Problems in sequential and parallel game tree search

Bernhard Balkenhol[*]
Fakultät für Mathematik,
Universität Bielefeld,
Posfach 100131,
33501 Bielefeld,
Germany

October 29, 1996

Abstract

The α—β– algorithm is an efficient technique for searching game trees. As parallel computers become more available, it is important to have good parallel game tree search algorithms. Until now it is an open problem whether a linear speedup can be achieved with respect to sequential alpha–beta. This note presents a sample of game trees with distinct leaf values, which are easy for sequential alpha–beta – independent of the move ordering in the trees. We conjecture that these trees are difficult test cases for parallel algorithms.

[*]email:bernhard@mathematik.uni-bielefeld.de

Contents

1 Introduction

In the last years the development of parallel computers has made impressive progress. From the algorithmic point of view, game tree searching and especially alpha–beta seems to be hard to parallelise. Normally the running time of the alpha–beta algorithm depends heavily on the ordering of the tree [11]. Karp and Zhang [8, 16] were the first to design a parallel algorithm for searching *bivalued* game trees, which has a linear speedup (even in the worst case). For multi-valued game trees it is still a widely open problem whether a linear speedup can be achieved [5].

More oriented towards practice is the approach of Hsu [7], who belongs to the team of the current computer chess world champion "Deep Thought". He designed parallel alpha–beta versions with a linear speedup (and good speedup constants), if deep cutoffs in sequential alpha–beta are neglected.

However, in [9] (p. 310) there is an example of a uniform binary game tree of depth 4 with distinct leaf values in which alpha–beta will always find two cutoffs no matter how the branches of the tree are ordered. In [4] the example of Knuth and Moore [9] is generalised. It is shown that a whole class of trees exists which are "alpha–beta easy".

These "alpha–beta easy" trees may be interesting candidates for testing parallel game tree search algorithms. Since sequential alpha–beta works well in these trees independently of the move ordering, parallel algorithms will have to work hard to achieve good speedups.

2 Description of the problem

We describe the problem of parallel game tree search using t processors. The main question is how much faster a problem is parallel solvable against the sequential alpha–beta in relation to the number t of processors. Someone who is not confide in this problematic may assume that t parallel processors, i.e. working at the same time, can solve a problem t times faster than a single one. But this is not true in general. The essential problems in parallel programming are

- data exchange between processors,
- a processor has to wait for the result of an other one,
- a processor has to react upon the results of others,
- a processor does useless work.

For a constructed parallel algorithm we have to calculate how much faster this algorithm works in relation to the number of used processors against a single one.

In this context we are speaking about **speed–up** of the parallel algorithm against the sequential one. The speed–up can be studied in the average and in the worst case.

3 Motivation and known results

Game tree evaluation arises in many two–player games. In these games, two adversary players alternate in making moves, each viewing the opponent's failure as their own success. At each turn, each player has complete information about his opponent's position, and picks the most effective move. The game starts from a specified initial state and terminates in a class of specified states with outcomes from a specified range.

Often a computer has to play the part of one of the players. Typical examples are game–playing programs, such as for chess and Go, and theorem–proving programs for the first–order logic.

Using the following definitions we can describe a game on a tree. The starting position constitutes the root r of the tree T. Each possible game situation is marked by a node of the tree. A move of the game defines a directed edge concatenating the concerning game situations. The final situations (the situations where the game terminates) are characterised by the leaves of the tree.

3.1 Bivalued trees

An algorithm should decide whether the player who moves first is able to enforce a winning position of the game. Therefore the algorithm has to assume that both players are playing optimal. To label a game situation S the algorithm has to analyse whether the moving player is able to enforce victory or not. If he is able then we mark the situation with the value 1 otherwise with 0, i.e. is player A able to move in situation S with one move to a situation labelled with 0, then player B is not able to enforce victory out of this situation which means player A is winner of the game and the value of situation S is 1. On the other hand, if all successors of the node S are labelled with 1, then player B wins the game independent of the move of player A, i.e. situation S has to be valued with 0. A leaf of the game tree is labelled with 0 if the player who has to move has lost the game, otherwise the leaf value is 1. The value of a node v of the tree is equal to the $NAND$–connection of the values of the successors of the node v. Therefore the value of a node v depends only on the values of the reachable leaves.

3.1.1 S–Solve and P–Solve

Let $val(v)$ denote the value of a node v. We define **S–Solve** as

```
S-Solve (v:node): boolean;
if v is leaf then
      calculate value of v;
      return(val(v));
else
      let v(1), ... , v(d) be the successors of v;
      for i = 1 to d do
         b := S-Solve(v(i));
         if b=0 then
           return(1);
      return(0)
```

S–Solve is a sequential recursive algorithm. In each step it inquires the value of one leaf and hence using the $NAND$–connection he is able to calculate the value of the root r of the tree T recursively. This algorithm was first defined by Pearl [12] under the name “SOLVE”. In [8] it is called S–Solve (“S” for sequential) and its parallel version P–Solve (“P” for parallel). S–Solve examines the successors of a node with corresponding ordering from left to right. Remarkable is the fact, that it is not necessary to examine each leaf of the tree, because using the $NAND$–connection it is enough to read one 0 and the value of the node is 1 independent of the values of the remaining successors. Furthermore if a value of a node is known then it is not necessary to examine leaves which are reachable from this node.

Theorem 1 *(see [8]) In a k–uniform tree of depth n S–Solve reads in the best case $A(k,n)$ many leaves, where*

$$A(k,n) = \begin{cases} k^{\lfloor \frac{n}{2} \rfloor} & \textit{if } n \textit{ is odd and } val(r) = 1 \\ k^{\lceil \frac{n}{2} \rceil} & \textit{if } n \textit{ is odd and } val(r) = 0 \\ k^{\frac{n}{2}} & \textit{if } n \textit{ is even.} \end{cases}$$

Now we want to describe the parallel algorithm P–Solve of Karp & Zhang [8] which reaches a linear speed–up against S–Solve.

