-up

[2] The reader who is not familiar with [14] may skip this section without loss of continuity.

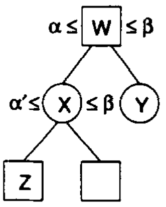

Fig. 4. Children never inherit their parent's stored values.

minimax value of node x, excluding its static value, is less than or equal to the static values of all of its children, including node z.

If this new minimax value of node x is within its upper and lower bounds, then the subtree below it will continue to be expanded until its minimax value falls outside these bounds. This can happen in one of two ways, by exceeding the upper bound β, or by dropping below the lower bound α'. If the minimax value of node x exceeds its upper bound β, then node x will still have the maximum value among its brothers, and the minimax value of its parent w will also exceed its upper bound of β, causing control to return to the parent of node w, erasing node x and all its brothers from memory.

Alternatively, the minimax value of node x could fall below its lower bound α'. α' is the maximum of α, the lower bound on its parent w, and the highest value among the brothers of node x, say node y. If $\alpha' = \alpha$, and the minimax value of node x falls below α, then the values of all the children of node w are below α, again causing control to return to the parent of node w, erasing node x and all its brothers from memory.

The only way that node x can get a new stored value that stays in memory is if α' equals the minimax value of a brother node y, and the minimax value of node x falls below α'. In that case, this new minimax value will become the stored value of node x, and control will return to its brother y. Since the minimax value of node x after it was first expanded was within its α' and β bounds, and the new stored value of x is less than its α' bound, the new stored value is less than the first backed-up minimax value. Since the first minimax value of x was less than or equal to the static values of all its children, its new static value is less than the static values of all its children. Similarly, as long as node x remains in memory, the sequence of stored values assigned to x form a strictly decreasing sequence, and all of them are less than or equal to the static values of all the children of node x.

The child of a MIN node could only inherit its parent's stored value if the child's static values were less than the parent's stored value. Since this can never happen, the children are always assigned their static values when they are generated. An entirely symmetric argument applies to the children of MAX nodes, and hence they never inherit their parent's stored values either.

Thus, the two-player generalization of RBFS behaves identically to the two-player generalization of SRBFS, and is no more efficient. The efficiency could be improved if a way could be found to explore old territory in a depth-first fashion, using only linear space, but this remains an open problem.

4. Incremental random game trees

The next issues to be addressed are the efficiency of best-first minimax search and its overall quality of play. The answers to these questions will depend on the particular game that is chosen as a domain. We will discuss experiments in two domains, a class of random game trees, and the game of Othello.

4.1. Why random games?

While standard games such as Chess, Checkers, and Othello have traditionally served as common testbeds for research in two-player games, they suffer from several drawbacks. The first is that they are relatively complex to implement. The second is that the best evaluation functions are well-kept secrets, due to the competitive nature of building high-performance game programs. This prevents the reproduction of results by other researchers. Finally, the search trees for these games are fixed. There is a unique game tree associated with each of these games, with fixed branching factors and search depths. This prevents the variation of these parameters by the experimenter. For example, due to the depth of a real game, we cannot determine whether individual moves are optimal or not, except near the endgame.

To overcome these limitations, we have experimented with a class of random game trees that are easy to implement, and allow experimental control over the branching factor, search depth, and heuristic evaluation function. Our goal is to employ a class of two-player games that are simple and compelling enough to encourage other researchers to adopt them, and hence facilitate the reproduction and sharing of research results. Briefly, an *incremental random game tree* is generated by assigning independent random values to the edges of a tree, and computing the heuristic evaluation of an interior node, as well as the exact value of a leaf node, as the sum of the edge costs from the root to that node. This produces a correlation between the heuristic value of a node and that of its descendants.

Such trees were first used by Fuller, Gaschnig, and Gillogly [5] to empirically determine the efficiency of alpha-beta pruning. Newborn [25] used the same model to analyze the complexity of alpha-beta on shallow-depth trees. Berliner used random trees to evaluate the B* search algorithm [3]. Karp and Pearl [9] and McDiarmid and Provan [22] used single-agent random trees to analytically determine the complexity of best-first search. We have also used these trees in our own studies of single-agent search algorithms [17,28,35]. Nau [24] used a different variation of this model to investigate the cause of minimax pathology. In his experiments, the tree was generated as described above, and terminal nodes with positive or negative values were labelled wins or losses, respectively. The evaluation function of an interior node was simply the number of winning terminal nodes in the subtree below it.

We describe two contributions to this model. The first is experimental evidence that the details of how the trees are generated can have a large effect on their statistical properties. Secondly, we describe an efficient technique for generating large incremental random trees in a reproducible manner, using space that is only linear in the tree depth.

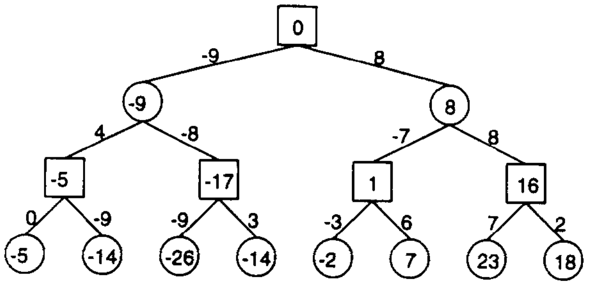

Fig. 5. Uniform binary incremental random game tree example.

4.2. What is an incremental random game tree?

An incremental random game tree consists of the tree structure itself, plus a numerical evaluation associated with each node. We first address the structure of the tree, and then consider the evaluation function.

In a uniform tree, every node has the same number of children b, and every leaf node is at the same depth d, where b and d are parameters chosen by the experimenter. Fig. 5 shows an example of a random game tree with uniform branching factor two and uniform depth three. To generate trees with non-uniform branching factor, the number of children of each node is an independent random variable, between one and some maximum branching factor B. The mean branching factor b of such a tree is the expected number of children of a node, which is $(B+1)/2$ for a uniform distribution. To generate trees of non-uniform depth, we allow nodes to have zero children, resulting in leaf nodes at varying depths.

Next we associate a number with each node of the tree, modeling a heuristic evaluation function. The most important property of the evaluation function is that the value of a given node must be correlated with the values of its descendants, but not perfectly. We achieve this correlation by independently assigning to each edge of the tree a random value from a common distribution function. The heuristic value of a node is then computed as the sum of the edge costs from the root to the given node. Thus, the closer two nodes are in the tree, the more edges their paths to the root have in common, and the more highly correlated their values will be. We choose the edge-cost distribution to be nearly symmetric around zero, to eliminate bias in favor of the maximizer or minimizer. In our experiments, the random edge costs were uniformly distributed from -2^{14} to $2^{14}-1$. The issue of determining the winner of a random game will be addressed in Section 8.2.

4.3. Generating incremental random trees

The easiest way to assign the edge costs in an incremental random tree is to call a pseudo-random number generator whenever a new edge is generated by the search algorithm. We call this the on-demand method. This works fine if a given tree is only searched once, and the same node is never revisited. For example, alpha-beta minimax performs a single depth-first search of a tree, and nodes are never revisited. Different trees are constructed by seeding the random number generator with different initial values.

Recursive best-first minimax search, however, revisits the same node more than once. When this happens, the same value must be assigned to a node each time it is generated. Furthermore, when comparing two different algorithms on the same tree, such as alpha-beta and best-first minimax, they will generally search different parts of the tree, and the same node must be assigned the same value by both algorithms. Finally, when playing the same game more than once, different plays will generate different parts of the tree, and the same value must always be assigned to the same node.

The simplest solution to this problem is to generate and store the complete tree in memory. A well-known technique for storing a tree of uniform branching factor in an array is to store the nodes in breadth-first order. For a binary tree, for example, the root goes in location 0, the left and right children go in locations 1 and 2, respectively, the left grandchildren go in locations 3 and 4, and the right grandchildren go in locations 5 and 6, etc. In general, for a uniform tree with branching factor b, the children of the node in location x go in locations $b \cdot x + 1, b \cdot x + 2, \ldots, b \cdot x + b$. For a non-uniform tree, we use the same scheme with b equal to the maximum branching factor B. We refer to this as a breadth-first enumeration of the tree. Since random number generators produce a sequence of pseudo-random values starting with an initial seed, we assign these random values to the edges of the tree in breadth-first order, associating each edge with the node below it.

Unfortunately, memory limitations severely restrict the size of trees that can be stored. This problem is exacerbated by non-uniform trees, which waste most of the space, and the fact that most algorithms examine only a tiny fraction of the entire tree.

We need a way to reproducibly generate each random edge cost from its parent node, without storing the entire tree in memory. One way to do this is to reseed the random number generator with the index of the node in the breadth-first enumeration, or some other function of the path to the node, and use the resulting random number as the value of the associated edge. This is the approach taken by Berliner [3].

4.4. *All random trees are not equally random*

So far, we have described three different methods for generating random trees: assign edge costs on demand as the trees are generated, assign edge costs breadth-first, and reseed the random number generator with the breadth-first index of the parent node before generating the edge costs of the children. An important question is which if any of these schemes produce trees with the desired random properties.

In order to address this question, we experimentally studied the number of nodes generated by two different search algorithms on all three types of random trees. We chose alpha-beta on minimax trees, and depth-first branch-and-bound on single-agent trees of MIN nodes, since both are one-pass algorithms, and hence allow generating random edge costs on demand. We found that the average numbers of nodes generated by both alpha-beta and depth-first branch-and-bound on the breadth-first random trees were statistically the same as for the on-demand random trees, in spite of their very different assignments of random values to the edges. Reseeding the random number generator at each node with its breadth-first index, however, produced trees in which both algorithms were significantly more efficient than on the other two types of trees.

