Parallel Randomized Best-First Search

Yaron Shoham and Sivan Toledo

School of Computer Science, Tel-Aviv Univsity
http://www.tau.ac.il/~stoledo, http://www.tau.ac.il/~ysh

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 a factor of 18 (using 35 processors). In comparison, Jamboree [Kuszmaul '95], speeds up by only a factor of 6. We have also evaluated the algorithm in a Chess-playing program using the board-evaluation code from Crafty, an existing Alpha-Beta-based program. On a single processor our program is slower than Crafty; with multiple processors it outperforms it.

1 Introduction

We present a new game-search algorithm that we call Randomized Best-First Search (RBF). It is the first randomized game-search algorithm. It is based on Korf and Chickering's deterministic Best-First Search [5]. Randomization fixes a defect in Best-First and makes it highly parallel.

RBF 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 Search (BF) 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, BF always chooses the same child until the score of the child drops below the score of another child. The defect in BF is that it never attempts to raise the score of non-best children. Indeed, there are search trees in which BF terminates without determining the best move from the root.

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

RBF 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. BF 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.

The rest of the paper is organized as follows. Section 2 describes the RBF algorithm. Section 3 summarizes our experiments that assess the efficiency of RBF. Section 4 shows that RBF can effectively utilize many processors. The experiments use two models, incremental random game trees [2] and Chess. On random trees, RBF speeds up more than Jamboree search (a parallel version of Alpha-Beta). For Chess, we show that RBF speeds up almost linearly with as many as 64 processors. The full paper describes more experiments, which investigate the behavior of RBF in more detail; they do not invalidate any of our conclusions. Section 5 presents the summary of our work.

2 The RBF Algorithm

RBF maintains the search tree in memory. Each node has a score, which represents the value of the game for the side to move. For the leaves of the tree, RBF 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 maximum of the negated scores of its children. In addition to the score, each node stores a distribution function. Section 2.3 describes how this distribution is derived.

RBF is an iterative algorithm. Each iteration starts at the root, and begins 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 2.2 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 below in Section 2.3. Figure 1 shows a simple example.

RBF runs until some stopping criterion is met. For example, the search can be stopped if the tree has reached a certain size or depth, if one child of the root emerges clearly as the best move, and so on.

2.1 A Pruning Rule

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.

The algorithm is either exploring the subtree rooted at the lowest scoring child c_l of the root or it is exploring another subtree rooted at c_o. 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

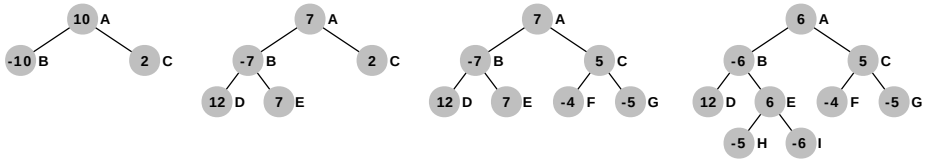

Fig. 1. An example that shows how RBF works. In iteration 1, the root is expanded, and its children are assigned their heuristic values. The score of the root is updated according to the negamax rule. In iteration 2 the algorithm needs to make its random choice between B or C. Node B has a lower score, and is therefore more likely to be chosen. Iteration 3 shows that RBF can check children other than the lowest scoring child. In iteration 4 RBF chooses the best child twice: first it chooses B and then it chooses to expand E.

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.

2.2 Child Selection

The motivation of Best-First (and RBF) is to examine the best move when analyzing a variation. In a minimax 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 Best-First and RBF use partial information about the game to guess which move is the best.

Best-First simply uses the current negamax scores to guess which move is best. RBF 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 RBF to make excellent decisions. But RBF does not know the real score; it tries to estimate it probabilistically.

RBF treats the real score of a node as a random variable. RBF uses information about the current tree to estimate the distribution of this random variable (see Sect. 2.3). When RBF 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.

[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.

2.3 Estimating Node Distributions

The details of estimating the distribution of the real score are described in the full paper. In short, the distribution depends on the negamax score of the node, the size of the subtree rooted at the node (a large subtree implies a smaller variance) and the number of children (a node with many children is expected to have a high score).