Definition 1 *(***speed–up***)*
Let $S(X)$ denote the number of steps which are necessary to solve a problem X with an arbitrary sequential algorithm S. Further let $P(X)$ denote the number of steps which are necessary to solve the same problem X with a parallel algorithm P using t processors.
We call a positive real constant c **speed–up–factor** *of P against S, if we have the equality*

$$P(X) = c \cdot \frac{S(X)}{t}.$$

In the case $c = 1$ we have that the parallel algorithm with t processors solves the problem t times faster than the sequential one. The value $c = 1$ is best possible if the sequential algorithm is optimal.

Suppose we have given a set of problems X_i with parallel algorithms P_i using t_i processors and sequential algorithms S_i for all $i \in I$, where I is a finite or infinite index set. Corresponding to the definition of speed–up we define $S_i(X_i)$ and $P_i(X_i)$.

Definition 2 *We call a positive real constant c* **linear** *speed–up–factor, if*

$$P_i(X_i) \leq c \cdot \frac{S_i(X_i)}{t_i} \textit{ for all } i \in I.$$

If the inequality above is not possible for a constant c but for a function $f(t_i)$ then this function defines the speed–up. This speed-up is not a linear one.

Definition 3 *(***dead, alive***)*
A node v of a tree T is called **dead** *in step t if his value or the values of his predecessors are known in step t. Otherwise he is called* **alive** *in step t.*

Definition 4 *(***pruning—number***)*
Let T be a game tree which is analysed by an algorithm A. Let r be the root of the tree and v a node of T of depth m, which is alive in step t. Furthermore let $r = u_{v,0}, \dots, u_{v,m} = v$ be the path from the root r to the node v. We define

$$a_{v,i} \triangleq \#\{w \mid w \text{ is a left alive brother of } u_{v,i} \text{ in step } t\},$$

for all $0 \leq i \leq m$. Now we can define the **pruning–number** *$PN_A(c,T,t)$ of v in step t as*

$$PN_A(v,T,t) \triangleq \sum_{i=0}^{m} a_{v,i}(t).$$

The notion "in step t" means, that step $t-1$ has finished and no further leaf has been examined.

Definition 5 P–Solve of width w
P–Solve of width w examines in each step all leaves with a pruning–number $\leq w$.

3.1.2 Known results on speedups for bivalued trees

Althöfer (1988): It is possible to attain an average–speed–up of $\frac{t}{201}$ with t processors on trees of depth $6 \cdot t \cdot \lceil log_2(t) \rceil$, if the leaf values are independent and identical distributed.

Karp & Zhang, Zhang (1989): In each binary tree of depth t it is possible to attain a linear speed–up of $\frac{t+1}{76}$ with $t+1$ processors.

Harting (1991): In each binary tree of depth t it is possible to attain a linear speed–up of $\approx \frac{t^m}{76^m}$ with t^m processors ($m \geq 1$ fixed).

Lin & Pippenger (1992): In each k–uniform tree of depth t it is possible to attain a linear speed–up with t^m processors.

3.2 Minimax trees

Minimax trees are game trees with an other recursive specification. Let v be a node of a minimax–tree T. We calculate the values of all nodes v with even depth as the minimum of the values of the successors. These nodes v are called **min–nodes**. The value of a node v with an odd depth is calculated as the maximum of the values of the successors. These nodes are called **max–nodes**.

It is possible to study these trees with more than two different leaf values [9]. Until now it is an open problem whether a linear speed–up can be achieved on this trees. In May 1989 Karp and Zhang [8] have announced a solution to this problem. Half a year later (December 1989) Zhang has revised this announcement (see [16]).

We choose the classical alpha–beta–algorithm as the sequential algorithm on minimax trees. We want to construct a parallel algorithm on multi valued game trees which attains a linear speed–up against this sequential algorithm. For the required time of the parallel algorithm we ignore loss of time by data transmission between processors. Rather we calculate the time by the number of steps of the program, where a step means that each processor examines one leaf of the tree. We have deviation from a linear speed–up if one or more processors examine leaves which are insignificant for the calculation of the root value. The less number of leaf values have to be examined the more difficult it is to define an order on the leaves, i.e. to attain a linear speed–up. The trees constructed in [4] are easy for the sequential alpha–beta, no matter how the tree is ordered. We conjecture that these trees are difficult test cases for parallel algorithms.

Now we are looking at the alpha–beta algorithm searching k–uniform minimax trees. The leaves we have to examine are depending on their values and heavily on the ordering of the tree [11]. In [9] Knuth and Moore have shown:

1. The alpha–beta algorithm has to examine in each k–uniform minimax tree of depth n with best possible ordering $k^{\lceil \frac{n}{2} \rceil} + k^{\lfloor \frac{n}{2} \rfloor} - 1$ many leaves.
2. There exist k–uniform minimax trees of depth n where the alpha–beta algorithm has to examine in the worst case k^n many leaves.

3.2.1 The sequential algorithm

In practice, the alpha–beta pruning procedure is the best known heuristic for game–tree evaluation: leaves are evaluated in the order visited by a depth–first search starting from the left; if the values seen so far make the evaluation of an unexplored sub–tree redundant, that sub–tree is pruned and none of its leaves inspected. In addition to the popularity of alpha–beta pruning in practice, it has been shown that within certain models of computation this heuristic (or its boolean version) is optimal [12].
The reader is refereed to Zhang's thesis [16] for an excellent survey of practical work on parallel game tree search and its speedup against alpha–beta pruning.

In order to construct an upper bound we are looking at the ordering of the tree where the number of leaves we have to examine is getting maximal. We call this ordering **worst ordering** of the tree. The alpha–beta algorithm stops searching subtrees if $\alpha \geq \beta$. The alpha–beta algorithm examines exactly the number of leaves presented by Knuth and Moore if all leaf values are equal. In order to get an upper bound for all k–uniform minimax trees of depth n we assume that the leaf values are distinct. So the task is to find a k–uniform minimax tree of depth n with distinct leaf values where the number of leaves the alpha–beta algorithm has to examine is getting minimal in the worst ordering of the tree. We call these trees **optimal**.

In [3] a construction is presented for the simplest nontrivial case (binary trees). The upper bound is generalised to the k–uniform case in [4].