One possible explanation for this is that a random number generator is designed to produce a sequence of pseudo-random values, assuming that the successive seeds are produced by the random number generator itself. Reseeding the random number generator each time with another seed is not guaranteed to produce a sequence with the desired properties. Since breadth-first and on-demand assignment produced trees that were very different than those produced by reseeding the random number generator, the actual scheme used to generate a "random" tree can significantly effect any results obtained using such trees.

4.5. *Efficient reproducible random trees*

Getting the same results from on-demand and breadth-first assignment strongly suggests that both types of trees are as random as the underlying pseudo-random number generator. Lacking any evidence to the contrary, we adopt the breadth-first assignment of random edge costs, since it is reproducible, yet generates trees with the same characteristics as the on-demand assignment. Starting with the initial seed, the sequence of random values are assigned to the edges in breadth-first order. Thus the breadth-first index of a node is also the index of its associated random edge cost in the pseudo-random number sequence. Given the random seed used to generate the parent edge cost, a child's edge cost is produced by "jumping ahead" in the sequence of random values as many steps as the difference between the indices of the two nodes in the breadth-first enumeration of the tree. The easiest way to do this is to make the requisite number of calls on the pseudo-random number generator, and discard all the intervening values, but this is very slow. We would like to jump ahead in the sequence of random values more efficiently.

To do this, we need to examine the details of a typical pseudo-random number generator, such as the standard C library function "rand" [11]. Given a seed s_i, the next seed is computed as $s_{i+1} = a \cdot s_i + c$, where $a = 1103515245$ and $c = 12345$ in this case, and all arithmetic is modulo 2^m, where m is the word size of the machine in bits. The random value returned is a subset of the bits of the seed. All linear congruential pseudo-random number generators are of this form, with different values for the constants a and c.

Expanding the recurrence further, we see that $s_{i+2} = a \cdot s_{i+1} + c = a^2 \cdot s_i + a \cdot c + c$, $s_{i+3} = a^3 \cdot s_i + a^2 \cdot c + a \cdot c + c$, and in general,

$$s_{i+n} = a^n \cdot s_i + c \cdot \sum_{j=0}^{n-1} a^j.$$

By successive squaring, a^n can be computed in $O(\log n)$ time, using the recurrences $a^{2n} = a^n \cdot a^n$ and $a^{2n+1} = a^{2n} \cdot a$. Unfortunately, the familiar closed-form solution for the sum of the powers of a cannot be used, since it involves division by $a - 1$, which requires that $a - 1$ have an inverse modulo 2^m, and it generally will not.

However, we can still compute the sum of the powers of a in $O(\log n)$ time, using the recurrences

$$\sum_{j=0}^{2n} a^j = \sum_{j=0}^{2n-1} a^j + a^{2n} \quad \text{and} \quad \sum_{j=0}^{2n-1} a^j = \sum_{j=0}^{n-1} a^j + a^n \cdot \sum_{j=0}^{n-1} a^j.$$

This allows us to move from one random seed to another that is n away in the sequence in $O(\log n)$ time, rather than $O(n)$ time.

Since a is a constant, this computation can be further speeded up by a factor of two as follows: Precompute and store the values of a^{2^i} for i from 1 to m, the word length of the machine. Then, to compute a^n, multiply together the powers a^{2^i} for those values of i where the ith bit of n is equal to one. Similarly, precompute and store the sums of the powers of a from a^0 to a^{2^i-1}, for i from 1 to m. Then, accumulate the sums of the powers of a from a^0 to a^{n-1} as follows. Initialize the accumulator to zero. Starting from the rightmost bit of n, bit 0, for each i where bit i of n is one, multiply the accumulator by a^{2^i} and add the sum of the powers of a from a^0 to a^{2^i-1}. This scheme reduces the number of multiplications and additions from order $\log n$ to the order of the number of one bits in the binary representation of n.

Note that all arithmetic above is unsigned integer arithmetic, and only the residues modulo 2^m are preserved. In a large tree, the node indices will eventually exceed 2^m and wrap around. No special action is taken when this happens, and the sequence of random values similarly wraps around when its length exceeds 2^m. Thus, the asymptotic complexity of jumping ahead in the random sequence as described above is bounded by a constant, which is proportional to the number of bits m used to represent an integer in the machine, such as 32 bits.

Once the random value for one child is computed, its sibling's values are computed by successive applications of the random number generator. This makes node expansion almost as efficient as node generation. Even using the improved scheme above, however, node generation and evaluation is still the dominant cost of alpha-beta or best-first minimax search in an incremental random game tree, as it is in most real games as well.

5. Efficiency of best-first minimax

Now that we have defined a suitable domain for experimentation, we turn back to the complexity of best-first minimax search, and in particular the relative time complexity of the exponential-space and linear-space versions.

Fig. 6 shows the performance of exponential-space best-first minimax search on uniform incremental random trees with various branching factors b. The horizontal axis is the search depth, ranging from 1 to 30. A depth-limited best-first minimax search is terminated when the principal leaf node is at the cutoff depth. The vertical axis is the number of nodes generated. This does not count the nodes regenerated in moving from one principal leaf to another, since the dominant cost is the expansion of a node for the first time. Each line represents a random tree with

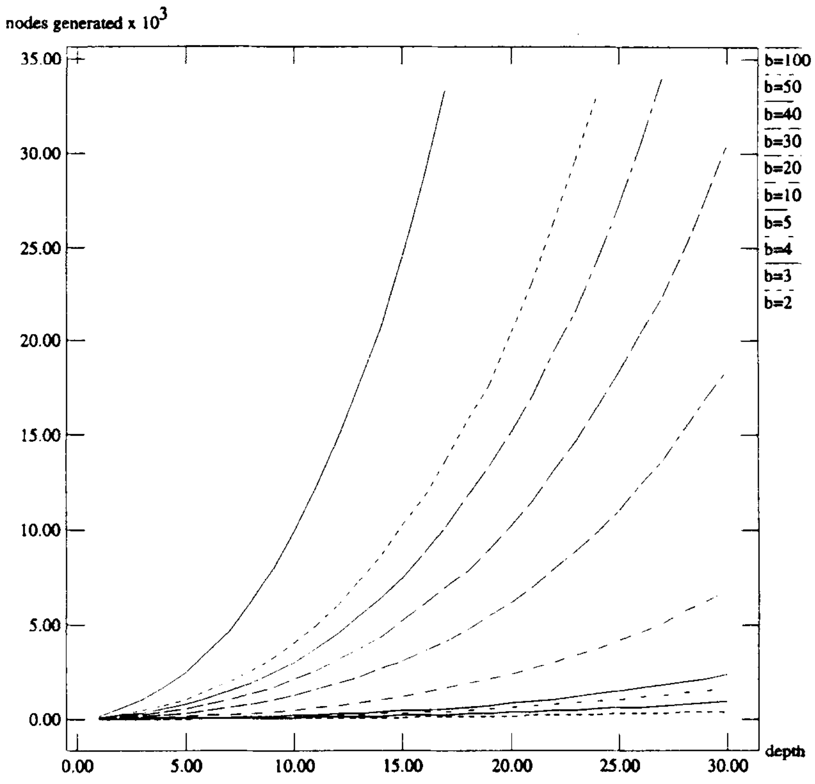

Fig. 6. Nodes generated by exponential-space best-first minimax search.

a different fixed branching factor, ranging from 2 to 100. For each branching factor and search depth, 1000 different searches were run, seeding the random number generator with the values 1 through 1000, and the number of node generations were averaged. The 95% confidence intervals were computed for the last data point on each curve, and the largest of these was plus or minus 3.36% of the sample mean.

Fig. 7 shows the same data as Fig. 6, except that the horizontal axis is the cube of the search depth. Fig. 7 suggests that the number of nodes generated grows no faster than a constant factor times the cube of the depth. Furthermore, the absolute number of node generations seems quite low. For example, in a tree with a branching factor of 100, a best-first minimax search to depth 17 generates less than 34,000 nodes on average.

Fig. 8 shows the performance of recursive best-first minimax search, the linear-space version of the algorithm, on uniform trees with various branching factors *b*. The horizontal axis is the search depth, ranging from 1 to 20, while the vertical axis is the total number of nodes generated, including all node regenerations, on a logarithmic scale. As in Fig. 6, each line represents a random tree with a different fixed branching factor, ranging from 2 to 100. For each branching factor and search depth, 10,000 different searches were run, starting with random seeds from 1 to 10,000, and the number of

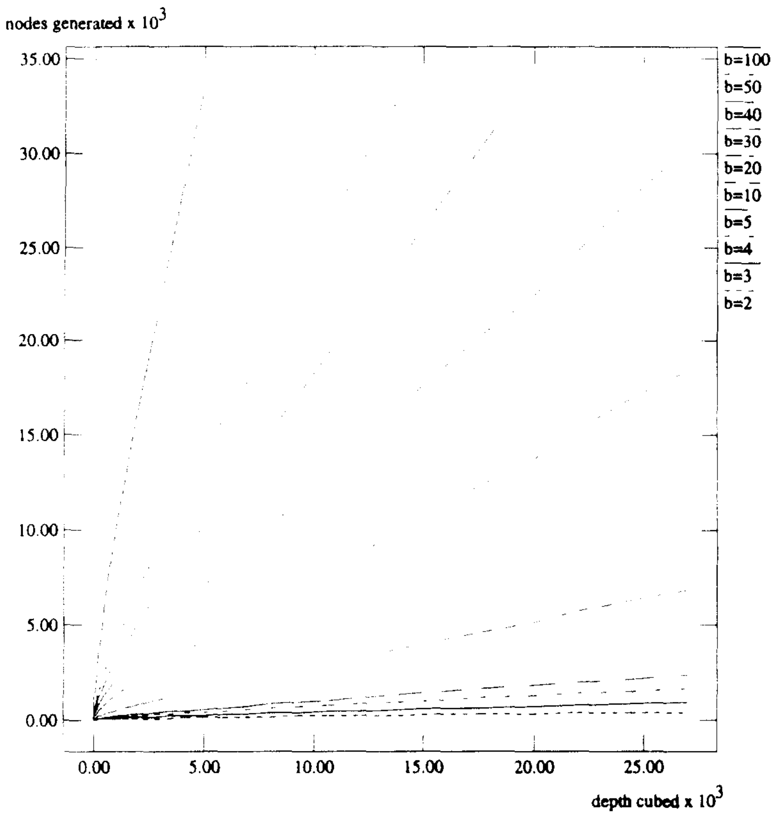

Fig. 7. Nodes generated by best-first minimax versus cube of depth.

node generations were averaged. The 95% confidence intervals were computed for the last data point on each line, and the largest of these was plus or minus 6.42% of the sample mean.