We estimate distribution parameters by generating many game positions. For each position, we use Alpha-Beta to depth 10 to compute the real score. We estimate the distribution using this training data.

3 Assessing the Efficiency of the Algorithm

We show that RBF is efficient by comparing it to Alpha-Beta search [4]. Alpha-Beta is widely considered to be one of the most efficient search algorithms and is used by nearly all game-playing programs. In the full paper we show that on a class of artificial games, called *incremental random game trees*, RBF outperforms Alpha-Beta. We also compare RBF to Alpha-Beta on Chess. This section only briefly overviews our results, which are fully presented in the full paper.

Incremental Random Game Trees. An *incremental random search tree* is a rooted tree with random numbers assigned to its edges. Random trees allow researchers to evaluate search algorithms while avoiding the complications of games like Chess. Random trees were used as test models by Berliner, Karp, Nau, Newborn, Pearl, and others (see [5] for full citations). 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 accurate the evaluation is. A tree can have a *uniform branching factor* if all nodes have exactly b children. For *random branching* trees 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).

Random branching trees are more difficult than uniform branching trees, because they have a less accurate evaluation function [5]. Hence, from now on we only discuss random-branching trees.

Random Trees Results. We compared the probability of finding the best move for RBF and Alpha-Beta, when both algorithms have the same computational resources. We generated random game trees of fixed depth, with different branching factors. The heuristic values of the leaves were treated as exact values. For each random tree, we ran Alpha-Beta at different search depths. For each search depth, we recorded whether it found the correct move, and the number of generated nodes. We then ran RBF until the same number of nodes was generated, at which point we stopped the algorithm and checked whether it had found the correct move.

The results showed that for all branching factors and all Alpha-Beta search depths, RBF makes better decisions than Alpha-Beta using the same number of generated

nodes. We confirmed this result by actually playing games of several turns between RBF and Alpha-Beta. The exact setup of the games is described in the full paper. RBF won 90% of the games.

Chess Results. We have implemented a Chess program using the RBF algorithm, in order to evaluate RBF in a more realistic setting. For the position evaluation, we used the program *Crafty* by Robert Hyatt[3] . Crafty is probably the strongest non-commercial chess program and is freely distributed with source code. In order to evaluate a leaf, we use Crafty to perform a shallow search of depth 3. The full paper explains this design decision. This evaluation function is fairly accurate, but slow (we evaluate about 150 positions per second).

In order to evaluate RBF performance, we use the Louguet Chess Test II (LCT-II), version 1.21 [7]. This is a test suite of 35 positions, which is commonly used as a benchmark tool[4]. In our experiments, we used only the positions that Crafty solves in $5 - 200$ seconds. This filtering left us with 19 positions.

We compared the speed of our Chess implementation to Crafty, which uses Alpha-Beta search (more precisely, the negascout variant). This comparison underestimates RBF, 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 [8] for a review of techniques used in modern Chess programs).

The results are shown in the first columns of Table 1. Our implementation was usually $7 - 15$ times slower than Crafty. The analysis of these results is complex and will appear in the full paper.

4 Parallel RBF

All parallel game-search algorithms use speculation to achieve parallelism. A speculative search that a parallel program performs might also be performed at some point by the sequential algorithm, or it might not. In the later case, it 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 sequential algorithm). The second one is the granularity of the speculation. We say that a speculation 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*.

The problem with a coarse-grained algorithm occurs when it 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.

[3] http://www.cis.uab.edu/info/faculty/hyatt/hyatt.html

[4] For example, it was used to evaluate the Chess programs DarkThought, Rebel-Tiger, and GromitChess.

We parallelize RBF by running P (the number of processors) RBF iterations concurrently and choosing P leaves (perhaps fewer than P if we allow multiple processors to choose the same leaf or if there are fewer than P leaves). We then expand all the leaves and backup the scores. This can be viewed as 1 non-speculative expansion and $P-1$ speculative expansions.

Speculation in RBF is, therefore, fine grained.

