w metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Elsevier - Publisher Connector

ELSEVIER
Artificial Intelligence 137 (2002) 165–196
Artificial Intelligence
www.elsevier.com/locate/artint

Parallel Randomized Best-First Minimax Search ☆

Yaron Shoham, Sivan Toledo *

School of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel

Received 7 April 2001; received in revised form 12 August 2001

Abstract

We describe a novel parallel randomized search algorithm for two-player games. The algorithm is a randomized version of Korf and Chickering's best-first search. Randomization both fixes a defect in the original algorithm and introduces significant parallelism. An experimental evaluation demonstrates that the algorithm is efficient (in terms of the number of search-tree vertices that it visits) and highly parallel. On incremental random game trees the algorithm outperforms Alpha-Beta, and speeds up by up to a factor of 18 (using 35 processors). In comparison, Jamboree [ICCA J. 18 (1) (1995) 3–19] speeds up by only a factor of 6. The algorithm outperforms Alpha-Beta in the game of Othello. We have also evaluated the algorithm in a Chess-playing program using the board-evaluation code from an existing Alpha-Beta-based program (Crafty). On a single processor our program is slower than Crafty by about a factor of 7, but with multiple processors it outperforms it: with 64 processors our program is always faster, usually by a factor of 5, sometimes much more.
© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Two-player games; Heuristic search; Alpha-Beta; Best-first; Chess; Othello; Jamboree search

1. Introduction

We present a new game-search algorithm that we call Randomized Best-First Minimax Search (RBFM). It is the first randomized game-search algorithm. It is based on Korf and Chickering's deterministic Best-First Minimax Search [9]. Randomization fixes a defect in Best-First and makes it highly parallel.

☆ This research was supported by Israel Science Foundation founded by the Israel Academy of Sciences and Humanities (grant number 572/00 and grant number 9060/99).

* Corresponding author.
E-mail address: stoledo@tau.ac.il (S. Toledo).
URLs: http://www.tau.ac.il/~ysh, http://www.tau.ac.il/~stoledo.

0004-3702/02/$ – see front matter © 2002 Elsevier Science B.V. All rights reserved.
PII: S0004-3702(02)00195-9

RBFM is a selective-search algorithm: instead of searching all nodes to a fixed depth, it attempts to focus on relevant variations.

RBFM maintains the search tree and associates both a minimax value and a probability distribution with each node. The algorithm works by repeatedly expanding leaves, thereby enlarging the search tree and updating the minimax values of nodes. The algorithm chooses a leaf to expand using a random walk from the root. At each step in the construction of the walk, the algorithm descends to a random child, where children with high minimax scores are chosen with a high probability.[1] The randomized nature of the algorithm creates parallelism, since multiple processors can work on multiple leaves selected by different random walks. The random-walk leaf-selection method is where the novelty of our algorithm lies.

Deterministic Best-First Minimax Search (BFM) also expands a leaf at each step. The leaf is chosen by recursively descending from a node to to the child with the highest (lowest in MIN nodes) minimax score. Expanding the leaf updates the minimax values along the path to the root. From a given node, BFM always chooses the same child until the score of the child drops below the score of another child. The defect in BFM is that it never attempts to raise the scores of non-best children of a MAX node (or to lower the scores of non-best children of a MIN node). Indeed, BFM may terminate without determining the best move from the root.

RBFM fixes this defect by descending to all children with some probability. This enables the score of the best move to become highest, even if the score of the current highest child does not drop. BFM is really a *highest first* (using the current scores), whereas the desired strategy is *best first* (in the sense of the minimax score after all the leaves are terminal). Thus, the probability with which our algorithm chooses a child is an estimate of the probability that the child is the best.

In this paper we strive to show that the algorithm is efficient and highly parallel. The efficiency of game-search algorithms is determined by the number of move generation and heuristic evaluations. We compare the efficiency of two algorithms by allowing both to visit the same number of nodes (per move or over an entire game) and comparing the quality of the decisions made by each. The number of nodes visited is usually proportional to the number of move generations and heuristic evaluations, and hence to the running times.

The speed and playing strength of parallel game-playing programs is determined both by their efficiency and by their ability to exploit multiple processors. We measure the parallelism in parallel game-search algorithms by measuring the time it takes them solve a set of test positions (for which the winning move is known) with a growing number of processors.

We evaluate RBFM using three models: Incremental random game trees, Othello, and Chess. While Chess and Othello represent more realistic challenges to a game-search algorithms, it is hard to compare search algorithms using such programs. For example, most Chess-playing programs are proprietary, all are complex and all use a variety of

[1] In MAX nodes; children with low minimax scores are chosen with a high probability in MIN nodes. Later in the paper we switch to the simpler negamax notation.

heuristics beyond the search algorithm. Consequently, we also use a synthetic model, incremental random game trees, which by now is a standard benchmark for game-search algorithms. A random search tree is a rooted tree with random numbers assigned to its edges. The heuristic evaluation of a node is defined to be the sum of the values of the edges along the path from the root to the node.

The rest of the paper is organized as follows. Section 2 describes the RBFM algorithm. We explain how RBFM chooses the leaf to expand and discuss our model of score distribution and child selection. In Section 3 we show that RBFM is an efficient search algorithm by comparing it to Alpha-Beta for incremental random game trees. We show that RBFM makes better decisions than Alpha-Beta and outperforms Alpha-Beta in actual games. Section 4 evaluates RBFM in the game of Othello. We show that RBFM outperforms both Alpha-Beta and BFM. Section 5 evaluates RBFM for the game of Chess. We implemented a Chess program using the RBFM algorithm, and we compare it to Crafty, a program that uses Alpha-Beta search. Section 6 shows that RBFM can effectively utilize many processors. For random trees, RBFM achieves a better speedup than Jamboree search (a parallel version of Alpha-Beta). For Chess, we show that RBFM speeds up almost linearly with as many as 64 processors; our program is faster than Crafty. Section 7 presents the conclusions of our research and surveys related work.

2. The RBFM algorithm

2.1. RBFM data structures

RBFM keeps the search tree in memory. Each node stores a score, which represents the value of the game for the side to move at the node. For the leaves of the tree, RBFM uses a *heuristic function* that is based only on the static state of the game that the leaf represents. For nodes that have children, we use the negamax notation: The score of a node is the

```
RBFM(root)
while (STOPPING-CRITERION() == false)
   EXTEND-NODE(root)

EXTEND-NODE(node v)
while (v is not a leaf)
   v = CHOOSE-RANDOM-CHILD(v)
EXPAND-LEAF(v)
BACKUP(v)

BACKUP(node v)
while (v != NULL)
   v.score = NEGAMAX-SCORE(v)
   v.distribution = UPDATE-DISTRIBUTION(v)
   v = v.parent
```

Fig. 1. The RBFM algorithm. The procedures that are not described here will be explained later in the paper.

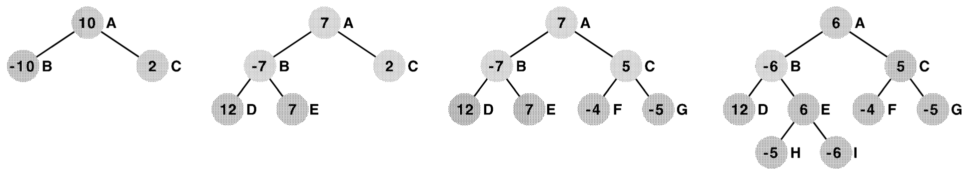

Fig. 2. An example that shows how RBFM works. In node expansion 1 (left), the root is expanded, and its children are assigned their heuristic values. The score of the root is updated according to the negamax rule. In node expansion 2 the algorithm makes a random choice between B or C. Node B has a lower score, and is therefore more likely to be chosen. Node expansion 3 shows that RBFM can explore non-lowest-scoring children. In node expansion 4 RBFM chooses the best child twice: first B and then E.

maximum of the negated scores of its children. In addition to the score, each node stores a distribution function. Section 2.6 describes how we assign a distribution to a node.

2.2. How RBFM works

RBFM is an iterative algorithm. Each iteration starts at the root, and performs a random walk to one of the leaves. In each step the next node in the walk is chosen randomly from the children of the current node. This random decision uses the distribution associated with each node. Children with a low negamax score are more likely to be chosen. Section 2.4 describes exactly how the random choice is done.

When the algorithm reaches a leaf v, that leaf is expanded: Its children are generated and assigned their heuristic value. The algorithm then updates the score of v using the negamax formula. The updated score is backed up the tree to v's parent and so on up to the root. We also update the distributions of the nodes along the path, as described in Section 2.6. Fig. 1 presents a pseudo code of RBFM and Fig. 2 shows a simple example.

From here on, we refer to an iteration of RBFM as a *node expansion*. In this terminology, a node expansion includes the random walk to a leaf, the expansion of the leaf, and score updates up the tree. We use this terminology to distinguish RBFM's node expansion from Alpha-Beta's iterations, which refer to a complete search to a given depth.

2.3. A pruning rule

In some cases expanding a node and updating its minimax score cannot change the decision at the root, no matter how large the change in the node's score. RBFM avoids expanding such nodes. Specifically, we choose deterministically the lowest scoring (best) child c of a node v when choosing any other child c' cannot change the decision at the root until the score of c changes.

Let r denote the root of the search tree. The algorithm is either exploring the subtree rooted at the lowest scoring child c_l of r (the current best move) or it is exploring the subtree rooted at c_o, any other child of r. In the first case, the decision at the root changes only if the algorithm raises the score of c_l; in the second case, only if it lowers the score of c_o. We can raise the score of a node by lowering the score of *any* of its children below that of the lowest scoring child. On the other hand, we can lower the score of a node only by raising the score of its *lowest scoring child*. Therefore, at any node v the algorithm

knows whether it must lower the score of v (choose deterministically) or raise it (choose randomly) to change the decision at the root.

We implement this pruning rule by replacing EXTEND-NODE with either EXTEND-UP, when we want to raise the score of v, or EXTEND-DOWN when we want to lower it. The procedure EXTEND-UP chooses a child randomly like EXTEND-NODE, but it calls EXTEND-DOWN with the selected child as argument. The procedure EXTEND-DOWN, on the other hand, always calls EXTEND-UP with the lowest scoring child as argument.