Proposition 1 *For each $n \geq 0$ there exists a k–uniform minimax tree of depth n with distinct leaf values where the alpha–beta algorithm has to examine at most*

$$\left(\frac{(k-1)^2 \cdot n^2}{8 \cdot k} + \frac{k^2-1}{4 \cdot k} \cdot n + 1\right) \cdot k^{\frac{n}{2}}$$

many leaves, if n is even and at most

$$\left(\frac{(k-1)^2 \cdot n^2}{8 \cdot k} + \frac{k^2-1}{2 \cdot k} \cdot n + 1 - \frac{k^2+2 \cdot k-3}{8 \cdot k}\right) \cdot k^{\frac{n+1}{2}}$$

many leaves, if n is odd independent of the ordering of the tree.

Proposition 2 *In each k–uniform minimax tree of depth n with distinct leaf values the alpha–beta algorithm has to examine in the worst ordering of the tree at least*

$$3 \cdot (k+1) \cdot k^{\frac{n}{2}} - \frac{(k-1)^2}{2} \cdot n^2 - \frac{k^2+2 \cdot k-3}{2} \cdot n - 3 \cdot k - 2$$

many leaves if n is even and at least

$$(k^2+4 \cdot k+1) \cdot k^{\frac{n-1}{2}} - \frac{(k-1)^2 \cdot (n-1)^2}{4} - (k^2-1) \cdot n - 3 \cdot k - 2$$

many leaves if n is odd.

Proposition 3 *For each $n \geq 0$ there exists a binary minimax tree of depth n with distinct leaf values in which the alpha–beta algorithm has to examine in the worst ordering of the tree less than $43 \cdot 2^{\frac{n}{2}}$ many leaves if n is even and less than $61 \cdot 2^{\frac{n-1}{2}}$ many leaves if n is odd.*

In section 4 we sketch the proves of the last three propositions. They can be found more detailed in [4].

3.3 Concluding remarks for the construction

Remark 1: Our construction of "alpha–beta easy" game trees is top–down. At first glance it seems strange to make a *top–down* construction of an object with a *bottom–up* structure. However, often it is easier to obtain theoretical results or to make simulation runs for top–down than for bottom–up models. Already Schrüfer [14] has developed a nice stochastic

game tree model with top–down probabilities, which turned out to be very convenient for reliability and complexity investigations [1].
One key point has to be observed in top–down models: Although the values of the inner nodes are known to the constructor during his construction, they are assumed to be not known to the person or machine that has to search and evaluate the tree.

Remark 2: We are aware of the fact that our "alpha–beta easy" game trees may be very rare objects in practice, for instance in computer chess, or in certain random tree models. So it is possible (or even probable) that problems, which parallel algorithms may have with our trees, may not be reflected in their performance in real life.

Remark 3: There are uniform binary minimax trees in which "worst first" move ordering in all inner nodes is less expensive than some other orderings, with respect to the number of leaves visited by alpha–beta.

Remark 4: In every ordered game tree the undirectional SSS* algorithm [15] visits only a subset of those leaves visited by alpha–beta [13]. Hence the trees of this note are also easy for SSS*.

4 Construction of the trees

4.1 The upper bound for k-uniform trees

We construct the "alpha–beta easy" trees in a top–down manner (from the root to the leaves). For every node u five real values a_u, b_u, c_u, d_u, e_u are fixed with

$$a_u < b_u < c_u < d_u < e_u \tag{4.1}$$

a_u, b_u, d_u and e_u are only auxiliary values. c_u finally becomes the game theoretic value of u. We set

$$X_u \triangleq \{x \mid a_u < x < b_u \text{ or } x = c_u \text{ or } d_u < x < e_u\}.$$

The root r is a min–node and gets $a_r = 0, b_r = 1, c_r = 2, d_r = 3, e_r = 4$.
Now assume that for some inner min node u the values $a_u, \ldots, e_u$ are already fixed. For the k successors $1, \ldots, k$ of u (all of them are max–nodes) we set:

$$
\begin{array}{rclcll}
a_1 & = & a_u, \quad b_1 = b_u \quad , & & c_1 = c_u & \\
a_i & = & \frac{5\cdot k+2-3\cdot i}{5\cdot k-4} \cdot d_u & + & \frac{3\cdot i-6}{5\cdot k-4} \cdot e_u & \forall \quad 2 \le i \le k \\
b_i & = & \frac{5\cdot k+1-3\cdot i}{5\cdot k-4} \cdot d_u & + & \frac{3\cdot i-5}{5\cdot k-4} \cdot e_u & \forall \quad 2 \le i \le k \\
c_i & = & \frac{5\cdot k-3\cdot i}{5\cdot k-4} \cdot d_u & + & \frac{3\cdot i-4}{5\cdot k-4} \cdot e_u & \forall \quad 2 \le i \le k \\
d_i & = & \frac{2\cdot i-1}{5\cdot k-4} \cdot d_u & + & \frac{5\cdot k-3-2\cdot i}{5\cdot k-4} \cdot e_u & \forall \quad 1 \le i \le k \\
e_i & = & \frac{2\cdot i-2}{5\cdot k-4} \cdot d_u & + & \frac{5\cdot k-2-2\cdot i}{5\cdot k-4} \cdot e_u & \forall \quad 1 \le i \le k
\end{array}
$$

such that:

$\big((a_u = a_1) < (b_u = b_1) < (c_u = c_1)\big) < \big((d_u = a_2) < b_2 < c_2\big) < (a_3 < b_3 < c_3) < \ldots < (a_k < b_k < c_k) < (d_k < e_k) < \ldots < (d_2 < e_2) < d_1 < (e_1 = e_u)$

Note that condition 4.1 is satisfied by these assignments.
For an inner max–node u with successors $1, \ldots, k$ we set:

$$
\begin{array}{rclcll}
c_1 & = & c_u & & & \\
a_i & = & \frac{3\cdot k+1-2\cdot i}{3\cdot k-1} \cdot a_u & + & \frac{2\cdot i-2}{3\cdot k-1} \cdot b_u & \forall \quad 1 \le i \le k \\
b_i & = & \frac{3\cdot k-2\cdot i}{3\cdot k-1} \cdot a_u & + & \frac{2\cdot i-1}{3\cdot k-1} \cdot b_u & \forall \quad 1 \le i \le k \\
c_i & = & \frac{i-1}{3\cdot k-1} \cdot a_u & + & \frac{3\cdot k-i}{3\cdot k-1} \cdot b_u & \forall \quad 2 \le i \le k \\
d_i & = & \frac{2\cdot i-1}{2\cdot k-1} \cdot d_u & + & \frac{2\cdot k-2\cdot i}{2\cdot k-1} \cdot e_u & \forall \quad 1 \le i \le k \\
e_i & = & \frac{2\cdot i-2}{2\cdot k-1} \cdot d_u & + & \frac{2\cdot k-2\cdot i+1}{2\cdot k-1} \cdot e_u & \forall \quad 1 \le i \le k
\end{array}
$$

such that:

$a_u < (a_1 < b_1) < (a_2 < b_2) < \ldots < (a_k < b_k) < c_k < \ldots < c_2 < b_u$
$< (c_1 = c_u) < d_u < (d_k < e_k) < \ldots < (d_2 < e_2) < (d_1 < e_1) < e_u$

These values again satisfy condition 4.1. Additionally the construction yields $X_i \cap X_j = \emptyset$ for all $i \neq j$ and $X_i \cup \ldots \cup X_k \subset X_u$.

4.1.1 Analysis of the tree

Now we are getting three possible constellations for the bound β and the parameters of a node u. (For our construction above an additional inspection of the α–bounds does not lead to better run time estimations.) These constellations are:

Number	0	1	2
β	$e_u < \beta$	$c_u < \beta < d_u$	$\beta < a_u$.

By the construction of the tree, only these three constellations can occur during an alpha–beta search, independently of the tree ordering. More formally, for $0 \leq i \leq 2$ let $m_i(t)$ denote the largest possible number of leaves visited by alpha–beta in an arbitrarily ordered subtree T_u of depth $2 \cdot t$, if the actual β–bound for the root u of T_u (u is a min–node) at the beginning of the search in T_u obeys inequality number i of the three possibilities. Analogously $M_i(t)$ is defined to be the largest possible number of leaves visited by alpha–beta in an arbitrarily ordered subtree of depth $2 \cdot t + 1$ with respective β–bound. With this notation we get:

1. $M_2(t) = m_2(t)$.

   Proof : From the k subtrees of the node u let us first examine the j'th one. Before we examine j we are in constellation 2 which means the inequality $\alpha < \beta < a_u \leq a_j$ holds. Therefore alpha–beta has to visit $m_2(t)$ many leaves of j (independent which subtree j is the first one). After that we set $\alpha(new) = c_j > a_u > \beta$ such that the alpha–beta algorithms stops.

2. $m_2(t+1) = k \cdot M_2(t)$

   Proof : $\alpha < \beta < a_u = a_1$. By construction $a_1 \leq a_i$ for all $1 \leq i \leq k$. Therefore during the examination of the subtrees β can not decrease, i.e. in each of the k subtrees we have to examine $M_2(t)$ many leaves (independent of the ordering).

3. From 1. and 2. we are getting by induction in t:

$$M_2(t) = m_2(t) = k^t$$

4. $M_1(t) = k \cdot m_1(t)$

   Proof : It holds: $\alpha < \beta$ and $c_u < \beta < d_u$. By the construction of the tree we have: $c_j \leq c_1 = c_u < \beta < d_u < d_k \leq d_j$ for all $1 \leq j \leq k$. Therefore we can either not change the constellation nor increase α bigger than c_1.

5. $m_1(t+1) = M_1(t) + (k-1) \cdot M_2(t)$

   Proof : In order to analyse the worst case we can assume that $\alpha < a_u$ holds. As long as $j > 1$ we have $a_j > d_u > \beta$,i.e. $M_2(t)$. For $j = 1$ we have $c_j < \beta < d_j$, i.e. $M_1(t)$. After the examination we get $\beta(new) = c_j$. In the case where $\alpha < c_1$ we have to examine in all further subtrees $M_2(t)$ many leaves, in the case $\alpha \geq c_1$ the inequality $\beta(new) < \alpha$ holds, i.e. the alpha–beta algorithm stops. In the worst ordering of the tree $j = 1$ is the last subtree.

6. From 3.,4. and 5. we get
$$m_1(t+1) = k \cdot m_1(t) + (k-1) \cdot k^t$$
and by induction in t:
$$m_1(t) = \left(\frac{k-1}{k} \cdot t + 1\right) \cdot k^t \text{ and } M_1(t) = \left(\frac{k-1}{k} \cdot t + 1\right) \cdot k^{t+1}.$$
7. $M_0(t) = k \cdot m_0(t)$
Proof : By construction $c_j < e_u < \beta$ for all $1 \leq j \leq k$, i.e. we can either not change the constellation nor get $\alpha > \beta$.
8. $m_0(t+1) = M_0(t) + (k-1) \cdot M_1(t)$
Proof : From 3. and 6. we get that $m_1(t) > m_2(t)$ and $M_1(t) > M_2(t)$ for all $t > 0$. As starting constellation we have $\alpha < \beta$ and $e_u < \beta$. Let j be the first subtree the alpha–beta algorithm examines. From j we have to read $M_0(t)$ many leaves then $\beta(neu) = c_j$. With the condition $c_j < d_u$ for all $1 \leq j \leq k$ the following three cases can occur: We have to add $M_1(t)$ or $M_2(t)$ or the algorithm stops. Choose the ordering of the subtrees as $k, \ldots, 1$ then we need for the first subtree $M_0(t)$ and for all others $M_1(t)$.
9. From 4.,7. and 8. we get
$$m_0(t+1) = k \cdot m_0(t) + (k-1) \cdot k \cdot \left(\frac{k-1}{k} \cdot t + 1\right) \cdot k^t$$
and by induction in t:
$$m_0(t) = \left(\frac{(k-1)^2}{2 \cdot k} \cdot t^2 + \left(k - 1 - \frac{(k-1)^2}{2 \cdot k}\right) \cdot t + 1\right) \cdot k^t$$
and
$$M_0(t) = \left(\frac{(k-1)^2}{2 \cdot k} \cdot t^2 + \left(k - 1 - \frac{(k-1)^2}{2 \cdot k}\right) \cdot t + 1\right) \cdot k^{t+1}.$$

Now the upper bounds of Proposition 1 are proved for even n with $m_0(\frac{n}{2})$ and for odd n with $M_0(\frac{n-1}{2})$. □

4.2 The lower bound for k-uniform trees

During the alpha–beta–search of a tree only 8 constellations can occur:

Case	Alpha	Beta
0	$\alpha < x$	$\beta > x$
1	$\alpha > x$	$\beta > x$
2	$\alpha < x$	$\beta < x$
3	$\alpha < x$	$\beta = +\infty$
4	$\alpha > x$	$\beta = +\infty$
5	$\alpha = -\infty$	$\beta < x$
6	$\alpha = -\infty$	$\beta > x$
7	$\alpha = -\infty$	$\beta = +\infty$

For $0 \leq i \leq 7$ let $l_i(t)$ denote the smallest possible number of leaves visited by alpha–beta in an worst ordered subtree T_u of depth $2 \cdot t$, if the actual root u of T_u is a min–node. Analogously $L_i(t)$ is defined to be the smallest possible number of leaves visited by alpha–beta in an worst ordered subtree T_u of depth $2 \cdot t + 1$, if the actual root u of T_u is a max–node.
We have a bad (but not necessary the worst one) ordering of a tree if the game theoretic values of the successors of a min–node are decreasing and the values of the successors of a max–node are increasing. We call this order "worst first" ordering of the tree.

4.2.1 Analysis of the cases

Case 0:

The game theoretical values of the successors of a min–node are in decreasing order. The smallest one is getting the game theoretical value of the min–node. Let x be the value of the min–node. In this case we have $\alpha < x$ which means α is less than all the values of the successors. Now it depends only from beta which cases we have to choose for the subtrees. If beta is larger then the value of the first successor then beta is larger then the values of all successors because the first one is the largest one by the ordering of the tree. In this case we have to choose case 0 for all subtrees. If beta is smaller then the value of the first successor then we have case 2 for the first successor and for the next one again 0 or 2. Therefore we get case 2 as long as beta is smaller then the values of the successors then we have to continue with case 0. In the initial state we are starting with $\beta > x$, i.e. in the last subtree we have to use case 0. Analogously for a max–node we have after case1 the possibilities 0 and 1, after case 0 we have to continue with 0, in the last subtree we have always to use case 0.

$$
\begin{aligned}
l_0(t+1) &= d \cdot L_0(t) + (k-d) \cdot L_2(t), && \text{for a } d \text{ with } 1 \leq d \leq k. \\
L_0(t) &= d \cdot l_0(t) + (k-d) \cdot l_1(t), && \text{for a } d \text{ with } 1 \leq d \leq k.
\end{aligned}
$$

Case 1:

$$
\begin{aligned}
&l_1(t+1) && = a \cdot L_0(t) + L_1(t) + b \cdot L_2(t), \\
&\text{for } a \text{ and } b \text{ with} && 0 \leq a \leq k-1, 0 \leq b \leq k-1, \quad a+b+1 \leq k. \\
&L_1(t) && = k \cdot l_1(t).
\end{aligned}
$$

Case 2:

$$
\begin{aligned}
&l_2(t+1) && = k \cdot L_2(t) \\
&L_2(t) && = a \cdot l_0(t) + b \cdot l_1(t) + l_2(t) \\
&\text{for } a \text{ and } b \text{ with} && 0 \leq a \leq k-1, 0 \leq b \leq k-1, \quad a+b+1 \leq k.
\end{aligned}
$$

Case 3:

$$
\begin{aligned}
l_3(t+1) &= L_3(t) + (k-1) \cdot L_0(t) \\
L_3(t) &= d \cdot l_3(t) + (k-d) \cdot l_4(t) \quad \text{for a } d \text{ with } 1 \leq d \leq k.
\end{aligned}
$$

Case 4:

$$\begin{array}{ll} l_4(t+1) & = L_4(t) \text{ or} \\ l_4(t+1) & = L_3(t) + d \cdot L_1(t) + (k-d-1) \cdot L_0(t), \\ & \text{for a } d \text{ with } 1 \leq d \leq k-1. \\ L_4(t) & = k \cdot l_4(t). \end{array}$$

Case 5:

$$\begin{array}{ll} l_5(t+1) & = k \cdot L_5(t) \\ L_5(t) & = l_5(t) \text{ or} \\ L_5(t) & = l_6(t) + a \cdot l_0(t) + b \cdot l_2(t) \\ \text{for } a \text{ and } b \text{ with} & 0 \leq a \leq k-1, 0 \leq b \leq k-1, \quad a+b+1 \leq k. \end{array}$$

Case 6:

$$\begin{array}{ll} l_6(t+1) & = d \cdot L_6(t) + (k-d) \cdot L_5(t), \quad \text{for a } d \text{ with } 1 \leq d \leq k. \\ L_6(t) & = l_6(t) + (k-1) \cdot l_0(t). \end{array}$$

Case 7:

$$\begin{array}{ll} l_7(t+1) & = L_7(t) + (k-1) \cdot L_6(t), \\ L_7(t) & = l_7(t) + (k-1) \cdot l_3(t). \end{array}$$

In each of the 8 cases we have to add k terms either for $l_i(t+1)$ or for $L_i(t)$, i.e. the inequalities $l_i(t) \geq k^t$ and $L_i(t) \geq k^t$ hold for all $0 \leq i \leq 7$. Therfore the functions $l_1(t+1), L_1(t), l_2(t+1), L_2(t), l_4(t+1), L_4(t), l_5(t+1), L_5(t)$ are minimal if we choose

$$l_1(t+1) = L_1(t), l_2(t+1) = k \cdot L_2(t), l_4(t+1) = L_4(t), L_5(t) = l_5(t).$$

Additionally the equalities yield

$$(L_2(t) = L_5(t)) \leq (L_1(t) = L_4(t)) \text{ and}$$

$$l_1(t+1) = l_2(t+1) = l_4(t+1) = l_5(t+1).$$

With these results we get:

Case	min–node $l_i(t+1)$	max–node $L_i(t)$
0	$L_0(t) + L_2(t) \cdot (k-1)$	$l_0(t) + l_1(t) \cdot (k-1)$
1	$L_1(t)$	$l_1(t) \cdot k$
2	$L_2(t) \cdot k$	$l_2(t)$
3	$L_3(t) + (k-1) \cdot L_0(t)$	$l_3(t) + (k-1) \cdot l_4(t)$
4	$L_4(t)$	$l_4(t) \cdot k$
5	$L_5(t) \cdot k$	$l_5(t)$
6	$L_6(t) + (k-1) \cdot L_5(t)$	$l_6(t) + (k-1) \cdot l_0(t)$
7	$L_7(t) + (k-1) \cdot L_6(t)$	$l_7(t) + (k-1) \cdot l_3(t)$