This graph suggests that the complexity of recursive best-first minimax search grows exponentially with depth, and thus is far less efficient than the exponential-space version of the algorithm. The reason is that RBFMS always searches the tree in best-first order, even when covering old territory. The exponential-space version, on the other hand, goes directly down the old principal variation when reexpanding a previously explored subtree.

5.1. *Full versus partial expansion*

The data in Fig. 8 correspond to the pseudo-code description of RBFMS in Fig. 3, where the children of a node are generated one at a time. The full-expansion version of the algorithm, in which all the children of a node are generated if any are generated, is obtained by deleting the fourth lines of BFMAX and BFMIN. Full expansion is more expensive than partial expansion for shallow searches, since it generates nodes that are

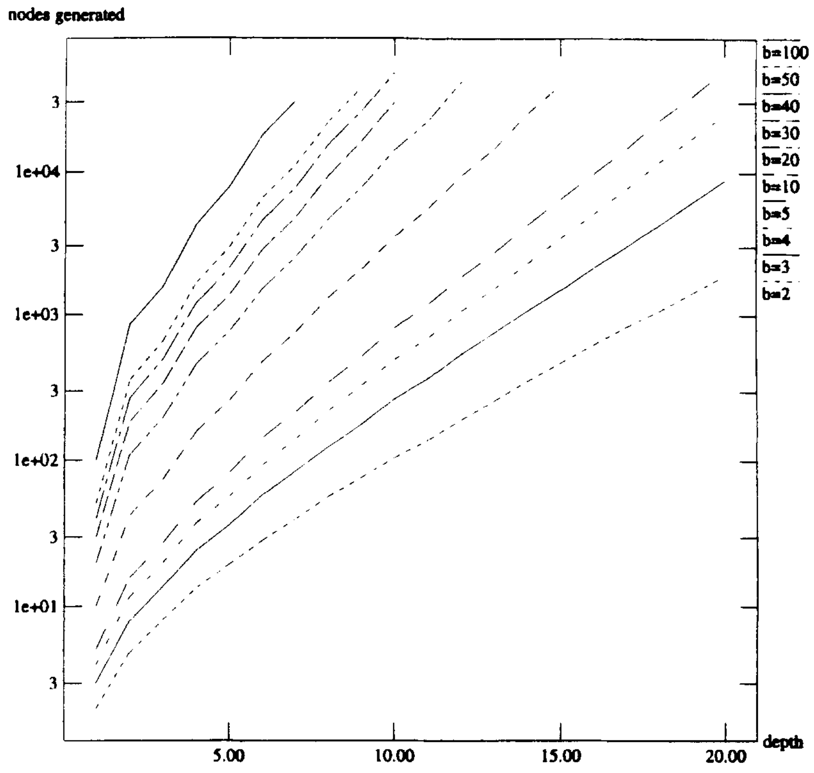

Fig. 8. Nodes generated by recursive best-first minimax search.

not generated by partial expansion. For deep searches, however, full expansion is more efficient, since fully expanding a node allows a more accurate minimax value to be stored with the node, which reduces the number of times the node must be revisited. For the exponential-space algorithm, however, partial expansion is always more efficient, since there is little node regeneration overhead.

5.2. Saving the tree

The memory issue is best understood from a historical perspective on this research project. We first rediscovered best-first minimax search in 1987, but didn't pursue it on account of its exponential-space complexity, feeling that this was a fatal flaw. In the meantime, we developed an algorithm to perform single-agent best-first search in linear space [14]. The fact that this algorithm easily generalized to minimax search rekindled our interest in best-first minimax.

The resulting algorithm, RBFMS, has two advantages over standard best-first search. The most obvious is that it runs in space that is linear rather than exponential in the search depth. The second advantage is that the time per node generation for RBFMS is less than for standard best-first search. In the standard implementation, when a new node is generated, the state of its parent is copied, and then changed to reflect the new state. The recursive algorithm does not copy the state, but rather makes only incremental changes to a single copy, and undoes them when backtracking. In chess, for example, it is much more efficient to make a move by making the necessary modifications to a single copy of the board, than to copy the entire board, and then make the changes to the new copy.

The drawback of RBFS, however, is the overhead of regenerating previously generated nodes. When the principal variation changes, the algorithm must backtrack up to the lowest common ancestor between the old and new principal variations, and then search best-first to find the new principal leaf. Due to the failure of inheritance in the two-player case, as explained in Section 3.2, this overhead is large in deep searches, and much greater than in the single-agent case. The experiments reported above indicate that on random trees, the time complexity of standard best-first minimax is cubic in depth, while the time complexity of RBFMS is exponential in depth.

In order to avoid most of the node regeneration overhead, but to retain the lower constant time per node generation, our actual implementation uses the recursive control structure of RBFMS, but when backing up the tree, the subtree is retained in memory. Thus, when a path is abandoned and then reexplored, the entire subtree is not regenerated, but only the nodes on the path to the new principal leaf. This eliminates most node regenerations and all node reevaluations. While this still requires exponential space, it is not a major problem, for several reasons.

The first 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 moves that weren't chosen by either player. This releases the corresponding memory, and the search for the next move then starts from the remaining subtree in memory. While memory may be exhausted in a matter of minutes, in a two-player game moves are made every few minutes, freeing most of the memory.

The second reason is that all that needs to be stored at a node is its backed-up minimax value, and pointers to its children. 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.

Finally, if memory is exhausted while computing a move, there are two options. One is to complete the current move search using 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 search tree in memory. Since all siblings of nodes on 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.

6. Efficiency of alpha-beta pruning

Since we will compare best-first minimax to alpha-beta minimax, we must implement alpha-beta as efficiently as possible. For a tree with branching factor b, the effective branching factor of alpha-beta can range from b in the worst case, to $\sqrt{b}$ in the best case [27]. The actual performance depends on the order in which the tree is generated. In the best case, the children of MIN nodes are generated in increasing order of their minimax values, and the children of MAX nodes are generated in decreasing order.

Unfortunately, we don't know the minimax values of the children of a node until we search them to the depth horizon. However, we can approximate their minimax values by their static evaluations. Given the static values of the children of a MIN node, we search them in increasing order, and search the children of a MAX node in decreasing order of their static values. This is called fixed or static node ordering [33]. Ordering the children of a node requires generating and evaluating them all. This is done to make the search below the children proceed more efficiently. There is no point in ordering the children at the search frontier, because this eliminates any pruning at that level with no possible benefit, since there is no further search below the frontier. Thus, at some depth in the tree, node ordering should be discontinued, and children generated one at a time to maximize pruning. We determined experimentally that fully expanding and ordering nodes at all levels of the tree except for the final level produces close to the best performance on random trees. While in some cases terminating node ordering sooner reduces the number of nodes generated, this reduction is typically small. Thus, when we refer to static node ordering, we mean ordering the children based on their static heuristic values, at all levels except the search horizon.

Iterative deepening can also be used to try to improve node ordering. In performing an alpha-beta search to depth d, a series of alpha-beta iterations from depth 1 to depth $d-1$ are performed first, with the backed-up minimax values from each iteration used to order the nodes in the next iteration. For those nodes that were not generated in any previous iteration, static ordering as described above is used to order the children above the search horizon. In our implementation, the entire tree is saved from one iteration to the next, in order to maximize the ordering information retained between iterations. Furthermore, we only count the first time a node is generated in our performance measure, since generating and evaluating a node is likely to dominate the cost of simply chasing pointers in an existing tree in memory. Note that in many published versions of the alpha-beta algorithm, if the minimax value of a subtree lies outside of its alpha or beta bounds, the violated bound is returned as the value of the child. In order to get the full benefit of node ordering from iterative deepening, however, the best value or bound achieved for the subtree should be returned, even if it falls outside the alpha or beta bounds.

Fig. 9 shows the performance of several different versions of alpha-beta minimax on random incremental binary trees ranging in depth from 1 to 26 moves. The horizontal axis is the depth of the tree, and the vertical axis is the average number of nodes generated, on a logarithmic scale. Each data point is the average of 100 random trees with initial seeds ranging from 1 to 100. The top line is alpha-beta without any node ordering, or equivalently, random node ordering. The next line down is alpha-beta with iterative deepening, plus static node ordering for new nodes. The next line is alpha-beta with just static node ordering, and the bottom line represents the theoretical optimum, or perfect node ordering. The largest of the 95% confidence intervals for any of these data points is 6.16% of the sample mean. As expected, both static node ordering and iterative deepening perform better than random node ordering, but worse than optimal node ordering.