This pruning rule is reminiscent of Beta pruning in Alpha-Beta search [8]. In both cases the algorithm avoids searching a subtree that is irrelevant to the decision at the root.

2.4. Child selection

The motivation of BFM (and RBFM) is to examine the best move when analyzing a variation. In a minimax (or negamax) game, the quality of a position for a player is determined by the best move the opponent can make from that position. Other moves have no effect on the quality of the position. Therefore, to determine the quality of a position, it suffices to explore the best move from it. The problem, of course, is that we do not know which move is best. Both BFM and RBFM use partial information about the game to guess which move is the best.

BFM simply uses the current negamax scores to guess which move is best. RBFM uses a more sophisticated strategy. We define the *real score* of a node to be the negamax score of the node when we search to depth 10.[2] The idea is that this value represents the outcome of a deep exhaustive search and should therefore enable RBFM to make excellent decisions. But RBFM does not know the real score; it tries to estimate it probabilistically.

RBFM treats the real score of a node as a random variable. RBFM uses information about the current tree to estimate the distribution of this random variable. Specifically, we use the size of the sub-tree rooted at the node, the number of children of the node, and its negamax score to estimate the distribution. In other words, the real score of each child is a random variable whose distribution we estimate. When RBFM needs to choose the move to explore, it uses the following rule: *A move is chosen with the probability that its real score is lowest among its siblings* (i.e., that it is the best).

We do not actually compute these probabilities. Instead, we sample the real score of the children using the estimated distributions and select the child whose random real score is lowest. We assume that the real scores of the children are independent random variables and that is how we sample them. This is not always true in practice since errors in the static evaluation function tend to be correlated for siblings, but assuming independence simplifies the algorithm. The same assumption simplifies other search algorithms [2,15].

2.5. Summary

Fig. 3 presents the full RBFM algorithm. Each node expansion starts at ROOT-DECISION, which chooses a child to explore. If the score of the chosen child is lowest

[2] Any other fixed value can be used. We use 10 in the parameter-estimation, so that is why we define the real score with depth 10. Small values result in unstable scores, and large values are impractical for our experiments.

```
CHOOSE-RANDOM-CHILD(node v)
Let V_1, ..., V_n be the children of v.
Generate n random variables, X_1, ..., X_n, using the distributions described at V_1, ..., V_n.
Find i such that X_i is minimal.
return V_i

EXTEND-DOWN(node n)
if (n is a leaf)
  EXPAND-LEAF(n)
  BACKUP(n)
else
  v = BEST-CHILD(n)
  EXTEND-UP(v)

EXTEND-UP(node n)
if (n is a leaf)
  EXPAND-LEAF(n)
  BACKUP(n)
else
  v = CHOOSE-RANDOM-CHILD(n)
  EXTEND-DOWN(v)

ROOT-DECISION(root)
v = CHOOSE-RANDOM-CHILD(root)
if (v == BEST-CHILD(root))
  EXTEND-UP(v)
else
  EXTEND-DOWN(v)
```

Fig. 3. The complete RBFM algorithm, including the pruning rule and child selection.

among its siblings (it is the current best move), the algorithm tries to increase the score, thereby making it less attractive. Otherwise, the child is a move that currently seems inferior. The algorithm tries to lower the score of the child, so that it would be more competitive. Note that calling EXTEND-UP does not mean the score of the node score is going to raise; it just represents the local objective of the search. It is possible for the score the actually drop after a leaf is expanded. The same goes for EXTEND-DOWN.

When the node expansion reaches a leaf, it calls EXPAND-LEAF, which generates the children of the leaf and evaluates them. The new score is backed up the tree and the distributions along the path to the root are updated.

The whole process is repeated until some stopping criterion is met. For example, the search can be stopped if the tree has reached a certain size or depth. We might want to stop the search based on time-management decisions or because one child of the root emerges clearly as the best move, and so on.

The two strategies used by RBFM, EXTEND-UP and EXTEND-DOWN, are somewhat similar to the strategies used by Berliner's B* algorithm [1]. B* stores in each node an upper bound and a lower bound on the value of the real score. B* uses the *prove-best*

strategy to raise the lower bound of the best child, and the *disprove-rest* to lower the upper bound of other children. While B* tries to *separate* the best move from the other moves, RBFM always questions the current order. Its strategies can be described as *disprove-best* or *prove-rest*.

2.6. Estimating node distributions

We now describe a simple heuristic way to estimate the distribution of real scores. We assume that the distribution of the real score comes from a simple family of 1-dimensional distributions, either normal, exponential or chi-square, which are characterized only by a mean and a variance. We estimate the mean and the variance of node v's distribution using three parameters: v's minimax score m_v, the number of children d_v that v has, and the number s_v of times that v has been expanded. More specifically, we use estimates of the following form. The estimated mean μ has the form $\mu = m_v + f(s_v, d_v)$, where f is a function represented by a lookup table, and the estimated variance has the form $\sigma^2 = g(s_v, d_v)$, where g is another function represented by a lookup table. In other words, we use s_v and d_v to shift the mean from the minimax score and to determine the variance.

We estimate the functions f and g using a sampling technique. We generate game positions (random positions for random game trees, positions from several actual games for Chess). We use Alpha-Beta to depth 10 to determine the real score of each position. For each position, we expand the root v repeatedly using RBFM. Before the first expansion and after each expansion, we obtain a minimax score for (d_v, s), where s is the number of expansions so far. We compute the difference δ between the real score and the minimax score and store δ in the set of samples for the pair (d_v, s). After collecting samples from all the game positions, we use the differences to estimate f and g.

Since we use RBFM to perform the node expansions from which we estimate f and g, we must bootstrap the process somehow. That is, we must start with some initial f and g. We start with a trivial distribution, either best-first ($f = g = 0$) or a nearly uniform distribution ($f = 0$ and $g = M$ for some large number M). We estimate f and g repeatedly using the technique of the previous paragraph until we observe convergence, normally after 3 or 4 repetitions.

We determine the family of distributions based on manual inspections of histograms of scores. We found that normal distributions and chi-square distributions work well for random game trees and for Othello. We found that for Chess, chi-square, normal, and exponential distributions all vanish too quickly. We addressed this using an ad-hoc approach that we describe in Section 5.4.

3. Assessing the efficiency of RBFM: random trees

We show that RBFM is efficient by comparing it to Alpha-Beta search [8]. Alpha-Beta is widely considered to be one of the most efficient search algorithms and is used by nearly all game-playing programs. In this section we show that on a class of artificial games, called *incremental random game trees,* RBFM outperforms Alpha-Beta. Section 4

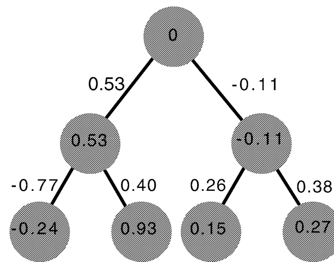

Fig. 4. An incremental random game tree with uniform branching factor $b = 2$. The edges have random values and the static evaluation of each node is the sum of the edges along the path from the root to the node. The static evaluation at the leaves is considered to be their exact value.

compares RBFM to Alpha-Beta on Othello, and Section 5 compares RBFM to Alpha-Beta on Chess.

We also show that RBFM outperforms BFM in the sense that it can effectively utilize more search time.

Section 3.1 describes the methodology that we use. We describe the incremental random game trees model and discuss its properties. We also discuss how to measure the effectiveness of a search.

Section 3.2 describes our experiments and their results. We conducted four experiments. The first experiment tries to approximate the distribution of the real score. Experiments 2 and 3 measure the decision quality of RBFM as a function of the search depth and of the number of generated nodes. Experiment 4 compares RBFM, BFM, and Alpha-Beta over an entire game. Our implementation of Alpha-Beta uses static move ordering. In Experiment 4, it uses the scores from the search in the previous move to order moves. Static move ordering was shown to be about as good as iterative deepening for incremental random trees [9].

3.1. Methodology

We show that given the same computational resources, RBFM outperforms both Alpha-Beta and BFM on incremental random game trees. This family of random trees is often used to assess game search algorithms (see [9]). Korf and Chickering show [9] that for random-branching game trees, Best-First Minimax is inferior to Alpha-Beta, especially for large search depths.

We use the same experimental methodology as Korf and Chickering. Like them, we compare the effectiveness of our algorithm to that of Alpha-Beta in terms of both decision quality (how often an algorithm chooses the best move) and overall play. We repeated exactly some of the experiments that Korf and Chickering describe to ensure that we get the same results. We did.

3.1.1. Incremental random game trees

An *incremental random search tree* is a rooted tree with random numbers assigned to its edges. We use independent random values from a uniform distribution in $[-1, 1]$. The static evaluation of a node is defined to be the sum of the values of the edges along the path from the node to the root. The static evaluation at the leaves is considered to be their

exact value. Thus the closer we are to the leaves, the more exact the evaluation is. A tree can have a *uniform branching factor* if all nodes have exactly b children (except the leaves, of course), or *random branching*. Like Korf and Chickering, we use independent uniform branching factors between 1 and b, except that the root always has b children (otherwise some trees, those with low-degree roots, are easier).

Incremental random game trees are a convenient model for testing search algorithms for several reasons:

- They are easy to generate.
- Their static evaluation function is simple but behaves much like that of actual game-playing programs. Evaluation functions of game-playing programs are accurate near terminal leaves, and errors in the evaluation of a node are correlated with the errors in the evaluation of its parent. But evaluation functions of game-playing programs are typically complicated and are kept secret.
- Real games have a fixed game tree. Incremental random search trees allow us to set the branching factor, evaluation function accuracy, and so on.

Random trees were used as test models by Berliner, Karp, Nau, Newborn, Pearl, and others (see [9] and the references therein).

One problem with random game trees is that some of the techniques that have been developed for real games do not work on random trees. Killer moves and transposition tables, for example, are effective techniques to improve the efficiency of search algorithms but they cannot be employed on random trees. In particular, techniques that attempt to identify the best child of a node improve the efficiency of Alpha-Beta search and its variants, but many of them do not perform on random trees as well as they do on Chess, for instance.