Using these recursions we get by induction on t:

1. $l_1(t) = l_2(t) = l_4(t) = l_5(t) = k^t$,
2. $l_0(t) = 2 \cdot k^t - 1$,
3. $L_0(t) = k^{t+1} + k^t - 1$,

4. $l_3(t) = \frac{k^2+k-2}{k-1} \cdot k^t - (k-1) \cdot t - \frac{k^2+k-2}{k-1} + 1$,
5. $L_3(t) = k^{(t+1)} + \frac{k^2-3}{k-1} \cdot k^t - (k-1) \cdot t - \frac{k^2+k-2}{k-1} + 1$,
6. $L_5(t) = l_5(t) = k^t$,
7. $l_6(t) = 3 \cdot k^t - (k-1) \cdot t - 2$,
8. $L_6(t) = 2 \cdot k^{(t+1)} + k^t - (k-1) \cdot t - k - 1$,
9. $l_7(t) = (3 \cdot k + 3) \cdot k^t - (k-1)^2 \cdot t^2 - (k^2 + 2 \cdot k - 3) \cdot t - 3 \cdot k - 2$,
10. $L_7(t) = (k^2 + 4 \cdot k + 1) \cdot k^t - (k-1)^2 \cdot t^2 - (2 \cdot k^2 - 2) \cdot t - k^2 - 3 \cdot k - 1$.

Now the lower bounds of Proposition 2 are proved for even n with $l_7(\frac{n}{2})$ and for odd n with $L_7(\frac{n-1}{2})$. □

4.3 Remarks on the results

Lemma 1 *Let $h(t) = p(t) \cdot k^t$ denote the largest possible number of leaves visited by alpha–beta in an arbitrarily ordered tree T of depth $2 \cdot t$, where $p(t)$ is a polynom in t. Let $m_0(t)$ denote the upper bound of Proposition 1. If $h(t) \leq m_0(t)$ for all $t \geq 0$ then $h(t) = m_0(t)$.*

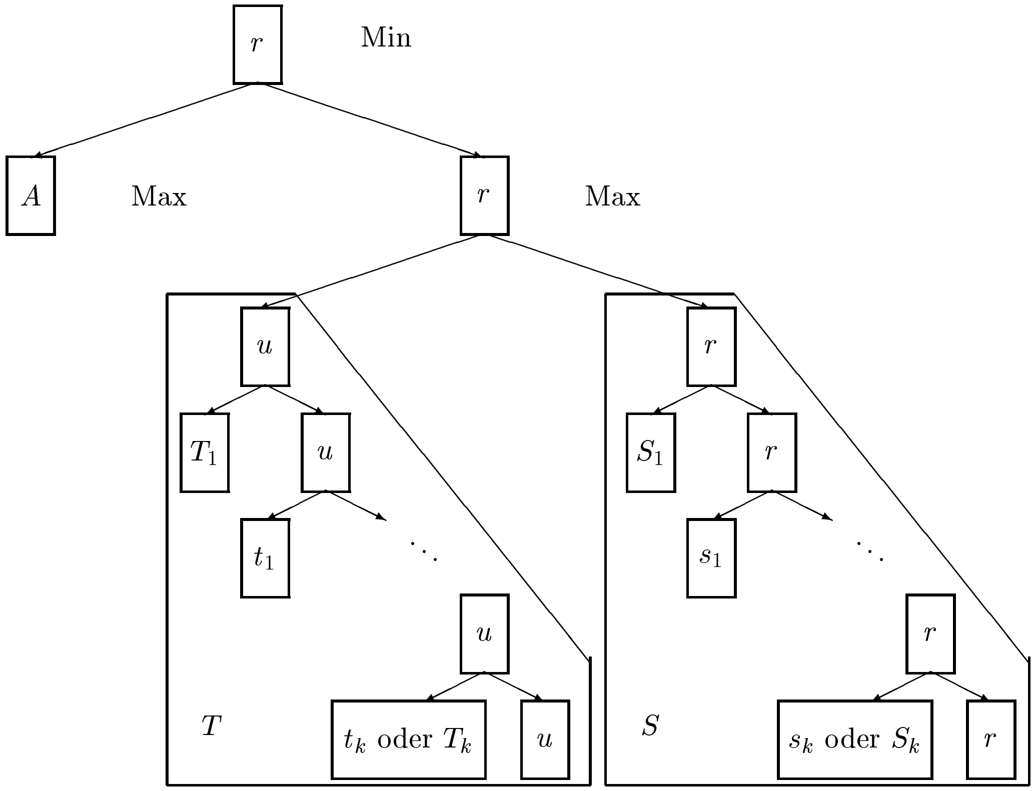
Proof : By the fact that $h(t) \leq m_0(t)$ for all $t \geq 0$ follows that the degree of p is less or equal to two. The lower bound of Proposition 2 and $m_0(t)$ are equal for all $0 \leq t \leq 2$. Suppose p is a polynom of degree two. By solving the system of linear equations we have proved the Lemma. □

Remark 1 *In order to get bounds it does not matter if the root of the tree is a min–node or a max–node, because if we replace each node w by $-w$ then we have changed the roll of min– and max–nodes and also of alpha and beta. Therefore we are getting the same bounds.*

Remark 2 *The worst–first ordering of a tree is not necessary the worst one.*

4.4 The upper bound for binary trees

Let M be a minimax–tree of depth n, which is constructed in the following way:



With T and S we denote the corresponding subtrees of M.

Notation: Let $B(T)$ denote the set of all leaf values of T. We write $B(T) < x$ if $y < x$ for all $y \in B(T)$ and we write $B(S) < B(T)$ if $x < y$ for all $x \in B(S)$ and for all $y \in B(T)$.