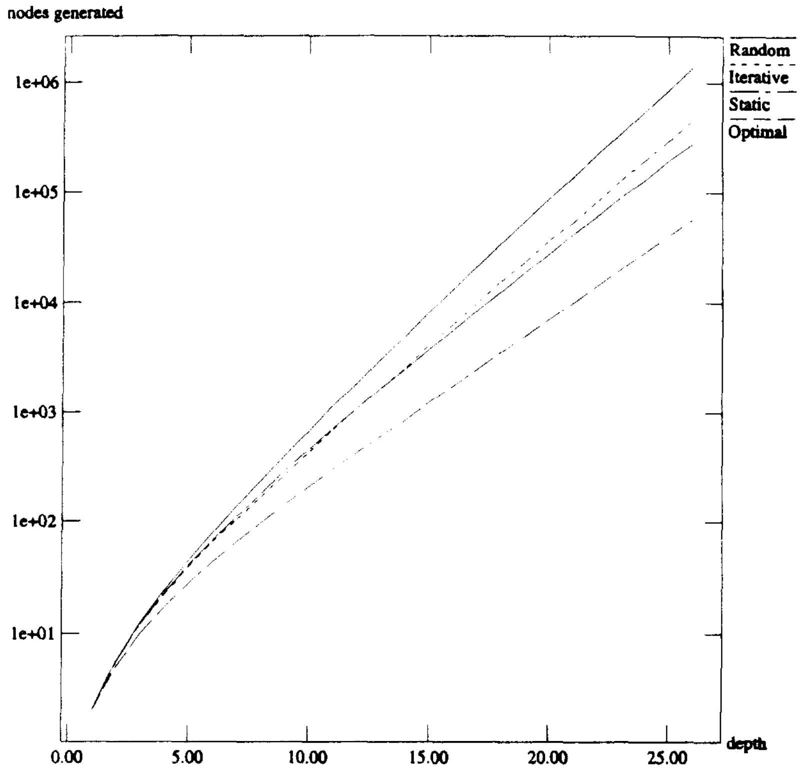

Fig. 9. Nodes generated by alpha-beta on random incremental binary tree.

What is surprising is the relative performance of iterative deepening compared to static node ordering. The algorithms are identical at depths one and two. For depths 3 through 12, iterative deepening performs slightly better than static node ordering, although the maximum improvement is less than 8%. At depths 13 and above, however, static node ordering outperforms iterative deepening, and the divergence of the two lines on the logarithmic scale suggests that the asymptotic effective branching factor is lower for static ordering than for iterative deepening. This seems particularly surprising when you recall that nodes are only counted the first time they are generated by iterative deepening. Certainly the node ordering from iterative deepening is at least as good as the static node ordering. The reason for the difference in performance is that in general, an alpha-beta search to depth $d-1$ will generate some nodes that are not generated in an alpha-beta search of the same tree to depth d. While this is not true for best-possible and worst-possible ordering, it is true in between. These additional node generations eventually overcome the small additional improvement in node ordering at greater depths. The same phenomenon is observed with larger branching factors, but the improvement of iterative deepening over static node ordering at shallow depths is even smaller. Furthermore, the depth at which static ordering outperforms iterative deepening decreases with increasing branching factor. At branching factor five, for example, iterative deepening only outperforms static ordering at depth three, and by only 1%.

In the experiments reported below, two different implementations of alpha-beta are used. In Section 7, we compare the quality of single decisions made by alpha-beta and best-first minimax. Since static node ordering is asymptotically more efficient, and iterative deepening is only slighly better at shallow depths, static ordering is used in the decision quality experiments. In Section 8, however, we actually play alpha-beta against best-first minimax, with both algorithms saving their relevant subtrees from one move to the next. In this case, alpha-beta is using a form of iterative deepening in addition to static node ordering. In searching to depth d, it starts with the subtree generated by its previous move search, which is to depth $d-2$ in the current subtree, since there was also a move by the opponent. This tree is reordered based on the backed-up values from the previous search. Thus, alpha-beta benefits from the improved node ordering of iterative deepening, without incurring any additional overhead, since the previous search had to be performed for the previous move decision anyway.

Note that other techniques, such as transposition tables, refutation tables, the killer heuristic, and the history heuristic [20], are not applicable to random trees, since they are trees and not graphs, and duplicate moves and nodes are very rare. This is true to a lesser extent in Othello, since there are very few cycles, and the same move often has very different effects in different states. We did experiment with the use of aspiration windows [8], but there was no improvement at shallow depths, and only slight improvement at greater depths. Since this technique requires careful tuning of the position and width of the aspiration window, with the ideal values depending on the branching factor and search depth, we did not consider it further. Thus, we implemented alpha-beta as efficiently as was practical for our experiments.

7. Decision quality

While efficiency is a consideration, the most important property of a selective search algorithm is the quality of the move decisions it makes. One measure of decision quality is the percentage of time that an optimal decision is made. Unfortunately, optimal decisions can only be computed for relatively shallow trees. Thus, for these experiments, we used random trees of depth ten, and treated the terminal heuristic values as exact values. Thus, we can compute the exact minimax value of the entire tree, and hence the optimal move decision at the root. Then, by comparing the optimal move to the move chosen by a particular search algorithm, we can calculate the percentage of optimal decisions made by the algorithm.

We compared the decision quality of best-first minimax to that of alpha-beta. Both algorithms were implemented as efficiently as possible. Best-first minimax was implemented using exponential space, but using the recursive control structure of RBFMS. Alpha-beta was implemented with static node ordering. The measure of running time was the number of new node generations. While best-first minimax incurs additional overhead whenever the principal variation changes, this overhead consists of returning back up the tree, and then following pointers in the stored tree down to the new principal leaf. The significance of this overhead depends on the relative cost of following a pointer compared to generating and evaluating a new board position. In both our random tree and Othello experiments, this pointer-following overhead was not significant, and both best-first search and alpha-beta took rougly the same amount of time per new node generation.

7.1. Random trees with uniform branching factors

For the first set of experiments, we generated incremental random game trees of depth ten, with uniform branching factors of two, three, five, and ten. The heuristic values of nodes at depth ten were treated as exact values. The move from the root to the child that leads to the exact minimax value was computed using alpha-beta minimax. We then compared the decision quality of alpha-beta minimax to that of best-first minimax at various search depths less than or equal to ten. For each algorithm and search depth, we computed the move recommended by the algorithm, and then determined the percentage of time that the algorithm made the optimal move decision. We generated 10,000 different trees, with initial random seeds ranging from 0 to 9999, for each algorithm and search depth.

Fig. 10 shows the results. The horizontal axis is the average number of new nodes generated per move by each algorithm, on a logarithmic scale, and the vertical axis is the percentage of time that the algorithm made the optimal move. Different data points were generated by running each algorithm at all search depths from one to ten. There are four pairs of lines shown, each corresponding to a different branching factor. Within each pair, the top line is for best-first minimax (BF), and the bottom line is for alpha-beta (AB). All of the 95% confidence intervals for these data points are less than plus or minus 1%.

The first data point for each line represents a depth-one search. At depth one, alpha-beta and best-first minimax behave identically, moving to the child of the root with the best static value. For example, in a binary random tree, the child of the root with the best value is actually the best child about 70% of the time in a depth-ten tree. The decision quality of a depth-one search decreases with increasing branching factor, because there are more moves to choose from.

The last data point on each alpha-beta line represents a search to depth nine, since a depth-ten alpha-beta search always returns the optimal move, and hence has a decision quality of 100%. Note that a depth-nine alpha-beta search in a depth-ten tree makes the best move less than 90% of the time.

The last data point on each best-first line represents a search to depth ten, but does not compute the exact minimax value of the tree. Recall that a best-first search to depth *d* terminates when the principal variation reaches depth *d*. Since our trees are only ten moves deep, best-first minimax cannot search any further at this point, which is why the best-first lines do not extend as far to the right as the alpha-beta lines.

In between these endpoints, we see that for both algorithms, increasing search depth leads to improved decision quality. Thus, we can confirm that incremental random game trees are not pathological [24]. In general, however, for the same number of

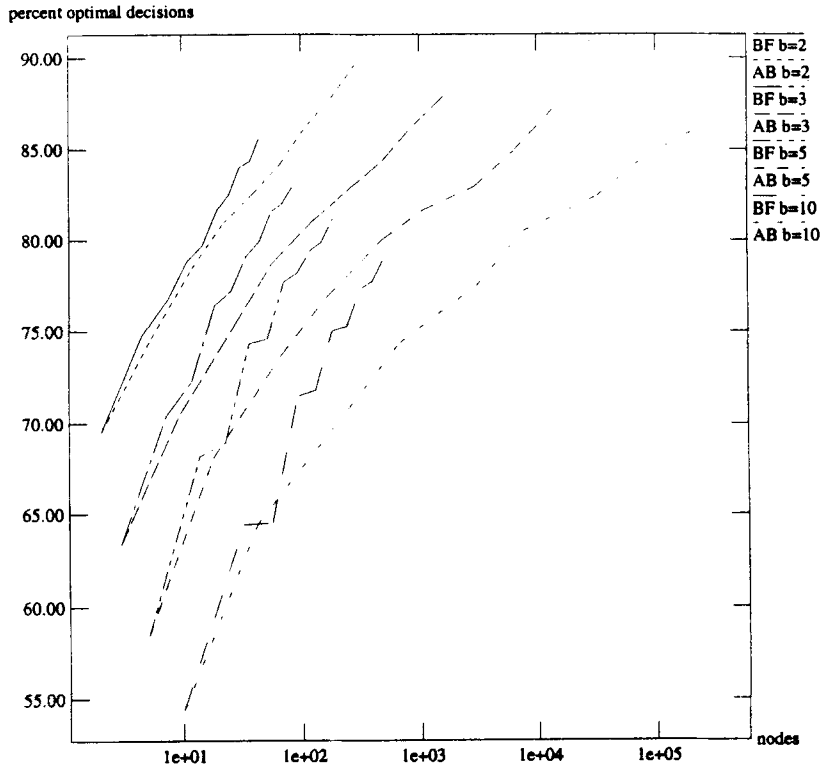

Fig. 10. Decision quality on uniform incremental random trees.

node generations, the decision quality of best-first minimax is better than that of alpha-beta.