We now turn our attention to the quality of speculation. We claim that the quality of speculations in RBF 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, the backing up of 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.

We compare RBF to Jamboree [6],[3] 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 [3], which won second place in the 1995 computer chess championship. Jamboree is a synchronized search algorithm: 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 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 RBF benefits from a large number of processors more than Jamboree and achieves a better speedup.

4.1 Parallel Implementations

We have implemented parallel RBF using both a shared-memory model and a master-slave model. In a shared memory model, all processes access a share search tree. Each process executes RBF iterations and updates the tree, regardless of what other processes are doing. This asynchronous implementation allows the processors to remain busy, rather than wait at global synchronization points. Processes lock nodes when they traverse the tree, to serialize tree updates. The full paper proves that the program is deadlock-free.

The main problem with the shared-memory algorithm is that 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 scores. 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.

4.2 Parallel Random Game Trees

We have implemented both parallel RBF and Jamboree search in the Cilk language [10]. 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 RBF 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 speculates conservatively and processors become idle, while in parallel RBF tree-update operations require expensive communication. Under the conditions of our experiment, RBF was able to utilize more processors before stagnating, and therefore achieved a better speedup.

Methodology. In order to measure the parallelism of RBF and Jamboree search, we conduct the following experiment. We generate a fixed set of 100 incremental random search trees with branching factor $b = 10$. We search these trees to depth 10 using Jamboree search, and to depth 26 using RBF. This results in approximately the same number of generated nodes.

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 $50,000$ iterations whenever a leaf is expanded. We compare the time it takes Jamboree and RBF to solve the test set, using different numbers of processors. The experiments were performed on an Origin-2000 shared-memory machine with 112 processors.

RBF Experimental Results. Figure 2 shows the speedups for multiple processors. Beyond 30 processors, the algorithm does not speed up. 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.).

To estimate the search overhead, we removed the delay in the evaluation function. A large part of the overhead is caused by tree operations, specifically tree updates. The tree is updated when RBF updates the number of extensions in nodes it visits, and in the backup of scores.

We conducted an experiment in which we replaced each write to memory with two consecutive writes. The results showed that for large numbers of processors, the program was 30% slower! Writes are expensive because of the memory system of the Origin-2000.

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 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 updates the caches of the other processors. The update is done by sending messages over a network.

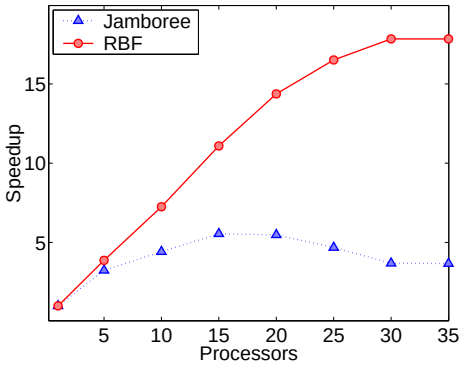

Fig. 2. Speedups of RBF and Jamboree

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. For large numbers of processors, each update to the tree involves sending many messages, and communication becomes a bottleneck.

When the evaluation function is slow, the overhead in tree operations is less significant, and RBF can speed up more (see Sect. 4.3).

Jamboree Experimental Results. Jamboree search is sensitive to the quality of move ordering both in the amount of work and the level of parallelism [3]. 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 instead of continuing the Jamboree recursion. We do so because Alpha-Beta is a better searcher for shallow depths [9]. In addition, invoking a Cilk thread incurs a considerable overhead. The Cilk manual and papers recommend calling a sequential code with enough work in the bottom of the recursion.

When Jamboree runs on a single processor, Cilk estimates (based on work and critical-path measurements) a parallelism of 14.8. Indeed, Fig. 2 shows that 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 serially. This results in relatively little parallelism. RBF, on the other hand, performs fine-grained high-quality speculative searches. Figure 2 shows that RBF uses 30 processors before stagnating. While Jamboree was able to speed up the search by a factor of 5.5, RBF achieves a speedup of 18.