We need a way to reproduce the same game tree in runs in which it grows in different ways. Therefore, we cannot assign edges values from a pseudo-random sequence as we encounter the edges. Instead, we use a method proposed by Korf and Chickering. We use breadth-first search numbering to order the edges. Edge number n is assigned the nth number from a pseudo-random sequence. Using a linear congruential pseudo-random number generator, it is easy to compute the nth number in the sequence directly. (Beware of simpler ways to generate random trees; Korf and Chickering also describe a naive way that results in trees that are not random.)

It appears that the search problem is too easy when the branching factor of the tree is uniform. Even a simple algorithm such as BFM outperforms Alpha-Beta on trees with uniform branching factors. Trees with random branching factors are more difficult because their heuristic evaluation is less accurate. This makes BFM inferior to Alpha-Beta [9]. Korf and Chickering [9] explain this phenomenon by observing that the variance of node scores after a search to a given depth is smaller in uniform-branching trees than in random ones.

Hence, from now on we focus mostly on non-uniform game trees.

3.1.2. Measuring efficiency

We measure the efficiency of a search algorithm in terms of the number of generated nodes. In practice, game-playing programs usually spend most of their time on move

generation and position evaluation, which are linear in the number of generated nodes. Incremental random game trees are an exception: Generating moves is efficient and position evaluation takes only one operation, but maintaining the tree and selecting nodes is relatively time consuming. Therefore, the number of generated nodes on random trees is a better predictor of the running time in real games than the running time on random trees.

3.2. The experiments and their results

3.2.1. Experiment 1: distribution approximation

In order to use RBFM, we need to find an approximation for the distribution of the real score. As described in Section 2.6, we use a distribution for each node that depends on the negamax score, the number of children and the number of extensions. In Experiment 1 we used the sampling technique described in Section 2.6 to generate histograms of the distribution for different branching factors. We noticed that normal and chi-square distributions seem to approximate the distributions well, so these were used in the following experiments.

3.2.2. The results of Experiment 1

Fig. 5 shows examples of histograms of difference between the real score and the minimax score.

We noticed that the mean of the distribution differs from the real score (that is the reason we use translation, as described in Section 2.6). For example, nodes which were not expanded are usually too pessimistic about their value. The magnitude of the translation decreases as the number of extensions grows. We have also noticed that when the number of children at the root is small, the phenomenon is reversed. For example, when the root has only one child, it is forced to choose it. The static value of such nodes is usually too optimistic.

For small branching factors, we approximate the distribution by normal distributions. We compute the mean and the variance of the samples, and then use normal distributions

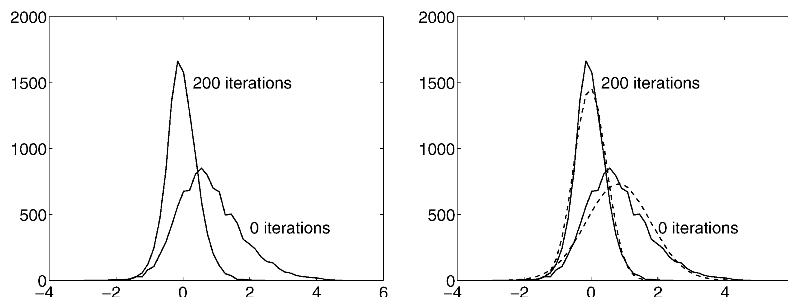

Fig. 5. Experiment 1—Histograms of the distance from the real score, for random trees with non-uniform branching factor $b = 5$ and 5 children at the root. The left diagram shows the histograms after 0 and 200 node expansions. As explained in the text, after 0 node expansions the score is likely to be pessimistic and inaccurate. After 200 node expansions, the score is more accurate and the variance is smaller. The dashed graphs on the left approximate the histograms using normal distributions with the same mean and variance.

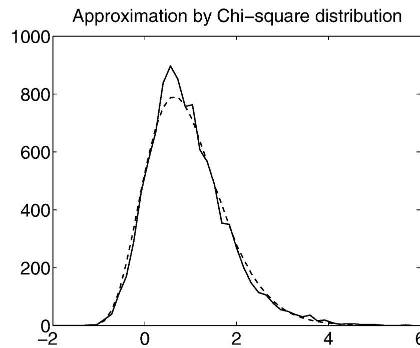

Fig. 6. Experiment 1—Histogram of the distance from the real score, for random trees with non-uniform branching factor $b = 10$ and 10 children at the root. The solid line represents the histogram after 0 node expansions. The dashed line shows a chi-square distribution (with parameter 16), translated to have the same mean and variance.

with the same parameters. The left diagram in Fig. 5 shows such approximation. The matching between the distributions is fairly good.

For trees with large branching factor, the distribution seems to be better approximated by chi-square distribution, as shown in Fig. 6. In our experiments we used normal distribution for small branching factors and chi-square distribution for large branching factors. For medium branching factors, we used a weighted average of these distributions. We do not claim that these choices are optimal in any formal sense. We merely note that our approximation fits the samples and produces better results than using a fixed distribution.

3.2.3. Experiments 2 and 3: decision quality

Experiment 2 evaluates the decision quality of RBFM in a single turn. We compare the probability of finding the best move for RBFM, BFM, and Alpha-Beta, when all three algorithms have the same computational resources. We generated 10,000 random game trees of fixed depth, with random branching factor b of 3, 5, 10 and 20. The depth of the trees was 10, except for branching factor $b = 20$, where the depth was 8. The heuristic values of the leaves were treated as exact values. We used Alpha-Beta to determine the best move at the root. For each random tree, we ran Alpha-Beta at all search depths, from one to nine (a search to depth ten would always find the best move). For each search depth, we recorded whether it found the correct move and the number of generated nodes. We then ran RBFM and BFM until the same number of nodes was generated, at which point we stopped the algorithm and checked whether it had found the correct move.

When the game tree is finite, RBFM and BFM can reach terminal nodes and find their exact value. In our implementation, they mark such leaves as *solved*. The backup function is also modified: if all the children of a node become solved, the node is marked as solved too. In future node expansions, when the procedure CHOOSE-RANDOM-CHILD is called, solved nodes are never explored. One may claim that Experiment 2 is unfair, because RBFM and BFM are allowed to examine the exact values of the leaves, while Alpha-Beta cannot. To address this concern, we performed another experiment. In Experiment 3, for

each Alpha-Beta depth, RBFM and BFM were allowed to run until their principal variation reached the Alpha-Beta depth, and they were then aborted. This ensures that RBFM and BFM see no deeper than Alpha-Beta. In this experiment the three algorithms generate a different number of nodes. We compare the success ratio and the number of generated nodes for all branching factors and search depths.

3.2.4. The results of Experiments 2 and 3

Fig. 7 summarize the results of Experiment 2, which evaluates the decision quality of RBFM and BFM when they are limited by the same number of node generation as Alpha-Beta. The results show that RBFM makes better decisions than both Alpha-Beta and BFM, given the same number of generated nodes. At shallow depths RBFM and BFM are roughly equally effective, but when the search deepens BFM's decision quality remains almost constant, whereas RBFM decision quality improves.

The results of Experiment 3, shown in Fig. 8, show that RBFM generates fewer nodes than Alpha-Beta in order to reach the same decision quality. For example, in order to reach a decision quality of 72% in trees with branching factor 10, Alpha-Beta must search to depth 6 and generate 2096 nodes on average. On the other hand, RBFM searches to depth 9 and generates only 549 nodes.

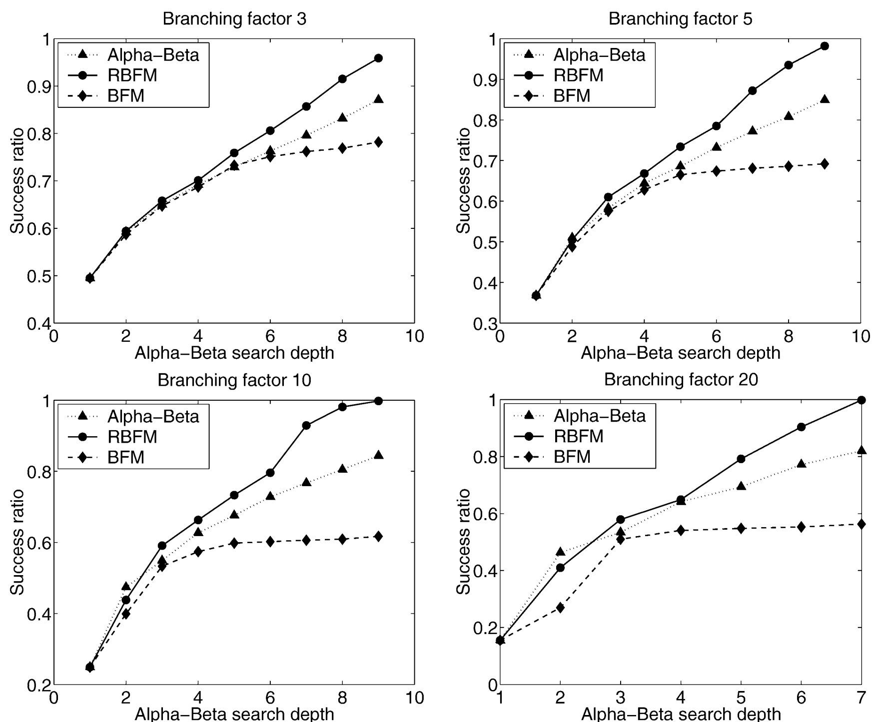

Fig. 7. Experiment 2. The plots show the success ratios of RBFM, BFM, and Alpha-Beta for each branching factor and each Alpha-Beta search depth.