7.2. Staircase in best-first decision quality

Another phenomenon exhibited in Fig. 10 is that while the decision quality of alpha-beta improves rather smoothly with increasing depth, the best-first lines exhibit a "staircase" shape, which becomes more pronounced with increasing branching factor. For example, while a depth-two search provides significantly better decision quality than a depth-one search, the improvement of a depth-three search over a depth-two search is much smaller. In general, going from an odd depth to an even depth provides a large improvement in decision quality, while going from an even depth to an odd depth produces a much smaller improvement. This is related to the tempo effect described in Section 2, and can be explained as follows.

Consider the tree fragment shown in Fig. 11. A depth-one search will generate nodes *b* and *c*, and move to the one with the largest value, say node *b*. In a depth-two best-first search, node *b* will be expanded next, generating its children *d* and *e*. Its new minimax value will be the minimum of its children's values. Thus, the expansion of node *b* will

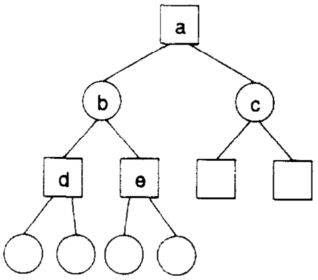

Fig. 11. Example tree fragment to illustrate staircase phenomenon.

tend to lower its value from its static value. This will tend to make it look worse than its brother node *c*, from the perspective of their parent node *a*. Thus, after expanding node *b*, it's likely that node *c* will be the new maximum-valued child of the root, and if so, it will be expanded next. Thus, a depth-two best-first search will tend to expand most of the children of the root to depth two, resulting in significantly improved decision quality. The only way that node *c* would not be expanded is if its initial static value is so low that it is still lower than the backed-up value of node *b*, even after the opponent has a chance to lower the value of node *b*.

Now consider a depth-three best-first search of the tree fragment in Fig. 11. Assume that node *d* is the principal leaf after the search reaches depth two. Expanding node *d* will set its new value to the maximum of its children's values, which will tend to increase its value. This will tend to cause node *e* to be expanded next, which will tend to increase its value as well. Thus, the expansions of both nodes *d* and *e* will tend to increase their values and hence increase the minimax value of node *b*, causing it to remain on the principal variation. Thus, a best-first search to depth three tends to make the same move decision at the root as a best-first search to depth two. A symmetric argument applies to MIN nodes at the root. In general, going from an even depth to an odd depth tends not to change the move decision, and hence provides little improvement in decision quality. This phenomenon is more pronounced with larger branching factors, since taking the maximum or minimum of a larger number of elements tends to have a larger effect on the value of a node.

7.3. Effect of increasing branching factor

Another phenomenon that can be observed from Fig. 10 is that the advantage of best-first minimax over alpha-beta tends to increase with increasing branching factor. This can be explained as follows:

Alpha-beta makes all of its decisions based on the evaluations at the search frontier. Interior node evaluations are used only for node ordering to increase efficiency. Best-first minimax, on the other hand, relies on interior evaluations for decision making, and will not expand certain nodes if their evaluations are poor. Thus, the performance of best-first minimax, as with other selective search algorithms, is much more sensitive to the accuracy of the evaluation function than alpha-beta. The more accurate the evaluation function, the better best-first minimax should perform relative to alpha-beta.

Before a node is expanded, its minimax value is its static value. When a MAX node is fully expanded, its minimax value becomes the maximum of its children's static values, which in a incremental random tree is its own static value plus the largest edge cost assigned to any of its child branches. Thus, the increment to its value is the maximum of b independent edge costs, where b is the branching factor. For a uniform distribution, the maximum of several independent samples has a smaller variance than the original distribution. Furthermore, the variance of the maximum decreases with the number of independent samples. Therefore, in our experiments, the variance of node values decreases with increasing branching factor. With a very large branching factor, the change in a node's value due to its expansion can be predicted fairly accurately, and is the same for all MAX nodes. A symmetric argument applies to MIN nodes, where the change in their values is the minimum of b independent edge costs. Since the change in a node's value becomes more predictable with increasing branching factor, the overall accuracy of the evaluation function also increases with increasing branching factor. Thus, we expect the performance advantage of best-first minimax over alpha-beta to increase with increasing branching factor, as we see in Fig. 10.

7.4. Random branching factor trees

To test our hypothesis that a more accurate evaluation function favors best-first minimax over alpha-beta, we experimented with a less accurate function. One way to decrease the accuracy of the evaluation function in a random tree is to make the number of children of each individual node a random variable, instead of a fixed constant for all nodes in the tree. By allowing varying numbers of children, the increment to the minimax value of a node due to its expansion becomes the minimum or maximum of a random number of samples, which has a higher variance and is less predictable than a fixed number of samples. This also may be a more realistic model of real games, since in most games the number of children of a node varies.

Using the same random seed generated by the breadth-first assignment described previously, we chose the branching factor of each node to be uniformly distributed from one to a maximum branching factor B. By setting the minimum number of children to one instead of zero, we guarantee that our trees still have uniform depth. The branching factor of the root node is always set to the maximum branching factor. Otherwise, decision quality would be heavily affected by how many possible move choices there were at the root. The average branching factor below the root is $(1 + B)/2$. Fig. 12 shows the results of the same experiment described in Section 7.1, but with random instead of uniform branching factors. In addition, we have added a pair of lines for branching factor four.

This figure shows that the evaluation function is less accurate with random branching factors. For example, with a fixed branching factor of two, the child node with the best static value is indeed the best move at depth ten almost 70% of the time. With random branching factors of one or two, however, the child with the best static value is in fact the best move at depth ten only about 63% of the time. This difference is even more dramatic with larger branching factors. At branching factor ten, for example, the accuracy of the evaluation function drops from about 55% to about 25%, when changing

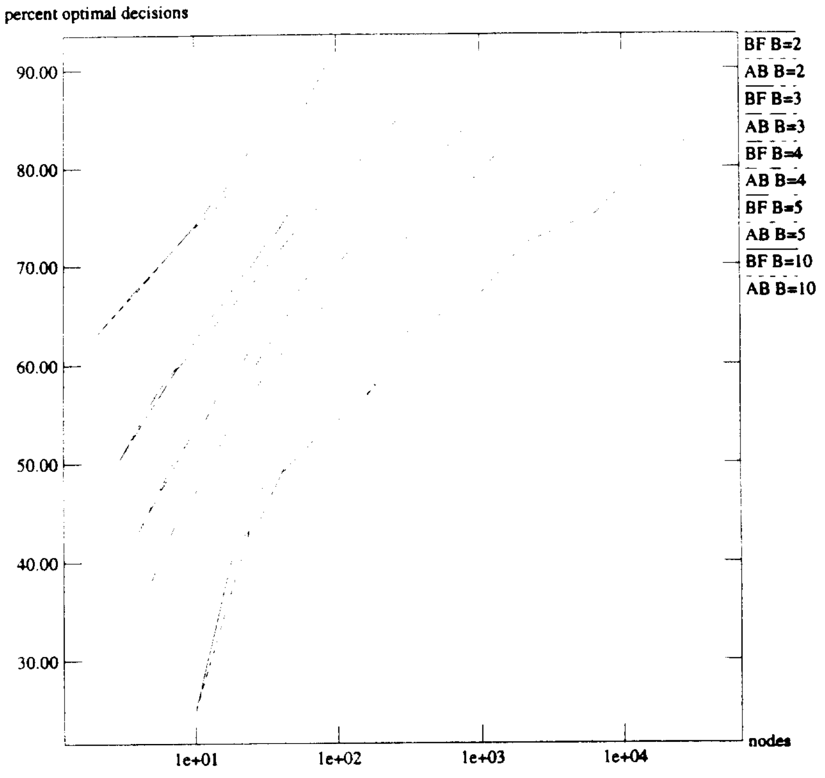

Fig. 12. Decision quality on random branching factor trees.

from fixed to random branching factors. All of the 95% confidence intervals for these data points are less than plus or minus 1%.

Furthermore, as predicted, the performance difference between best-first minimax and alpha-beta is smaller with random branching factors. In fact, we will see that at greater search depths, the decision quality of best-first minimax on random branching factor trees drops below that of alpha-beta.

8. Quality of play

While the quality of individual decisions is an important predictor of performance for a two-player game algorithm, the real test is how well it plays against an opponent in a complete game. We played best-first minimax against alpha-beta both on random game trees and the game of Othello. We first discuss saving the game tree, and then describe the results.

8.1. Saving the tree

A complete game consists of a sequence of moves, which allows a search algorithm the opportunity to save information gathered in the course of making one move, and apply it to subsequent moves as well. Best-first minimax saves its search tree as a pointer structure in dynamically allocated memory. Once it makes a move, it prunes the subtrees of the moves it didn't make, freeing the resulting memory. The opponent then makes a move, and best-first minimax again prunes the subtrees of the moves not chosen by the opponent. The remaining subtree in memory is rooted by the current game state, and best-first minimax begins its search for its next move with this tree. 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. As a result, saving the relevant subtree between moves improves its performance considerably.

In contrast, the standard depth-first implementation of alpha-beta doesn't save the tree from one move to the next. The main reason is that since it searches every move to the same depth, very little of the subtree computed during one move is still relevant after the player's and opponent's moves, as is easily shown. First, assume that alpha-beta doesn't prune any nodes, and the full tree is searched. A search to depth d with a branching factor of b will generate b^d frontier nodes. After the player and the opponent both move, what is left of this tree is a subtree of only b^{d-2} frontier nodes. Thus, the fraction of the original search tree that is still relevant after a pair of moves is only $1/b^2$. In a game such as chess, where the average branching factor is about 35, only 0.08% of the search tree for one move is relevant to the next.