4.3 The Performance of Parallel RBF 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 [1]

Table 1. Running times of our Chess program when it was executed on 1, 32 and 64 processors, and the relative speedups. 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	14.30	4.05	3.53	4.25	3.36
Pos4	28.33	125.90	3.55	35.46	3.75	33.57
Pos5	80	930.50	30.30	30.71	21.85	42.59
Pos9	7.67	16.80	1.10	15.27	1.05	16.00
Pos11	69	1104.05	38.55	28.64	31.60	34.94
Pos12	9.41	72.05	2.15	33.51	2.95	24.42
Cmb1	16.52	71.95	1.00	71.95	1.00	71.95
Cmb4	13.44	220.00	3.25	67.69	3.00	73.33
Cmb5	5.14	2.80	1.00	2.80	1.00	2.80
Cmb6	7	126.00	3.70	34.05	1.95	64.62
Cmb7	157	331.75	2.35	141.17	1.10	301.59
Cmb8	6.79	270.20	12.25	22.06	3.15	85.78
Cmb9	57.86	581.85	17.90	32.51	6.70	86.84
Cmb10	64	2724.30	89.45	30.46	37.80	72.07
Fin3	16.13	5.80	0.80	7.25	0.45	12.89
Fin4	155	162.40	7.10	22.87	3.65	44.49
Fin5	24.51	136.35	6.10	22.35	3.25	41.95
Fin6	11.25	206.45	6.55	31.52	3.05	67.69
Fin7	102	2831.30	123.15	22.99	53.85	52.58

to communicate between processes. Since RBF 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.

Table 1 shows that RBF often achieves a linear speedup, even on 64 processors. In some cases the speedup was super 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 super linear 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).

5 Summary

We have presented RBF, a selective search algorithm which is a randomized version of the Best-First algorithm. We have shown that RBF is efficient and highly parallel.

For incremental random game trees, RBF makes better decisions than Alpha-Beta search, and wins 90% of its game against it. We attribute this success to the accuracy of the evaluation function, which allows RBF to prune moves that seem bad without taking large risks.

In Chess the evaluation function can be inaccurate. This might cause RBF to examine the right move with a small probability. Consequently, RBF is slower than Alpha-Beta. Bear in mind that our experimental program was compared to a state-of-the-art Chess program.

As a parallel search algorithm, RBF performs fine-grained and high-quality speculations. This results in a high level of parallelism with little extra work. For random trees, RBF achieved a better speedup than Jamboree. On Chess test problems, our program achieved a linear speedup even on 64 processors.

Acknowledgement. Thanks to Richard Korf for pointing us to the Best-First algorithm. Thanks to Robert Hyatt for making Crafty publicly available and for allowing us to use it in this research.

References

1. Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputing Applications*, 8(3–4), 1994.
2. S. Fuller, J. Gaschnig, and J. Gillogly. Analysis of the alpha-beta pruning algorithm. Technical report, Department of Computer Science, Carnegie-Mellon University, July 1973.
3. Christopher F. Joerg and Bradley C. Kuszmaul. The *Socrates massively parallel chess program. In *Parallel Algorithms: Proceedings of The Third DIMACS Implementation Challenge*, number 30 in DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 117–140. AMS, 1996.
4. D. Knuth and R. Moore. An analysis of alpha-beta pruning. *Artifical Intelligence*, 6:293–326, 1975.
5. Richard E. Korf and David Maxwell Chickering. Best-first minimax search. *Artificial Intelligence*, 84:299–337, 1996.
6. Bradley C. Kuszmaul. The StarTech massively parallel chess program. *Journal of the International Computer Chess Association*, 18(1), March 1995.
7. Frederic Louguet. The Louguet Chess Test II (LCT II) FAQ. Available online at `http://www.icdchess.com/wccr/testing/LCTII/lct2faq.html`, April 1996.
8. Tony Marsland. Anatomy of a chess program. Brochure of the 8th World Computer Chess Championship, May 1995. Available online at `http://www.dcs.qmw.ac.uk/~icca/anatomy.htm`.
9. J. Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley, 1984.
10. 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`.