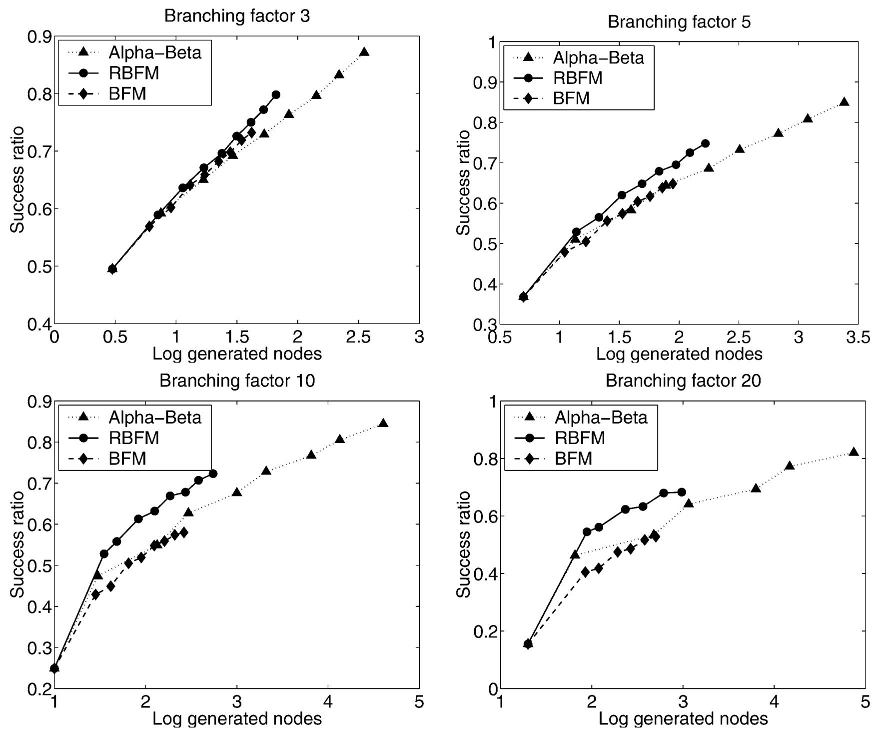

Fig. 8. Experiment 3. Each plot shows the number of generated nodes (on a logarithmic scale) and the success ratio for each search depth.

The results also show that BFM's stopping criterion prevent it from reaching high-quality decisions, although for small depths it is about equally effective as Alpha-Beta and RBFM.

We conclude that for incremental random search trees, RBFM makes better decisions than Alpha-Beta and BFM.

Furthermore, in our experiments Alpha-Beta was always allowed to finish its search. If both algorithms were limited by a fixed number of generated nodes, Alpha-Beta could search to a certain depth, but would not have enough time to search one move deeper. RBFM can be stopped at every node expansion, so it exploits the allotted time better.

We repeated Experiments 2 and 3 on random trees with uniform branching factors. The results were qualitatively similar to the results presented here, so we omit further details. One minor difference between the results on uniform- and random-branching trees is that on uniform trees Alpha-Beta outperforms RBFM at small search depths.

3.2.5. Experiment 4: performance in a full game

So far we have discussed the decision quality of a single move. We are also interested in how the algorithm plays a complete game. In a full game, an algorithm can use the computation it has done in previous turns to speed up the current computation. After a player has moved and its opponent replied, a sub-tree of the previous search is relevant

to the current search. Alpha-Beta implementations often use this sub-tree to get improved node ordering. RBFM can simply continue to expand the sub-tree. Selective algorithms have the potential to use a larger part of the tree, compared to Alpha-Beta, because Alpha-Beta spends a lot of time proving that some moves are not optimal. Such computation is irrelevant in subsequent turns. Selective algorithms try to focus on reasonable variations, so a larger part of the tree can be used.

Experiment 4 compares RBFM to Alpha-Beta over a full game using the methodology described in [9]. We generated incremental random search trees of depth 100 and used each one to play two games. In the first game, Alpha-Beta was the first player, and RBFM replied. This game ends when RBFM makes its 50th move. At this point, we take the node's heuristic value as the outcome of the game. Note that since an even number of moves was played, this value represents the game's score from the first player's point of view. In the second game, RBFM gets to be the first player, and the process is repeated. We then compare the value reached in the end of the first game to the value of the second game. The algorithm that reached the higher number is declared to be the winner.

When Alpha-Beta plays, it searches to a fixed depth. RBFM aborts the search when its principal variation reaches a certain depth. For each branching factor and each Alpha-Beta depth, we determined (using simulations) the search depth of RBFM that results in approximately the same number of generated nodes, when summed over all the games. This was the depth used by RBFM.

Toward the end of the game we use a slightly different stopping criterion for RBFM. During most of the game, RBFM aborts when its principal variation reaches the search depth. When the remaining depth of the game is smaller than the search-depth, this stopping condition is not met, and RBFM might do a full search before terminating. This would take too much time. Once again, we used the rule described in [9]: When the principal variation reaches the leaves, the computation is aborted. When other variants reach the leaves, they mark nodes as solved. Note that this approach is somewhat unfair to RBFM: in the final stages of the game, when the remaining depth is less than Alpha-Beta's search depth, Alpha-Beta performs a full search, finding the optimal moves, while RBFM aborts the search when its principal variation reaches a leaf. The last stages of the game have a large impact on its result; we chose this solution nonetheless to remain consistent with the methodology in [9].

We compare RBFM not only to Alpha-Beta, but also to BFM.

We used tournaments with the following structure to compare the three algorithms. For each branching factor, and for each Alpha-Beta Search depth, we found (using simulations) the search depths that result in approximately the same number of generated nodes, for both RBFM and BFM. We then played 1,000 games between each pair of algorithms.

3.2.6. The results of Experiment 4

Table 1 summarizes the results of Experiment 4, which evaluates the performance of RBFM in a full game. The results show that for incremental random search trees, RBFM outperforms Alpha-Beta at all branching factors and all search depths. The results also show that RBFM outperforms BFM at large search depths. At small search depths BFM sometimes outperforms RBFM, but the shallow searches are atypical of real games. The BFM vs. Alpha-Beta results reported here are consistent with the results reported in [9].

Table 1
Experiment 4: Results over a full game. The first three columns show the search depths of RBFM and BFM for each Alpha-Beta depth. The column labeled RBFM-AB shows the winning percentages of RBFM when it played against Alpha-Beta. BFM-AB shows the winning percentages of BFM against Alpha-Beta, and RBFM-BFM is the winning percentages of RBFM against BFM. The last three columns show the total number of generated nodes over all games (in thousands of nodes). Since we used Alpha-Beta to determine the search depths, RBFM sometimes generated more nodes while BFM generated fewer nodes, or vice versa

Branching factor 3

$D_{\alpha\beta}$	D_{RBFM}	D_{BFM}	BFM-AB	RBFM-AB	RBFM-BFM	AB	RBFM	BFM
2	2	3	65	50	38	1018	963	1089
3	4	6	83	65	30	1876	1754	1841
4	6	9	84	70	37	2970	2729	2795
5	9	13	80	83	42	4824	4780	4480
6	12	18	83	87	50	7223	7599	7213
7	15	23	78	86	55	11069	11638	10671
8	17	29	73	84	53	16233	15168	15997
9	21	35	63	83	67	24281	24876	22721
10	24	43	62	84	69	35153	35041	33013

Branching factor 5

$D_{\alpha\beta}$	D_{RBFM}	D_{BFM}	BFM-AB	RBFM-AB	RBFM-BFM	AB	RBFM	BFM
2	2	2	37	59	69	1638	1625	1446
3	4	6	84	71	34	4155	3695	4062
4	7	10	84	84	47	7852	8594	8052
5	10	15	79	90	66	15949	16942	15851
6	13	20	70	92	74	27954	30117	26824
7	16	28	63	92	79	53241	50544	52657
8	20	35	49	92	88	91426	94849	86276
9	24	47	41	90	89	171763	173343	165763
10	27	59	32	84	88	290545	269537	279250

Branching factor 10

$D_{\alpha\beta}$	D_{RBFM}	D_{BFM}	BFM-AB	RBFM-AB	RBFM-BFM	AB	RBFM	BFM
2	2	2	33	80	89	3161	3768	3173
3	4	6	84	74	46	12445	10041	12643
4	7	10	71	86	68	29399	29356	29284
5	10	18	63	91	81	84353	70304	94283
6	14	24	40	95	94	178376	184822	181532
7	18	37	30	93	95	489118	462449	524663
8	22	48	17	89	97	1020199	1049817	1033780

Branching factor 20

$D_{\alpha\beta}$	D_{RBF}	D_{BFM}	BFM-AB	RBFM-AB	RBFM-BFM	AB	RBFM	BFM
2	2	2	34	91	96	6090	10293	7925
3	4	6	78	84	61	38115	32136	42578
4	7	11	51	85	74	110956	118381	133792
5	10	21	39	89	90	478581	362006	587442
6	14	28	14	87	96	1201360	1289543	1254802

4. Othello

Korf and Chickering assessed Best-First Minimax not only on random trees, but also on the game of Othello. They used the evaluation function of Bill, the program which won first place in the 1989 North American Computer Othello Championship [11]. Their experiments showed that for small search depths BFM outperforms Alpha-Beta. For greater depths the advantage of BFM over Alpha-Beta diminishes. Korf and Chickering estimated that BFM would eventually lose to Alpha-Beta.

We repeated their experiments using the RBFM algorithm. The following sections describe our modifications to the program, the methodology, and the experimental results.

4.1. Endgame play

Korf and Chickering observed that when BFM reaches terminal leaves, it requires much more time. BFM stops when the principal variation reaches the search depth. A terminal leaf can be on the principal variation only if the search algorithm has proved that the root wins or loses. Therefore, at endgame, BFM stops only when it finds a nonterminal leaf or determines the outcome of the game, which often takes a long time.

In order to address this problem, Korf and Chickering stopped BFM whenever a terminal leaf was encountered. This solution is too simplistic, and in fact insufficient. The problem might occur when the scores are large but not necessarily plus/minus infinity. Therefore, we used a different approach. During its execution, RBFM stores the average number of generated nodes in previous turns. RBFM aborts a computation if it takes more than five times that number. This stopping rule was applied to BFM as well.

4.2. Methodology

We used the same methodology described at [9]. We generated 244 boards, representing the possible board positions after 4 moves. Each of these boards was played twice, with either RBFM or Alpha-Beta moving first. Both Alpha-Beta and RBFM saved the relevant subtree from one move to the next. For each Alpha-Beta search depth, we found experimentally the search depth of RBFM which resulted in approximately the same number of generated nodes.

We performed the same experiments twice more, once playing BFM against Alpha-Beta and once playing BFM against RBFM. In the BFM-against-RBFM games, we used the same search depths as in the Alpha-Beta games.