This is a worst-case scenario, however. The more effective alpha-beta pruning is, the larger the fraction of the search tree that can be saved between moves. In the best case, alpha-beta searches the minimal game tree, which has $O(b^{d/2})$ frontier nodes. After each player has moved, the remaining game tree has $O(b^{(d-2)/2})$ frontier nodes. Thus the fraction of the original tree that is still relevant to the player's next move is $1/b$. Again, in the case of chess, this is less than 3% of the original tree.

However, to ensure a fair comparison with best-first minimax, which does save the tree, our implementation of alpha-beta also saves the relevant subtree from one move to the next. The primary advantage of this is improved node ordering. The remaining subtree is reordered based on the heuristic values of the frontier nodes. This allows alpha-beta to prune more of the search tree than it could with just static node ordering alone. This doesn't contradict our previous claim that for a single decision, iterative deepening is not effective, because in actual play, the previous iterations have to be performed anyway for the previous move decisions.

8.2. Random game tree results

In order to complete a random game, the game tree must be artificially terminated at some point. We chose a depth of 100 for the game tree, with each side making 50 moves. At that point, we must decide who won the game. The simplest way to do this is to label positive terminal values as wins for MAX, negative terminal values as wins for MIN, and zeros as draws. The drawback to this scheme is that any particular random game will either be a forced win for MAX, a forced win for MIN, or a draw, depending on whether the exact minimax value of the tree is positive, negative, or zero, respectively.

To remove this bias, we adopt another approach. Every random game is played twice, with each player alternately playing MAX, and moving first. A play of the game consists of a sequence of alternating edge choices by each player until a leaf node is reached. The heuristic value of this leaf is the outcome of the game. The winner of a pair of games is the player playing MAX when the larger outcome was obtained. If the outcome is the same in both cases, the pair of games is declared a tie. Except when both algorithms search to depth one, and hence make the same decisions, ties are rare in practice, and are ignored in the results presented below. This is similar to the idea of duplicate bridge, where the random deal of the cards is factored out by having each side play the same cards.

The next issue is how to give both algorithms the same amount of computation per game. We measured time by the number of new node generations, which is the dominant operation. A tournament consisted of 200 pairs of random games, with initial seeds ranging from 0 to 199. For a given branching factor, we played alpha-beta against best-first minimax, with each algorithm initially searching to depth one. Whichever algorithm generated the fewest total nodes in the last tournament had its search horizon incremented by one in the next tournament. This generated a series of tournaments for each branching factor. For each alpha-beta search depth, we selected the best-first search depth that generated most nearly the same number of nodes, and reported the results for that tournament.

An alternative approach would be to give each algorithm a fixed number of node generations. When best-first minimax exhausted the available node generations, it would make the first move along the principal variation at that point. Alpha-beta would have to use iterative deepening, and when the allocated node generations were used up, make the move recommended by the last completed iteration. Much of the time of alpha-beta would be wasted on the last incomplete iteration, however. Best-first minimax would also suffer from this scheme, in cases where a decision was made shortly after the principal variation shifted from one child of the root to another. On the whole however, such a scheme would probably hurt alpha-beta more than best-first minimax, and hence was not used.

When the principal leaf in a random tree is a terminal node, best-first minimax makes the first move along that path. If the opponent also moves along the principal variation, then best-first minimax will immediately respond with the next move along the path, with no further search. If the opponent makes an unexpected move, however, then best-first minimax will search the new subtree until reaching a new terminal principal leaf node.

Fig. 13 shows the results for incremental random trees with uniform branching factors. The horizontal axis is the search depth for alpha-beta. There are two groups of lines, one on the top and one on the bottom, and the interpretation of the vertical axis is different for the two groups. The bottom group of lines, labelled "BFd", represent the equivalent best-first search depth, with one line for each branching factor from two to five. For each branching factor and alpha-beta depth, the data point on the bottom line gives the best-first search depth that generated most nearly the same number of nodes as the alpha-beta depth. For example, on a random binary tree, an alpha-beta search to depth 10 generates nearly the same number of nodes as a best-first search to depth 52.

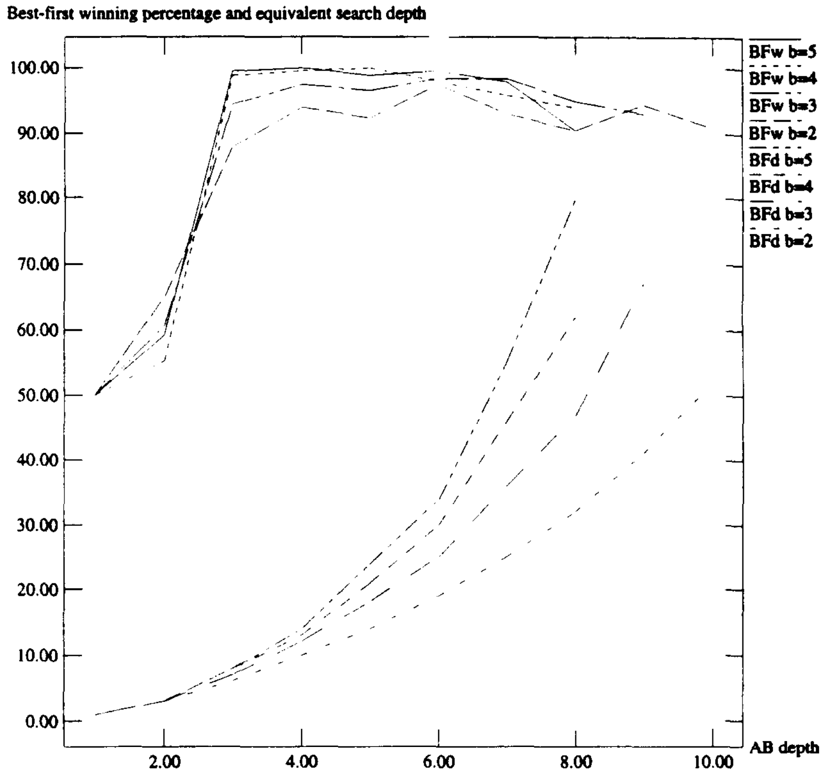

Fig. 13. Best-first minimax performance on uniform random trees.

The first thing to notice is that as the alpha-beta search depths increase, the corresponding best-first depths increase much faster. As an extreme case, in trees of branching factor five, with alpha-beta searching to depth 8, best-first minimax can search to depth 80 while generating roughly the same number of nodes. The reason that all the lines don't go all the way to alpha-beta depth ten is that in order to generate the same number of nodes as alpha-beta, best-first would have to search deeper than 100 moves in some cases. This is impossible, since the games end after 100 moves.

The top group of lines, labelled "BFw", shows the percentage of games won by best-first minimax over alpha-beta, when best-first searched to the equivalent depth shown in the bottom group. Again there are four lines, one for each branching factor from two to five. The first data point on each line represents both algorithms searching to depth one. In this case, both algorithms behave identically, making their moves based solely on the static values of their immediate children. As a result, each pair of games ends up in the same final position, and is declared a tie. For convenience, we represent this as each side winning 50% of the games.

Beyond depth one, however, best-first minimax outperforms alpha-beta. At alpha-beta depth two, best-first searches to depth three, and wins between 55% and 65% of the time. Beyond that, however, best-first completely dominates alpha-beta, in most cases

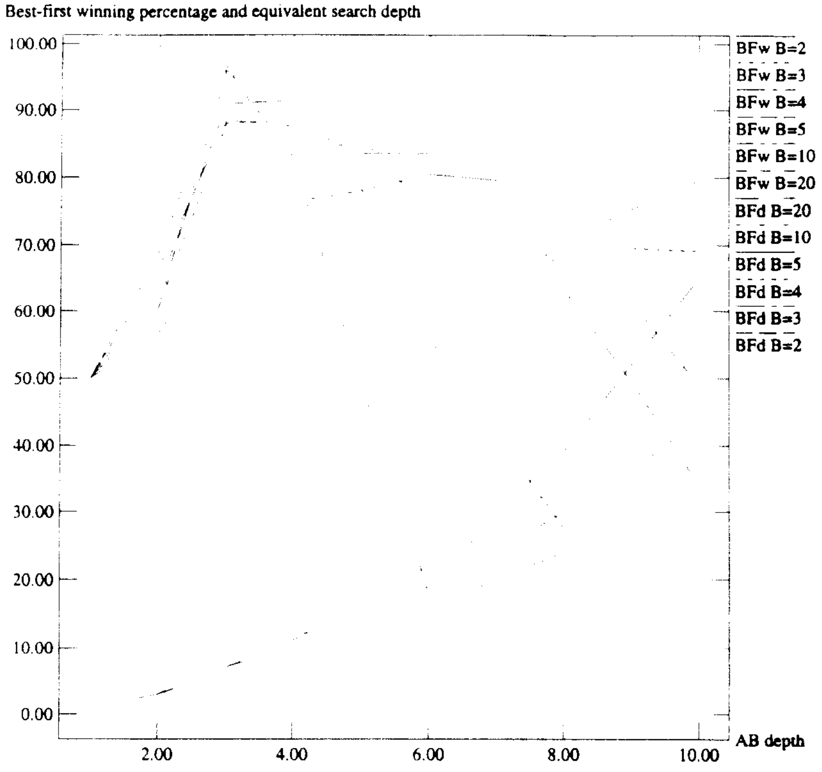

Fig. 14. Best-first minimax performance on random branching factor trees.

winning over 90% of the games, and in some cases winning every game. At greater depths, the relative performance of best-first over alpha-beta begins to drop off slightly. This effect is more prominent in the random branching factor data presented in Fig. 14, and will be discussed in more detail in Section 9. All the 95% confidence intervals for these data points are less than plus or minus 7%.