We construct the subtrees S and T such that
$B(s_k) < B(s_{k-1}) < \ldots < B(s_1) < B(t_k) < \ldots < B(t_1) < u < r < B(A) < B(T_1) < B(T_2) < \ldots < B(T_j) < B(S_1) < \ldots < B(S_j)$
with $j = k$, if n is even and $j = k - 1$, if n is odd.
The subtrees t_i and s_i are constructed as the complete tree M, the subtrees T_i and S_i like the inversion of the complete tree M.
Let n_T denote the depth of the subtree T and let U be the root of this subtree.

Lemma 2 *If $B(t_i) < \alpha \le u \le \beta < B(T_j)$ for all i, j then the alpha–beta–algorithm visits $r(n_T) \triangleq 2^{\frac{n_T}{2}+1} - 1$ many leaves of T if n_T is even and $r(n_T) \triangleq 2^{\frac{n_T+1}{2}} + 2^{\frac{n_T-1}{2}} - 1$ many leaves if n_T is odd.*

Proof : With $\beta < B(T_j)$ we have to examine only one successor of each max–node of the T_j ‘s and with $B(t_i) < \alpha$ we have to examine only one successor of each min–node of the t_i ‘s. Now the proof is done by induction in the depth of the subtrees. $\square$

Remark 3 *The function $r(n)$ represents the number of leaves visited by alpha–beta in a tree of depth n, if the tree ordering is best possible (see [9]).*

Lemma 3 *Let $m(n)$ denote the number of leaves visited by alpha–beta in a tree of depth n in the worst ordering of the tree. $f(n) \triangleq m(n) - r(n)$ is a monotone increasing function in n.*

Proof : Clearly $f(n) \geq 0$ for all $n \geq 0$.
In a tree M of depth n we have to visit in the worst case $m(n-1)$ many leaves of the left subtree of M. Using the result of Knuth and Moore we have to examine of the right subtree at least $\frac{1}{2} \cdot 2^{\frac{n}{2}}$ many leaves, if n is even and $2^{\frac{n-1}{2}}$ many leaves, if n is odd. Therefore $m(n) \geq m(n-1) + \frac{1}{2} \cdot 2^{\frac{n}{2}}$, if n is even and $m(n) \geq m(n-1) + 2^{\frac{n-1}{2}}$, if n is odd. With $r(n) - r(n-1) = \frac{1}{2} \cdot 2^{\frac{n}{2}}$, if n is even and $r(n) - r(n-1) = 2^{\frac{n-1}{2}}$, if n is odd we get
$m(n) - r(n) \geq m(n-1) - r(n-1)$. □

Lemma 4 *We examine the subtree T of M in the worst ordering (for arbitrary alpha and beta), if we examine the subtrees in the order $T_1, t_1, T_2, t_2, \ldots$.*

Proof :

1. Suppose we examine T_1 first.

   If alpha and beta are chosen in such a way, that we have some cutoffs in T_1 the we have them independent of the ordering of the tree. Therefore we can assume that $\alpha = -\infty$ and $\beta = \infty$. In this case we have to examine $m(n_T - 1)$ many leaves of T_1. By construction $B(T_1) > B(t_1)$, i.e. the updating of the beta–bound has no influence on the number of leaves visited in t_1, in other words we have to examine in the worst case $m(n_T - 2)$ many leaves of t_1. In all other subtrees of T we have the constellation $B(t_i) < \alpha \leq u \leq \beta < B(T_j)$ for all i, j. Using Lemma 2 we get $m(n_T) = m(n_T - 1) + m(n_T - 2) + r(n_T - 2)$.

2. Suppose we examine the right subtree of T first.

   By Induction we have to examine $m(n_T - 2) + m(n_T - 3) + r(n_T - 3)$ many leaves of this subtree. Now $\beta = u$ and by construction $B(T_1) > u$, i.e. alpha–beta visits $2^{\frac{n_T - 1}{2}}$ many leaves of T_1, if n_T is odd and $2^{\frac{n_T - 2}{2}}$ many leaves, if n_T is even.

It remains to show:

$$m(n_T-1)+m(n_T-2)+r(n_T-2) \geq m(n_T-2)+m(n_T-3)+r(n_T-3)+2^{\frac{n_T-1}{2}},$$

if n_T is odd and

$$m(n_T-1)+m(n_T-2)+r(n_T-2) \geq m(n_T-2)+m(n_T-3)+r(n_T-3)+2^{\frac{n_T-2}{2}},$$

if n_T is even.

Using the definition of $r(n)$ we get

$$m(n_T-1) \geq m(n_T-3) + \begin{cases} \frac{1}{2}\cdot 2^{\frac{n_t-1}{2}}, & \text{if } n_T \text{ is odd and} \\ \frac{1}{4}\cdot 2^{\frac{n_t}{2}}, & \text{if } n_T \text{ is even.} \end{cases}$$

From Lemma 3 it follows

$$\begin{aligned} m(n_t-1) &\geq m(n_T-3) + r(n_T-1) - r(n_T-3) \\ &= m(n_T-3) + \begin{cases} \frac{3}{2}\cdot 2^{\frac{n_T-1}{2}} & \text{if } n_T \text{ is odd and} \\ \frac{3}{4}\cdot 2^{\frac{n_T}{2}} & \text{if } n_T \text{ is even.} \end{cases} \end{aligned}$$

Therefore the statement of Lemma 4 is proved. □

Now we analyse the tree M in two steps.

1. We examine the subtree A first.

   In this case alpha–beta has to visit $m(n-1)$ many leaves of A. The root of M is a min–node. Therefore we reduce the beta–bound to the root–value of A. By construction it does not matter whether we search S first or T first. Using Lemma 4 we examine the subtrees of T (of S) from the left to the right. With $B(A) < B(T_1)$ we have to examine $2^{\frac{n-3}{2}}$ many leaves of T_1, if n is odd and $2^{\frac{n-4}{2}}$ many leaves, if n is even. Now we have the constellation $\alpha < B(t_1) < \beta$, i.e. we have to examine $m(n-4)$ many leaves of t_1. After we have increased alpha to the root–value of t_1 we have for all other subtrees the constellation $B(t_i) < \alpha \leq u \leq \beta < B(T_j)$ for all $i > 1$ and $j > 1$. With Lemma 2 alpha–beta visits $r(n-4)$ many leaves of these subtrees. Furthermore we have now the constellation $b(s_i) < \alpha \leq r \leq \beta < B(S_j)$, i.e. by Lemma 2 alpha–beta visits $r(n-2)$ many leaves of S. Totally we get

   $$m(n) \leq m(n-1)+m(n-4)+r(n-2)+r(n-4)+\begin{cases} 2^{\frac{n-3}{2}}, \\ \text{if } n \text{ is odd and} \\ 2^{\frac{n-4}{2}} \\ \text{if } n \text{ is even.} \end{cases}$$