4.3. Experimental results

The results of our experiments are shown in Fig. 9. Both BFM and RBFM outperform Alpha-Beta. However, RBFM wins over Alpha-Beta at greater percentages at depth 5 or more. The BFM results are similar to results reported in [9], and indicate that BFM probably loses to Alpha-Beta at search depths greater than 8. The advantage of RBFM over Alpha-Beta also seems to decline beyond search depth 5. We extrapolate that RBFM would match Alpha-Beta around search depth 13. In this depth the average search time is

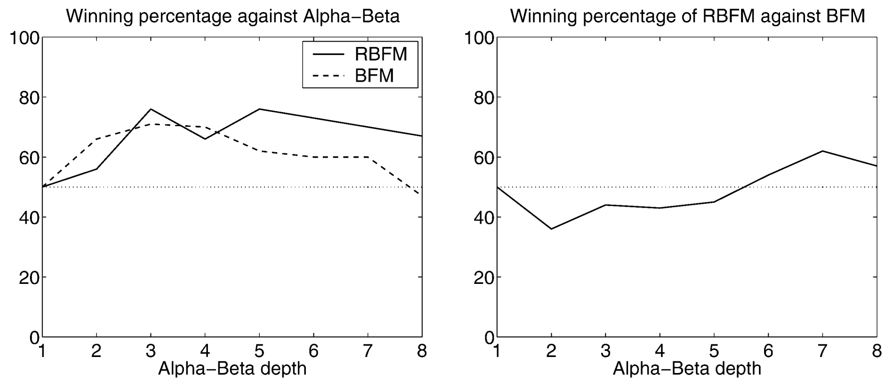

Fig. 9. Performance of RBFM in the game of Othello. The graph on the left shows the winning percentages of RBFM and BFM against Alpha-Beta. The graph on the right shows the winning percentages of RBFM against BFM. In both graphs the horizontal axis represent playing time per move, normalized using Alpha-Beta depth.

about 90 seconds, which is reasonable for a real game. Therefore, we expect that serial RBFM and Alpha-Beta would perform about equally well in practice. The experiments also show that RBFM beats BFM at depth 6 or more, but loses at shallower depths.

5. Chess

Chess is more complex than random tree and Othello. The evaluation function is typically not as accurate; it is usually based on many weighted heuristics for position evaluation. It can be totally wrong in some cases, declaring a winning position to be losing or vice versa.

We have implemented a Chess program using the RBFM algorithm. For the position evaluation, we used version 17.9 of the program *Crafty* by Robert Hyatt [6]. Crafty is probably the strongest non-commercial chess program and is freely distributed with source code. Our program evaluates positions by calling Crafty as a black-box. Crafty performs a shallow search and returns its value. We discuss the reasons for this strategy below.

In order to evaluate RBFM performance, we use both full games and a suite of test problems.

Our initial experiment used the Louguet Chess Test II (LCT-II), version 1.21 [12]. This is a test suite of 35 positions, divided into three main themes: positional, tactical and endgame. The test suite can be used to estimate the rating of a program, and is commonly used as a benchmarking tool.[3] In our experiments, we used only the positions that Crafty solves in 5−200 seconds. This filtering left us with 19 positions, which are shown in Fig. 13.

[3] For example, it was used to evaluate the Chess programs DarkThought (http://supertech.lcs.mit.edu/~heinz/dt/node98.html), Rebel-Tiger (http://www.rebel.nl/cpmisc.htm), and GromitChess (http://home.t-online.de/home/hobblefrank/gromit.htm).

We also evaluated RBFM by playing full games against Crafty. We used a suite of 20 middle-game positions to start the games. Each position starts two games, one in which RBFM plays white and another in which Crafty plays white. The positions were proposed by John Nunn, a British Grand Master [14]. He proposed them specifically to test computer-chess programs against each other independently of their opening books. (Simply playing from the initial position without opening books often leads to repetitions of games.)

The rest of this section describes our RBFM Chess program, how we estimated the distribution of the real score, and the results of our experiments.

5.1. Implementation: the static evaluator

In order to evaluate a leaf, we use Crafty as a black box that scores a position by performing a shallow search. The depth of this shallow search is denoted by SEARCH-DEPTH. In our implementation, SEARCH-DEPTH is set to 3. Increasing SEARCH-DEPTH results in more accurate, but slower evaluation. We measured the performance of the program with SEARCH-DEPTH set to 2, 3, and 4 and found that neither 2 nor 4 performed consistently better than 3. Berliner used a similar approach in his B* algorithm [2].

Another possible implementation is to use Crafty's evaluation function directly. This approach would not work well because we do a considerable amount of work when expanding a leaf. We need to set up the board position, go down the tree while generating random numbers, make the moves and update the board. In the parallel implementation, we also need to pack a message and send it to another processor. Therefore, if we simply compute the position evaluation, we will suffer significant overheads. Performing a shallow search balances the cost of the position evaluation with the associated overhead.

A second possible implementation is to replace the search algorithm in Crafty. This approach is technically too complex to implement, since Crafty is not modular enough to allow one to easily replace the search algorithm. In other words, Crafty's search algorithm (a variant of Alpha-Beta) is integrated with other aspects of the program and is, therefore, difficult to replace.

Our shallow-search approach, while perhaps not optimal, is technically feasible and benefits to some extent from Crafty's enhancements to the search algorithm (e.g., the transposition table).

The shallow search can be viewed as a tradeoff between node generation and evaluation accuracy. It allows us to reduce memory requirements, because we generate fewer (but more accurately evaluated) leaves. On the other hand, usually speed is preferred over accuracy in computer Chess. In recent years there is a tendency to make the evaluation function as fast as possible in order to search to a greater depth [17]. Some programs even reduce usage of chess knowledge in the evaluation function to make the evaluation faster. Our implementation does exactly the opposite: we have a fairly accurate but extremely slow evaluation (the rate is only 150 evaluations per second).

Eliminating the shallow searches would cause RBFM to use much more memory than with shallow searches. Algorithms like RBFM and B* that keep the search tree in memory consume a constant amount of memory per node evaluation, and hence their space requirement grows proportionally to the amount of work they perform. A fast evaluation

function, such as Crafty's, performs about 200,000 heuristic evaluations per second per processor. At around 60 seconds per move, the tree grows to around 12 million nodes. Even with a relatively compact representation for the tree, such an implementation requires hundreds of megabytes per processor. These memory requirements are indeed high, but not unacceptable. We chose to use shallow searches to reduce the memory consumption of the algorithm, but on machines with sufficient memory, running RBFM with a fast heuristic evaluator is feasible.

Note that our implementation performs redundant work. When a node is expanded, we perform a search of depth SEARCH-DEPTH for each of its children. This has the same effect as doing a search of SEARCH-DEPTH+1 from the node itself, except that we also know the exact values of the children. However, because of Alpha-Beta pruning, performing b searches to depth D is more expensive than performing a single search to depth $D + 1$, so our implementation does some redundant work.

5.2. Heuristics

We use four heuristics to speed up the search. The first two can be applied to any RBFM implementation, and the other are Chess-specific.

The first heuristic tries to overcome the redundancy in the search. When we need to expand a leaf, we actually search it to depth SEARCH-DEPTH+1, and compute its new value. The best child, who is returned by the search, gets this (negated) value, while all other sons are set to be *unknown*. We only know that their score is no better than the score of the best child. Three cases are now possible:

- If we never reach the node again, we have just profited from not evaluating its children exactly.
- If we reach the node in an EXTEND-DOWN call, we must proceed to its best child. This information is available, and the other children can remain unevaluated. However, if the score of the best child becomes greater than its initial value, then we have to compute the score of its siblings, since one of them may become the new best child.
- If we reach this node again in an EXTEND-UP call, we need to evaluate all its children, in order to determine the probabilities for each child. This process is time consuming compared to the initial SEARCH-DEPTH+1 search, so the redundant work we have done in the first search is relatively small.

The second heuristic slightly modifies the behavior of EXTEND-DOWN calls. When EXTEND-DOWN expands a leaf v, we search to depth SEARCH-DEPTH-1 and immediately search the best child to depth SEARCH-DEPTH+1. This results in a lower bound on the value of v. We use this heuristic because usually EXTEND-DOWN is called after one player made a mistake. For example, consider a move which puts a piece where it could be captured. When calculating the reply of the opponent, we should not waste time on a full analysis—we should just capture the piece and see what happens. To sum up, an EXTEND-DOWN call is executed by a very shallow search followed by an EXTEND-UP call, which returns a reliable score. This enables us to see one move deeper at about the same cost.

The third heuristic involves search extensions. When the variant (series of moves) leading to the searched position contains checks and captures, SEARCH-DEPTH is increased. Each check increases SEARCH-DEPTH by one ply, and each trade of equal-value pieces increases SEARCH-DEPTH by two plies.

The last heuristic that we use tests checking moves before other moves, regardless of their score.

5.3. Choosing the move to play

We use the following strategy to choose the move to play when the search ends.

RBFM always tries to lower the score of the current best move and raise the score of other moves. Therefore, it may happen that a poor move would temporarily become the best move from the root for a few iterations, until it is rejected again. Since we do not want to return such poor moves, we use a buffer which keeps the best moves of the last 100 iterations. The move that we actually return is the most common move in the buffer.

5.4. Parameter estimation

We evaluate the distribution of the real score by executing RBFM on different positions, as explained in Section 2.6. In our Chess implementation, we let RBFM play 30 chess games against itself. The length of each game was 60 moves (120 plies) so we got 3,600 positions. For each such position, we ran 200 RBFM node expansions and measured the distribution of the difference from the real score. The real score was obtained by using Crafty with SEARCH-DEPTH $= 10$. We assume that the variance of the distribution depends only on s (the number of extensions), and not on d (the number of children). We do so because our evaluation function is slow, so evaluating enough positions for each (s, d) pair would be too slow.

Fig. 10 shows the histogram of the difference from the real score. Since the distribution falls off quickly near its mean, we use an exponential distribution to represent the difference from the real score. Plotting the histogram on a log scale, however, reveals that large differences from the mean have significant probability, more than they would in an exponential distribution. We use an ad-hoc approach to address this difficulty.