These results are consistent with the results on decision quality, namely, that best-first minimax is superior to alpha-beta. We also repeated the same experiment on incremental random trees with random branching factors, to test the algorithm with a less accurate evaluation function. In this case, the number of children of each node is a random variable from one to a maximum branching factor B, which ranged from two to twenty. The results from these experiments are shown in Fig. 14, in the same format as Fig. 13. The early termination of some of the lines is due to the excessive time required to run these experiments with the larger branching factors.

For moderate search depths, best-first minimax significantly outperforms alpha-beta. At greater depths, however, this advantage decreases, and eventually alpha-beta outplays best-first. For example, on trees with a maximum branching factor of 20, with alpha-beta searching to depth 6, the best-first horizon that generates most nearly the same number of nodes is 39, and best-first wins only 18% of the games. The depth at which the tables

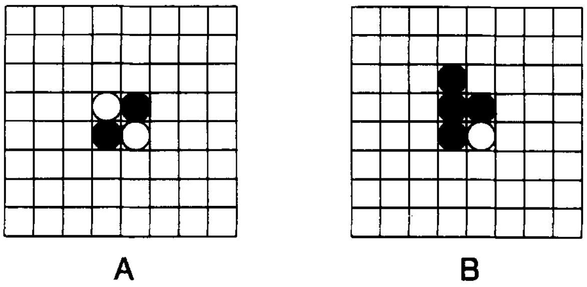

Fig. 15. The game of Othello.

turn decreases with increasing branching factor. Again, all the 95% confidence intervals for these data points are less than plus or minus 7%.

8.3. Othello

While random game trees are a convenient testbed for these experiments, one may question whether the results obtained in random games will apply to real games as well. Of course, any particular game, be it a "real" game or a "random" game, is likely to exhibit different characteristics, and there is no guarantee that results obtained in a real game such as Chess would apply to other games either. In any case, as a further test of our results, we ran a similar set of experiments in the game of Othello.

Othello is a two-player game played on an eight-by-eight board. The game pieces are discs that are black on one side and white on the other, with each color associated with a player. The initial setup of the board is shown in Fig. 15A. The players alternate turns, with black going first. Each move consists of placing a piece of one's own color on the board so that it captures at least one of the opponent's pieces. A contiguous line of pieces, either horizontal, vertical, or diagonal, are captured by bracketing them on both ends along the same line by two opposing pieces, one of which was just played. As soon as the pieces are captured, they are turned over to show the color of the capturing player. For example, Fig. 15B shows the result of a legal first move for black. Every legal move must capture at least one piece. If a player doesn't have a legal move, the opponent goes again. The game ends when neither side has a legal move, usually because the entire board is filled, and the winner is the player with the most discs showing their color.

Othello has received as much attention from the AI community [19,30] as any other game except for Chess. The best Othello programs today are generally considered superior to the best human players, although computers have not been allowed to play in human championship tournaments. For our experiments, we used the evaluation function from Bill, which won first place in the 1989 North American Computer Othello Championship [19]. The reason we didn't compare best-first minimax against the full Bill system is that any performance program has a large number of enhancements, such as thinking on the opponent's time, most of which could be applied to best-first minimax as well. Developing a full-scale performance system around best-first minimax would require significantly more work, and detract from the main goal of evaluating the search algorithm itself.

8.3.1. Endgame play

While the outcome of an Othello game is the difference between the number of discs of each color on the board at the end, this is not a good heuristic function for non-terminal positions, nor are these values comparable to Bill's evaluation of interior nodes. In order for best-first minimax to correctly handle endgame positions, winning terminal positions for MAX are evaluated at ∞, and losing terminal positions at $-\infty$. Tied positions are evaluated at one more than losing positions.

When best-first minimax reaches a terminal position, it will keep searching, since ∞ is the worst-possible outcome for the nearest MIN ancestor of the position, and $-\infty$ is the worst-possible outcome for the nearest MAX ancestor. It will keep searching until it has found a forced winning strategy, or found that all moves result in a loss or tie.

The problem with the endgame play of best-first minimax is that since it searches much deeper than alpha-beta in the same amount of time, it reaches endgame positions much sooner, and long before it can complete the exhaustive search that is triggered by terminal values of ∞ and $-\infty$. Thus, if a position appears to be a win, some time cutoff is required, or the algorithm may exhaust all of its remaining time trying to verify that it is a forced win, resulting in a forfeit on time. Similarly, continuing to search when all terminal positions are losses could consume all the remaining time in a game, also resulting in a forfeit on time. At some point, the search should be terminated, and a move made.

For simplicity in our Othello implementation, when the principal leaf is either at the best-first search horizon or a terminal node, the search is terminated and the first move along that path is made. This could certainly be improved. When the search horizon of the alpha-beta opponent reaches the end of the game, however, best-first minimax also uses alpha-beta to search the rest of the tree. The reason is that when the end of the game is within reach, alpha-beta is an optimal algorithm, and best-first minimax should also use it at that point.

8.3.2. Othello results

To generate different games, all possible first four moves were made. This produces 244 different game boards. Each board was played twice, with each side moving first, in order to eliminate the effects of a particular board favoring either player. Thus, each tournament consisted of 488 games. About 3% of the games resulted in ties, with each player having the same number of discs at the end, but these are ignored in the results presented below.

For each alpha-beta search horizon, we experimentally determined the best-first horizon which took most nearly the same amount of time. In the Othello experiments, actual runtime was used, as opposed to node generations, although node generation and evaluation was still the dominant operation. Alpha-beta saved the relevant subtree from one move to the next, but the time required to save the tree was insignificant, and was more than made up for by the improved node ordering that resulted. Fig. 16 shows the results of the experiment, in a similar format to Figs. 13 and 14. The top and bottom lines of

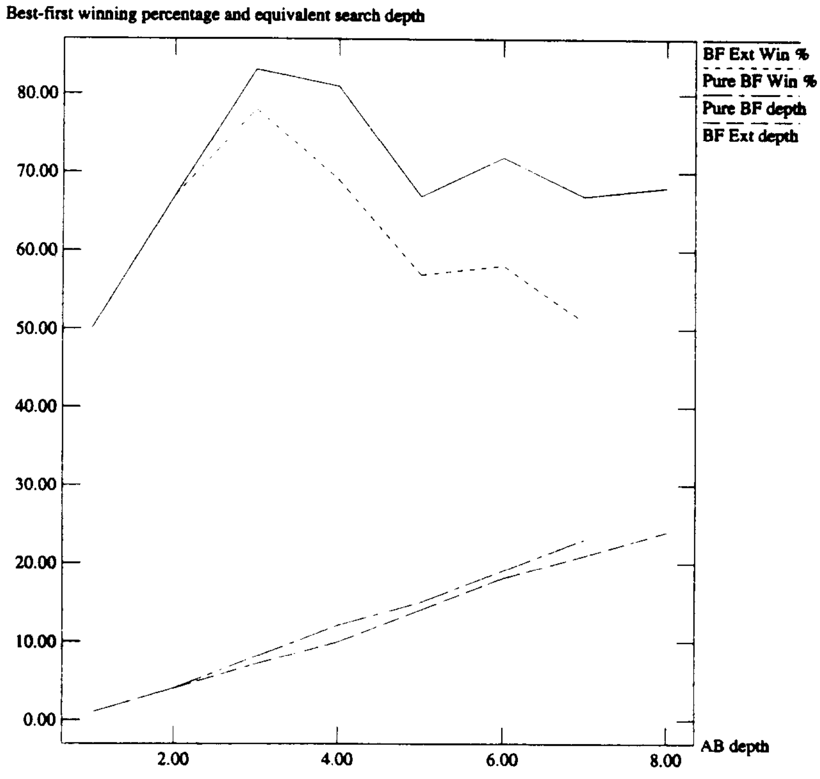

Fig. 16. Performance of best-first minimax in the game of Othello.

the graph will be discussed in the next section, but for now consider only the middle two lines. The lower of these two lines shows the equivalent best-first depth for each alpha-beta depth, and the upper of the two lines shows the percentage of games won by best-first minimax when searching to the equivalent depth.

Both algorithms are identical at depth one. At greater depths, best-first searches much 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. These results are qualitatively consistent with the results from the random game trees with random branching factors shown in Fig. 14. All the 95% confidence intervals for these data points are less than plus or minus 4.5%.

9. Best-first extension

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 many 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, minimax 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 [1].

Best-first extension is governed by 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 the pure alpha-beta opponent. The best-first horizon was set to 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. Fig. 17 shows the results for random game trees with random branching factors, in the same format as Fig. 14.

At alpha-beta depths greater than one, best-first extension performs significantly better than alpha-beta. At increasing depths, there seems to be some degradation in the relative performance of best-first extension, but it still beats alpha-beta at least three out of four games, and often more than four out of five games. All the 95% confidence intervals for these data points are less than plus or minus 7%. We did not run best-first extension on random game trees with uniform branching factors, since even pure best-first search performs so much better than alpha-beta in that case.

The top and bottom lines of Fig. 16 show the performance of best-first extension against alpha-beta on Othello. Again, the initial alpha-beta search depth for best-first extension is one less than the search horizon of the pure alpha-beta opponent. The best performance of best-first extension is against alpha-beta depths of three and four, and the relative performance of best-first drops at alpha-beta depth five. Beyond that, however, its performance appears to stabilize with increasing depth, and best-first extension beats alpha-beta about two out of three games. All the 95% confidence intervals for these data points are less than plus or minus 4.5%.The proximity of the two bottom lines shows that the initial alpha-beta search doesn't take much time away from the best-first extension.