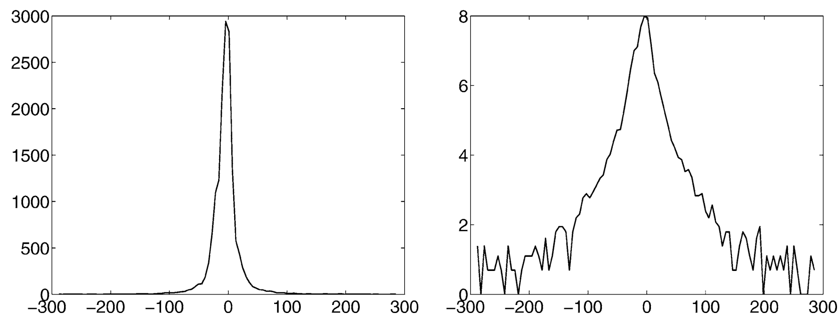

Fig. 10. The left histogram shows the difference from the real score after 200 node expansions. The value of a pawn is 100. The right graph shows the same data on a logarithmic scale.

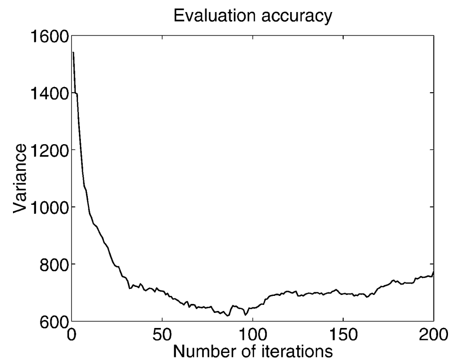

Fig. 11. The graph shows how the variance of the real score drops as the number of RBFM node expansions increases.

We estimate the functions f and g with an exponential distribution as explained in Section 2.6, but we damp the resulting distribution to bound from below the probability that each child is selected, no matter how poor its minimax score. The damping is done in CHOOSE-RANDOM-CHILD. When this function is called, it usually uses the exponential distributions. However, with a probability of 25%, it chooses a totally random child. We chose the amount of damping (25%) based on a limited amount of experimentation. This hybrid of exponential and uniform distributions allows some level of tactical play. We damp the distribution in this way only for Chess. We acknowledge that this is an ad-hoc approach and we expect that using heavy-tailed family of distributions would eliminate the need to damp the distribution.

The graph in Fig. 11 shows that performing more node expansions improves the accuracy of the score. Interestingly, after 100 node expansions the variance starts to raise. A plausible explanation is that in Chess, many moves are forced (recaptures, checks evasions etc.). In such cases, the forced move is extended only using EXTEND-UP because it remains the best child. Therefore, the more the position is searched, the worse it seems.

5.5. Methodology

We compare the speed of our Chess implementation to Crafty, which uses Alpha-Beta search (more precisely, the negascout variant). This comparison underestimates RBFM, because Crafty contains many enhancements to the search algorithm. These include, for example, transposition tables, killer-moves, null-move heuristics, iterative deepening, extensions and so on (see [13] for a review of techniques used in modern Chess programs). These enhancements have been shown to be effective in Alpha-Beta-based Chess programs. Determining whether they are as effective when used in conjunction with RBFM is beyond the scope of this paper.

5.5.1. Methodology for test positions

The time we report includes the time needed for buffering. This somewhat underestimates RBFM, because after it finds the correct move it still needs to fill the buffer in order to report it. However, the verification prevents us from stopping prematurely.

Since RBFM is a randomized algorithm, the solution time varies between executions, even on the same input. The time we report is the average solution time over 20 experiments.

The LCT-II experiments were done on an Origin-2000 machine. In Section 6.3 we shall see that using multiple processors reduces the solution time.

5.5.2. Methodology for full games

In the full-game experiments we played RBFM against Crafty. We ran three tournaments of 40 games (20 starting positions, each starting two games). We always gave Crafty 15 seconds per move. We gave RBFM 90 seconds per move in one tournament, 105 seconds in the second tournament, and 120 in the third. These 1 : 6, 1 : 7, and 1 : 8 time ratios were determined based on the results of the LCT-II experiments, which indicated that RBFM is 7–15 times slower than Crafty. Neither program was computing while the other was making a move.

The full-game experiments were performed on a dual-processor Pentium-III computer. We only used one processor.

5.6. Experimental results

The LCT-II solution times are shown in Table 2. Our implementation is usually 7–15 times slower than Crafty, but in some positions RBFM is 30 times slower.

The full-game results, shown in Table 3, indicate that RBFM is about 7 times slower than Crafty.

Table 2
Solution times of RBFM and Crafty on the subset of the LCT-II test suite that we use

Test name	RBFM time	Crafty time
Pos3	24.7	18.14
Pos4	143.9	28.33
Pos5	939.0	80
Pos9	23.5	7.67
Pos11	1120.9	69
Pos12	83.9	9.41
Cmb1	77.7	16.52
Cmb4	241.2	13.44
Cmb5	37.1	5.14
Cmb6	142.3	7
Cmb7	354.9	157
Cmb8	278.8	6.79
Cmb9	591.1	57.86
Cmb10	2736.8	64
Fin3	6.1	16.13
Fin4	165.4	155
Fin5	140.3	24.51
Fin6	207.4	11.25
Fin7	2834.4	102

Table 3
Results of the three tournaments of full games between Crafty and RBFM. The time ratios are all in favor of RBFM

Time ratio	RBFM wins	Draws	Crafty wins	Total RBFM points
6	6	12	22	12.0
7	16	12	12	22.0
8	15	16	9	23.0

We attribute RBFM's poor performance on the test positions to two main factors:

- The solutions to most of the test positions involve a material sacrifice, in order to gain an attack. When RBFM does not see a compensation right away, it examines the move with a small probability. Even when performing a shallow search instead of a single evaluation, the error of the evaluation function in such cases can be quite large.
- Crafty implements many enhancements to the search algorithm, as explained in 5.5. For example, some problems that Crafty solved much faster (*Cmb6, Cmb8, Cmb10*) contain mate threats. Crafty uses the null-move heuristic to detect the threats, and extends the search in such cases.

6. Parallel RBFM

All parallel game-search algorithms use speculation to achieve parallelism. Let us first explain what is speculation. The basic action of a game-search algorithm is to heuristically evaluate a node and to update its data structures accordingly. The algorithm uses the information gathered from previous node evaluations, which is stored in the data structures, to select the next node to evaluate. Even when such an algorithm runs on a parallel computer, the amount of work is minimized if the algorithm waits for the completion of one node evaluation before it decides on the next node to evaluate, since this provides the algorithm with the maximal amount of information for making the decision. But waiting for one node evaluation before starting the next completely serializes the algorithm. Therefore, a parallel algorithm must decide to evaluate a node before all the previous evaluations terminate, or to decide on several node evaluations in parallel. A node evaluation that starts before all the previous ones terminate or that starts in parallel with other node evaluations is said to be *speculative*.

A speculative search that a parallel program performs might also be performed at some point by the serial algorithm, or it might not. When the speculative search is also performed by the serial algorithm, the parallel algorithm performs the same work in parallel using multiple processors. Otherwise, the speculative search is extra work that serves no useful purpose.

The amount of extra work depends on two factors. The first one is the quality of speculation (the likelihood that the speculative search is also performed by the serial algorithm). The second one is the granularity of the speculation. We say that a decision to perform a speculative search that commits the algorithm to performing a large search is *coarse grained*, whereas a speculation that commits the algorithm to a small search is *fine*

grained. Some algorithms can abort speculative searches if it turns out that they are not useful. In such algorithms, granularity is not an issue.

Coarse-grained algorithms either perform more extra work than fine-grained ones, or they do not exploit processors effectively. The problem occurs when a coarse-grained algorithm decides to perform a large speculative search and discovers that it is not useful before the search terminates, but cannot abort it. Such an algorithm either performs more extra speculative work than necessary to keep the processors busy, or it speculates conservatively in order not to perform too much extra work. Conservative speculation may lead to idle processors.

The same problem may happen in a weaker form. An algorithm may discover in the middle of a speculative search that the search is unlikely to be useful and that another speculation is more likely to be useful but does not abort the first search. In this case resources are spent on a search that is not strictly useless, but that is less useful than another search.

We parallelize RBFM by running P (the number of processors) RBFM node expansions concurrently and choosing P leaves. We expand all selected leaves and backup the scores. This can be viewed as 1 non-speculative expansion and $P - 1$ speculative expansions. In the next node expansion, RBFM reassesses the search tree and uses the most up-to-date information to choose new speculations. Speculation in RBFM is, therefore, fine grained.

We now turn our attention to the quality of speculation. We claim that the quality of speculations in RBFM is high. If the static evaluator is reliable, expanding a leaf does not change its value much. Therefore, the probabilities of choosing other leaves do not change significantly. In addition, backing up scores continues only while the node is the best child of its parent. Therefore, most changes are expected to be near the leaves. Such changes do not affect the probabilities of choosing leaves in remote subtrees.

Our implementation is asynchronous. Whenever a processor reaches a leaf, it expands it, backups the score, and starts over, regardless of what the other $P - 1$ processors are doing. This allows the processors to remain busy, rather than wait at global synchronization points. There are, however, implicit synchronization points: when two processors need to update the same node, they use locks to serialize the updates. When the search tree is large enough, we do not expect much contention for locks.

We have also implemented RBFM using a master-slave technique, in which only a single processor updates the search tree. This serialization does not hurt performance in practice, as long as leaf expansion is significantly more expensive than selecting leaves and backing up scores.

We compare RBFM to Jamboree [7,10] search, a parallel version of Alpha-Beta (more precisely, of the scout variant). Jamboree was used as the search algorithm in the chess program *Socrates [7], which won second place in the 1995 computer chess championship. Jamboree serializes some of the processing of a node: it first evaluates one move, and it then tests, in parallel, that the other alternatives are worse. Moves that fail the test (i.e., are better than the first move) are evaluated in a sequence. The speculations in Jamboree involve evaluating large subtrees, and are therefore coarse grained. Jamboree does not reduce the magnitude of a search whenever it can; and it does not allocate processors to speculative searches based on the likelihood that they are useful. To avoid large amounts of extra work, Jamboree often prefers to wait for the termination of previous searches before

embarking on a speculative search. We show that this prevents Jamboree from utilizing all the available processors. Empirical measurements show that RBFM benefits from a large number of processors more than Jamboree and achieves a greater speedup.

Note that several parallel game-search algorithms have been proposed in the literature, besides Jamboree. For a thorough survey, see [3].

The rest of this section describes our implementations of parallel RBFM, the results of experiments with the shared-memory implementation on random trees, and the results of experiments with the master-slave implementation on Chess.

6.1. Parallel implementations

We have implemented parallel RBFM using both a shared-memory model and a master-slave model. In a shared memory model, all processes access a shared search tree. Each process executes RBFM node expansions and updates the tree, regardless of what other processes are doing. Processes lock nodes when they traverse the tree. The combinatorial structure of the tree does not change, only grows, so processes only need to lock one node at a time. Hence, no deadlock can occur even if processes lock nodes both on the way down to a leaf and on the way up backing up scores. (In practice we do not lock on the way down; we may read inconsistent structures but the only effect on the algorithm would be poor speculation.) This shared memory algorithm is simple. The only changes to the serial code involve locking nodes. The main problem with the shared-memory algorithm is that on machines whose memory is physically distributed, like the Origin-2000, a lot of time is spent communicating tree updates between processor caches.

In a master-slave model, one processor keeps the search tree in its local memory. This processor selects leaves and backs up scores. Whenever it selects a leaf, it sends a message to an idle slave processor. The slave computes the possible moves from the leaf, and their heuristic evaluations. It sends this information back to the master, which updates the tree. While the master waits to the reply of the slave, it assigns work to other idle slaves. In this model, leaf selections and score updates are completely serialized, but tree updates involve no communication. The serialization limits the number of processors that can be effectively utilized. The maximum number of processors that this implementation can use depends on the cost of leaf evaluation relative to the cost of tree updates and leaf selections.

6.2. Parallel random game trees

We have implemented both parallel RBFM and Jamboree search in the Cilk language [5,19]. Cilk adds to the C language shared-memory parallel-programming constructs. Cilk's run-time system allows the user to estimate the parallelism of an algorithm, even when the algorithm is executed on a small number of processors (or even one processor). Both RBFM and Jamboree speed up as the number of processors grows. However, at some point the speedup stagnates. We show that the reasons for the stagnation are different: Jamboree search is limited by the length of its critical path, while in parallel RBFM tree-update operations require expensive communication. Under the conditions of our experiment, RBFM was able to utilize more processors before stagnating, and therefore achieved a greater speedup.

6.2.1. Bounds on parallelism

In many parallel algorithms some part of the computation must be completed before another part can proceed. This creates chains of events that must be computed sequentially. The longest chain is called the *critical path*. The execution time is bounded from below by the time required to perform the critical path, which is denoted by T_∞. Another bound on the execution time of the program is the amount of work. If on a single processor the program runs in time W, on P processors it cannot run faster than W/P. The execution time is bounded from below by T_∞ and by W/P. When T_∞ becomes the dominant term, adding more processors cannot speed the execution significantly. Cilk's randomized scheduler guarantees that the program executes in $\mathrm{O}(W/P + T_\infty)$ expected time. Therefore a good estimate of a program's parallelism is W/T_∞. Cilk measures both W and T_∞, so it can estimate the program's parallelism.

6.2.2. Methodology

In order to measure the parallelism of RBFM and Jamboree search, we conducted the following experiment. We generated a fixed set of 100 incremental random search trees with branching factor $b = 10$. We searched these trees to depth 10 using Jamboree search, and to depth 26 using RBFM. This resulted in approximately the same number of generated nodes. We compare the time it takes Jamboree and RBFM to solve the test set, using different numbers of processors.

By measuring the time it takes to solve 100 trees, we get robust results, since the parallelism of an algorithm might be influenced by the quality of move ordering of specific trees. In addition, RBFM is a randomized algorithm, and its running time changes even for a fixed tree.

In Section 3.1 we assessed performance in terms of generated nodes. When measuring parallelism, there are other important parameters, such as communications between processors, cache hierarchy, scheduling, and so on. From now on, we discuss the running time of the algorithms. Static evaluation in random trees costs a single machine operation, which is unrealistic. In order to model real evaluation functions, we perform an empty loop with either 50,000 or 200,000 iterations whenever a leaf is expanded. This simulates the cost of an evaluation function, or a slow evaluation function. We still focus on the effectiveness of the search, but for a large number of processors the other performance determinants become significant.

While the use of an empty loop may seem artificial, it is indeed necessary in this experiment. Random trees are a synthetic model of real games. As such, they allow us to assess the behavior of a search algorithm independently of a specific evaluation function. Random trees also allow multiple researchers to repeat each other's experiments easily and without access to complex and often proprietary game-playing programs. But since random trees do not model well the *computation cost* of the evaluation function, they cannot be used directly to compare the running times of search algorithms. The empty loop is used to more accurately model real evaluation functions. We use two loop counts in order to assess the effect of the cost of the evaluation function on the performance of the algorithms.

6.2.3. RBFM experimental results

On a single processor, the running time of RBFM is 1552 seconds. The length of the critical path is 0.35 seconds. The critical path is very short, because there are no explicit dependencies between threads. Implicitly, the locks create such dependencies.

Fig. 12 shows the speedups for multiple processors. For the normal evaluation function, the algorithm does not speed up beyond 30 processors. We decompose the execution time into time spent in the evaluation function (in our implementation, the empty loop) and the rest of the time, which we call *search overhead*. This overhead includes tree operations (generating random numbers, expanding leaves, backing up scores, etc.) and Cilk overhead (spawning threads, thread scheduling, etc.). We estimate that a large part of the search overhead is caused by *cache coherency contention*.

The memory system of the Origin-2000 uses caches and a directory-based coherence protocol (memory is sequentially consistent). Each processor stores a part of the address space used by the program in its main memory. Each processor also stores recently-used data from all of the program's address space in a private cache. Directory hardware stores for each main-memory block (whose size is that of a cache line) the identity of processors that have a copy of the block in their cache. If one of the processors changes a value in its cache, it updates the processor that owns the block, and this processor invalidates the caches of the other processors. The invalidation is done by sending messages over a network.

In our implementation, all processes access the search tree, so large parts of it, especially the nodes near the root, are likely to be stored in the caches of all processors. When the number of processors increases, the number of cache misses increases, and more messages are required for tree updates. We have verified this using the *perfex* tool, which counts cache misses using hardware counters. We have not been able to reduce the number of cache misses by altering the layout of the tree in the Origin's distributed memory system.

6.2.4. Jamboree experimental results

Jamboree search is sensitive to the quality of move ordering both in the amount of work and the length of the critical path [7]. Therefore, in our implementation we use static move ordering, in spite of the cost of the evaluation function. When the depth of the searched subtree becomes smaller than 3, we switch to Alpha-Beta search rather than continue the Jamboree recursion. We do so because Alpha-Beta is a better searcher for shallow depths [16]. In addition, invoking a Cilk thread incurs a considerable overhead. The Cilk manual and papers recommend calling a serial code with enough work in the bottom of the recursion.

When Jamboree runs on a single process, Cilk reports that the work is 1,952 seconds, and the length of the critical path is 132 seconds. Therefore, Cilk estimates a parallelism of 14.8.

Fig. 12 shows the speedups when using multiple processors. The results show that Cilk's estimate is indeed accurate. Using more than 15 processors does not increase the speedup. If fact, using more than 20 slows down the program. Jamboree search speculates conservatively, so it does some computations sequentially. This results in a long critical path and relatively little parallelism.

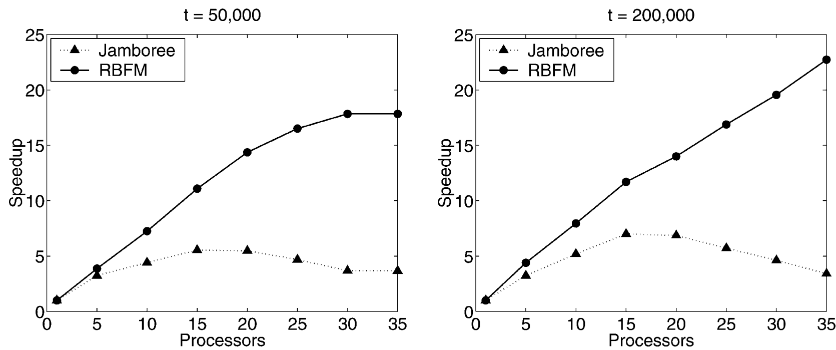

Fig. 12. Speedups of RBFM and Jamboree. t is the size of the empty loop in the evaluation function.

6.2.5. A comparison of RBFM and Jamboree

Fig. 12 compares the speedups of RBFM and Jamboree. RBFM was able to utilize more processors and achieved a greater speedup than Jamboree.

Both RBFM and Jamboree stagnate, but for different reasons. RBFM stagnates because of the search overhead. When the evaluation function is slow, RBFM can utilize more processors before stagnating. Jamboree is bounded by its critical path and a slower evaluation function does not improve its parallelism.

We conclude that parallel RBFM using a shared memory is applicable for a moderate number of processors, or for a large number if the evaluation function is slow.

6.3. The performance of parallel RBFM on Chess

We implemented a parallel version of our Chess program, using the master-slave model. We performed our experiments on an Origin-2000 machine. The program uses MPI [4,18] to communicate between processes.

Table 4 shows that RBFM often achieves a linear speedup, even for 64 processors.

In some cases the speedup was greater than linear. Sometimes the speculative search in a parallel computation finds the solution that enables the program to stop before completing all the work of the serial computation. Specifically, in some of the test positions the right solution initially appears to be a bad move, and the serial program considers it with a small probability. When the search tree is small, the parallel program must examine bad moves to generate work for all the processors. Therefore, it discovers the solution earlier.

Small superlinear speedups could also have been created by inaccuracies in the time measurements. (The reported running times are averages of 20 runs, measured to within 1 sec each.)

7. Conclusions, discussion, and related work

We have presented RBFM, a selective search algorithm which is a randomized version of Best-First Minimax search [9]. We have shown that RBFM is efficient and highly parallel.