10. Conclusions and further work

We described a very simple selective minimax search algorithm. 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. We also presented a recursive implementation of the algorithm that reduces its space complexity from exponential to linear in the

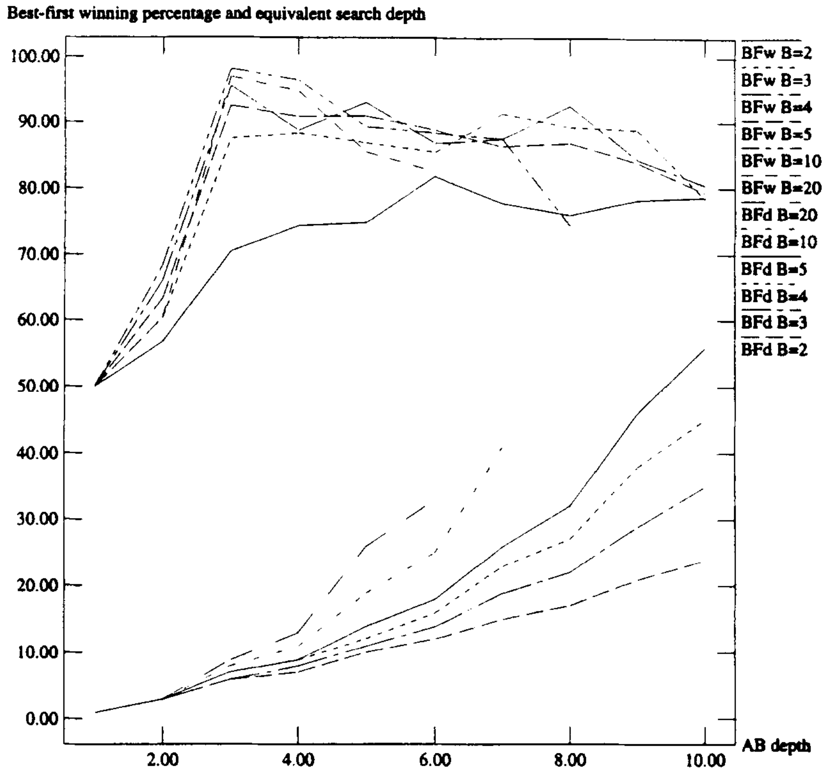

Fig. 17. Best-first extension performance on random branching factor trees.

search depth, but that raises its time complexity from polynomial to exponential. Our actual implementation uses the recursive control structure, but saves the actual tree in memory. In practice, memory is not a problem because in a two-player game, a move must be made every few minutes. Once a move is made, the children of the root that are not chosen can be pruned, and the memory in their subtrees reclaimed. In fact, one advantage of best-first minimax is that it can save much of the tree from one move search, and apply it to subsequent searches.

We experimentally evaluated best-first minimax by comparing it to alpha-beta in two different domains, a class of incremental random game trees and the game of Othello. The random game trees are constructed by assigning random edge costs to a synthetically generated tree, and computing the heuristic value of a node as the sum of the edge costs from the root to that node. We showed how to efficiently generate such trees in a reproducible manner, meaning that whenever a particular node is generated, it always receives the same value. We experimented both on random trees with uniform branching factors, and random trees where the number of children of each node is also a random variable.

On uniform random trees, best-first minimax outperforms alpha-beta, giving both algorithms the same number of node generations. On random trees with random branching factors, and on Othello, best-first outplays alpha-beta up to a given search depth, but starts to lose beyond that depth. The poorer performance of best-first minimax on random trees with random branching factors, compared to fixed branching factors, is due to the fact that the evaluation function is less accurate on the trees with random branching factors. This is because best-first minimax uses the evaluation function both to make its move decision, and to decide where to search.

Finally, we 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 algorithm outperforms alpha-beta roughly two out of three games in Othello, and at least three out of four random games with random branching factors.

Pure best-first minimax performs best when only relatively shallow searches are feasible, and in games with accurate evaluation functions. Thus, it is likely to be most valuable in games with large branching factors, and/or expensive evaluations. These games, such as Go, are those in which computers have been least successful against humans.

Acknowledgements

We are grateful to Kai-Fu Lee for providing the source code for Bill. We would also like to thank Peter Montgomery and Milos Ercegovac for technical assistance in efficiently implementing reproducible random trees. Thanks also to Judea Pearl, Joe Pemberton, and Weixiong Zhang for many helpful discussions concerning this research. This work was supported by NSF Grant No. IRI-9119825, and a grant from Rockwell International.

References

[1] T.S. Anantharaman, A statistical study of selective min-max search in computer chess, Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA (1990).

[2] T. Anantharaman, M.S. Campbell and F.-H. Hsu, Singular extensions: Adding selectivity to brute-force searching, *Artif. Intell.* **43** (1990) 99-109.

[3] H.J. Berliner, The B* tree search algorithm: A best-first proof procedure, *Artif. Intell.* **12** (1979) 23-40.

[4] P.P. Chakrabarti, S. Ghose, A. Acharya and S.C. de Sarkar, Heuristic search in restricted memory, *Artif. Intell.* **41** (1989) 197-221.

[5] S.H. Fuller, J.G. Gaschnig and J.J. Gillogly, An analysis of the alpha-beta pruning algorithm, Tech. Rept., Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA (1973).

[6] L.R. Harris, The heuristic search under conditions of error, *Artif. Intell.* **5** (1974) 217-234.

[7] H. Kaindl, Searching to variable depth in computer chess, in: *Proceedings IJCAI-83*, Karlsruhe (1983) 760-762.

[8] H. Kaindl, R. Shams and H. Horacek, Minimax search algorithms with and without aspiration windows, *IEEE Trans. Pattern Anal. Mach. Intell.* **13** (1991) 1225-1235.

[9] R.M. Karp and J. Pearl, Searching for an optimal path in a tree with random costs, *Artif. Intell.* **21** (1983) 99-117.

[10] R. Keene, B. Jacobs and t. Buzan, *Man v. Machine: The ACM Chess Challenge: Garry Kasparov v. IBM's Deep-Blue* (B.B. Enterprises, Sussex, 1996).

[11] B.W. Kernighan and D.M. Ritchie, *The C Programming Language* (Prentice-Hall, Englewood Cliffs, NJ, 2nd ed., 1988) 46.

[12] D.E. Knuth and R.E. Moore, An analysis of alpha-beta pruning, *Artif. Intell.* **6** (1975) 293–326.

[13] R.E. Korf, Best-first minimax search: Initial results, Tech. Rept., CSD-920021, Computer Science Department, University of California, Los Angeles, CA (1992).

[14] R.E. Korf, Linear-space best-first search, *Artif. Intell.* **62** (1993) 41–78.

[15] R.E Korf and D.M. Chickering, Best-first minimax search: first results, in: *Proceedings AAAI Fall Symposium on Games: Planning and Learning*, Raleigh, NC (1993) 39–47.

[16] R.E Korf and D.M. Chickering, Best-first minimax: Othello results, in: *Proceedings AAAI-94*, Seattle, WA (1994) 1365–1370.

[17] R.E Korf, J. Pemberton and W. Zhang, Incremental random search trees, in: *Proceedings AAAI-94 Workshop on Experimental Evaluation of Reasoning and Search Methods*, Seattle, WA (1994) 15–18.

[18] E.W. Kozdrowicki and D.W. Cooper, COKO III: the Cooper-Koz chess program, *Commun. ACM* **16** (1973) 411–427.

[19] K.-F. Lee and S. Mahajan, The development of a world-class Othello program, *Artif. Intell.* **43** (1990) 21–36.

[20] T.A. Marsland, Computer chess and search, in: S.C. Shapiro, ed., *Encyclopedia of Artificial Intelligence* (Wiley, New York, 2nd ed., 1992) 224–241.

[21] D.A. McAllester, Conspiracy numbers for min-max search, *Artif. Intell.* **35** (1988) 287–310.

[22] C.J.H. McDiarmid and G.M.A. Provan, An expected-cost analysis of backtracking and non-backtracking algorithms, in: *Proceedings IJCAI-91*, Sydney, Australia (1991) 173–177.

[23] D.S. Nau, The last player theorem, *Artif. Intell.* **18** (1982) 53–65.

[24] D.S. Nau, An investigation of the causes of pathology in games, *Artif. Intell.* **19** (1982) 257–278.

[25] M.M. Newborn, The efficiency of the alpha-beta search on trees with branch-dependent terminal node scores, *Artif. Intell.* **8** (1977) 137–153.

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

[27] J. Pearl, *Heuristics* (Addison-Wesley, Reading, MA, 1984).

[28] J. Pemberton and R.E. Korf, Incremental search algorithms for real-time decision making, in: *Proceedings Second International Conference on Planning*, Chicago, IL (1994) 140–145.

[29] R.L. Rivest, Game tree searching by min/max approximation, *Artif. Intell.* **34** (1987) 77–96.

[30] P.S. Rosenbloom, A world-championship-level Othello program, *Artif. Intell.* **19** (1982) 279–320.

[31] S. Russell and E. Wefald, On optimal game-tree search using rational meta-reasoning, in: *Proceedings IJCAI-89*, Detroit, MI (1989) 334–340.

[32] C.E. Shannon, Programming a computer for playing chess, *Philos. Magazine* **41** (1950) 256–275.

[33] J.R. Slagle and J.K. Dixon, Experiments with some programs that search game trees, *J. ACM* **16** (1969) 189–207.

[34] G. Stockman, A minimax algorithm better than Alpha-Beta?, *Artif. Intell.* **12** (1979) 179–196.

[35] W. Zhang and R.E. Korf, Performance of linear-space search algorithms, *Artif. Intell.* **79** (1995) 293–326.