Table 4
Running times of our program when it was executed on 1, 32 and 64 processors, and the relative speedups. The time is an average of 20 runs. The first column shows the solution times of Crafty (using a single processor)

Test name	Crafty time	$n = 1$	$n = 32$	Speedup	$n = 64$	Speedup
Pos3	18.14	24.7	4.38	5.64	4.41	5.60
Pos4	28.33	143.9	4.11	35.01	4.03	35.71
Pos5	80	939.0	30.57	30.72	21.98	42.72
Pos9	7.67	23.5	1.31	17.94	1.15	20.43
Pos11	69	1120.9	39.08	28.68	31.86	35.18
Pos12	9.41	83.9	2.52	33.29	3.13	26.81
Cmb1	16.52	77.7	1.18	65.85	1.09	71.28
Cmb4	13.44	241.2	3.91	61.69	3.33	72.43
Cmb5	5.14	37.1	2.07	17.92	1.54	24.09
Cmb6	7	142.3	4.21	33.80	2.20	64.68
Cmb7	157	354.9	3.07	115.60	1.46	243.08
Cmb8	6.79	278.8	12.52	22.27	3.28	85.00
Cmb9	57.86	591.1	18.19	32.50	6.84	86.42
Cmb10	64	2736.8	89.84	30.46	38.00	72.02
Fin3	16.13	6.1	0.81	7.53	0.45	13.55
Fin4	155	165.4	7.19	23.00	3.70	44.70
Fin5	24.51	140.3	6.22	22.56	3.31	42.39
Fin6	11.25	207.4	6.58	31.52	3.06	67.78
Fin7	102	2834.4	123.25	23.00	53.90	52.59

For incremental random game trees, RBFM makes better decisions than Alpha-Beta search, and wins 90% of its game against it. We attribute RBFM's success to the accuracy of the evaluation function. In random trees the evaluation function is usually accurate and easy to model. This allows RBFM to prune moves that seem bad without taking large risks.

In Chess, on the other hand, the evaluation function can be inaccurate. This may cause RBFM to prune the right move (or examine it only with a small probability when damping is used). Consequently, RBFM is slower than Alpha-Beta. More specifically, our RBFM program is about 7 times slower than Crafty on full games and 7–15 times slower on tactical test positions. Bear in mind that our experimental program was compared to Crafty, a state-of-the-art Chess program. It is difficult to draw practical conclusions from such a comparison.

In Othello, RBFM outperformed Alpha-Beta for the search depths of our experiments (up to 8). Othello is a simpler game than Chess, so evaluation functions are typically more accurate. RBFM exploits the accuracy of the evaluation function.

As a parallel search algorithm, RBFM performs fine-grained and high-quality speculations. This results in a high level of parallelism with little extra work. For random trees, RBFM achieved a better speedup than Jamboree. On Chess test positions, our program often achieves a linear speedup even on 64 processors. On the Chess test positions, our program is usually more than 5 times faster than the sequential Crafty.

The idea of using distributions instead of a single value to represent the state of a node is certainly not new; randomizing the search is. Berliner, in his B* algorithm [1] was the first to use distributions. Berliner suggested a model in which each node has a lower bound and

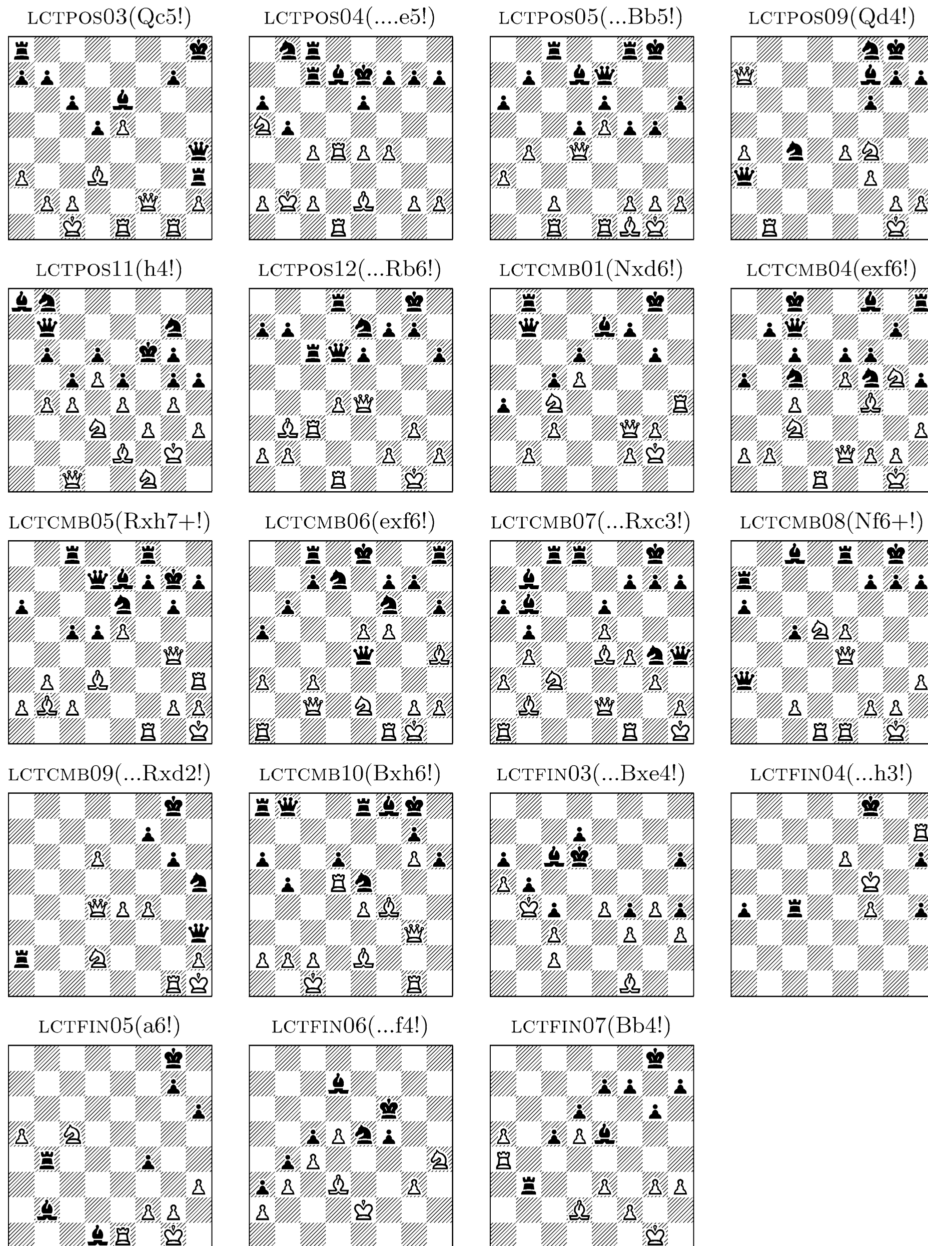

Fig. 13. Chess diagrams of our test positions. The title is the name of the position in the LCT-II test. The move in brackets is the correct solution.

an upper bound on the value of the real score. Palay extended this idea [15]. He showed how to back distributions up a search tree. We used a simpler model that we have shown to

be effective. Whether Palay's method can improve our results is beyond the scope of this paper. However, neither Berliner nor Palay suggested a randomized search algorithm.

Many parallel game-search algorithms have been proposed in the literature. In his PhD thesis, which proposes one of them, Brockington provides an excellent survey of the area [3]. From our point of view, the most important aspect of his survey is that most of the algorithms are synchronous and most attempt to parallelize Alpha-Beta. RBFM is neither.

Acknowledgements

Thanks to Richard Korf for providing us with the Othello-experiments source code. Thanks to Robert Hyatt for making Crafty publicly available and for allowing us to use it in this research. Thanks to Kai-Fu Lee for allowing us to use the board evaluation function of his Othello-playing program Bill. Thanks to the two anonymous referees for their criticism and for their helpful suggestions.

References

[1] H. Berliner, The B* tree search algorithm: A best-first proof procedure, Artificial Intelligence 12 (1979) 23–40.
[2] H.J. Berliner, C. McConnell, B* probability based search, Artificial Intelligence 86 (1996) 97–156.
[3] M. Brockington, Asynchronous parallel game-tree search, PhD Thesis, Department of Computing Science, University of Alberta, Edmonton, AB, 1994.
[4] Message Passing Interface Forum, MPI: A message-passing interface standard, Internat. J. Supercomput. Appl. 8 (3–4) (1994).
[5] M. Frigo, C.E. Leiserson, K.H. Randall, The implementation of the Cilk-5 multithreaded language, ACM SIGPLAN Notices 33 (5) (1998) 212–223.
[6] R. Hyatt, Crafty. Computer program, available online from ftp://ftp.cis.uab.edu/pub/hyatt. Some additional documentation available at http://www.cis.uab.edu/info/faculty/hyatt/hyatt.html, 1999.
[7] C.F. Joerg, B.C. Kuszmaul, The *Socrates massively parallel chess program, in: S.N. Bhatt (Ed.), Parallel Algorithms: Proceedings of The Third DIMACS Implementation Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 30, American Mathematical Society, Providence, RI, 1996, pp. 117–140.
[8] D. Knuth, R. Moore, An analysis of alpha-beta pruning, Artificial Intelligence 6 (1975) 293–326.
[9] R.E. Korf, D.M. Chickering, Best-first minimax search, Artificial Intelligence 84 (1996) 299–337.
[10] B.C. Kuszmaul, The StarTech massively-parallel chess program, ICCA J. 18 (1) (1995).
[11] K. Lee, S. Mahajan, The development of a world-class Othello program, Artificial Intelligence 43 (1990) 21–36.
[12] F. Louguet, The Louguet Chess Test II (LCT II) FAQ. Available online at http://www.icdchess.com/wccr/testing/LCTII/lct2faq.html, April 1996.
[13] T. Marsland, Anatomy of a chess program. In the brochure of the 8th World Computer Chess Championship, May 1995. Available online at http://www.dcs.qmw.ac.uk/~icca/anatomy.htm.
[14] J. Nunn, Nunn-Test II. A collection of 20 middle-game positions, available online with documentation from http://www.computerschach.de/test/nunn2.htm. No date.
[15] A.J. Palay, Searching with Probabilities, Pitman Publishing, 1985.
[16] J. Pearl, Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley, Reading, MA, 1984.
[17] E. Schroder, Chess in 2010. Available online at http://www.rebel.nl/ches2010.htm.

[18] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, MPI: The Complete Reference, Vol. 1: The MPI Core, 2nd edition, MIT Press, Cambridge, MA, 1998.
[19] Supercomputing Technologies Group, MIT Laboratory for Computer Science, Cambridge, MA. Cilk-5.3 Reference Manual, June 2000. Available online at http://supertech.lcs.mit.edu/cilk.