2. We examine the right subtree of M first.

   By construction it does not matter weather we examine T or S first. As we have seen during the proof of Lemma 4 ($n_T = n-2$) we have to examine $m(n-3) + m(n-4) + r(r-4)$ many leaves of T. Now $\alpha = u < B(S_1)$, i.e. alpha–beta visits $m(n-3)$ many leaves of S_1. For all other subtrees the inequalities $b(s_i) < \alpha \leq r \leq \beta < B(S_j)$ hold, such that alpha-beta visits $r(n-3)$ further leaves. By construction $r < B(A)$ and we have to examine $2^{\frac{n-2}{2}}$ many leaves of A, if n is even and $2^{\frac{n-1}{2}}$ many leaves, if n is odd.

   Totally we get

$$m(n) \leq 2{\cdot}m(n-3)+m(n-4)+r(n-3)+r(n-4)+\begin{cases} 2^{\frac{n-1}{2}}, \\ \text{if } n \text{ is odd and} \\ 2^{\frac{n-2}{2}}, \\ \text{if } n \text{ is even.} \end{cases}$$

Using both results Proposition 3 can be proved by induction in n. □

5 A new parallel algorithm

Before we are able to present the parallel algorithm we have to define an ordering of the leaves of a tree T. Therefore each node v of the tree T gets a further value $val_{ord}(v)$. Let r be the root of the tree T. We define these further values by starting the following procedure with Const–Node(r);

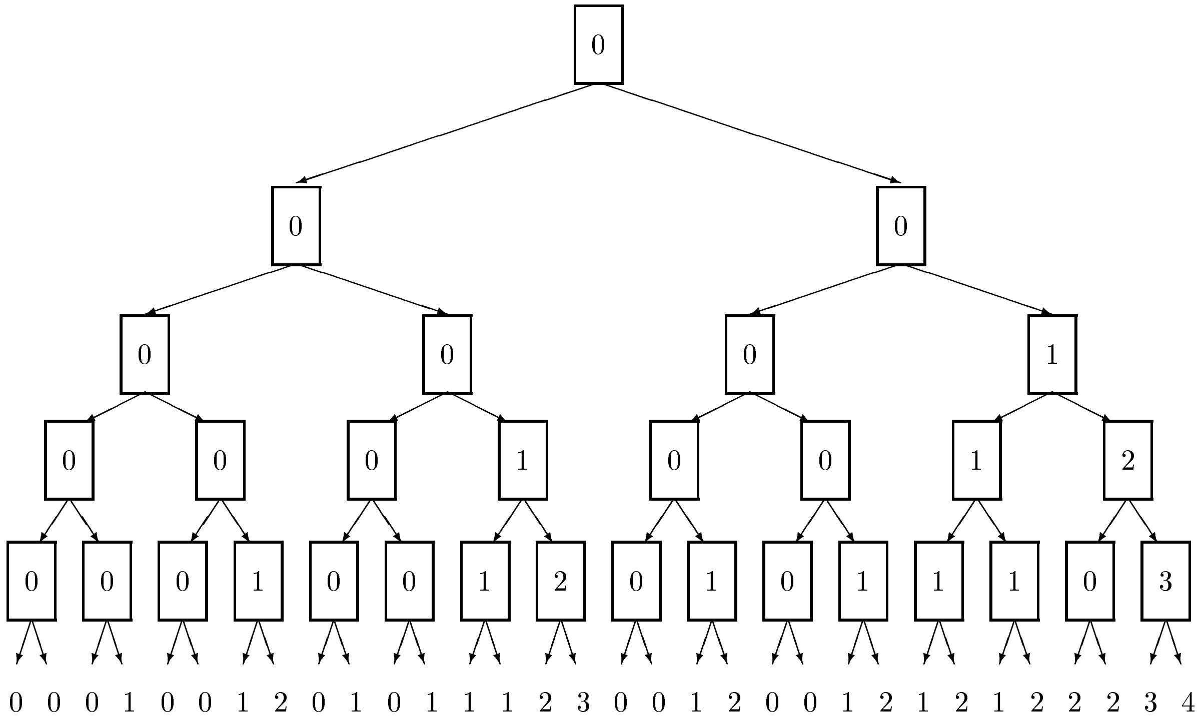
```
Const-Node (v:node)
   set val_ord(v)=0;
   if v is not leaf then
        let v(1), ... , v(k) be the successors of v;
        Const-Node (v(1));
        for j = 2 to k d
            Const-Node1 (v(j),0);

Const-Node1 (v:node , i:integer)
   set val_ord (v) = i;
   if v is not leaf then
        let v(1), ... , v(k) be the successors of v;
        Const-Node2 (v(1),i);
        for j = 2 to k do
            Const-Node1(v(j),i+j-1);

Const-Node2 (v:node , i:integer)
   set val_ord (v) = i;
   if v is not leaf then
        let v(1), ... , v(k) be the successors of v;
        for j = 1 to k do
            Const-Node1 (v(j),i);
```

The order of the leaves is given in the following way:

Now we define the order of the leaves in the following way:

Let v and w are two leaves of the tree, then v is less than w if and only if $val_{ord}(v) < val_{ord}(v)$ or $(val_{ord}(v) = val_{ord}(w)$ and v is somewhere left of $w)$. Now we choose the leaves in increasing order.

As an example the values of the nodes are represented in the following tree.



For an algorithm with t processors we choose the first t leaves labelled with 0 in the order defined above. After that we have to calculate all possible node values. Now we cut off all leaves reachable from a node whose value is known. If the leftmost leaf of the tree in depths n (the first leaf who's game theoretical value is unknown) is labelled with 1 (only the values 0 and one 1 are possible for the leftmost one) then we update the labelling of the tree by starting the following procedure with the leftmost leaf as parameter v:

```
Const_Update(v:Node)
   Let w be the processor of $v$.
   if val_ord(w)=1 then Const_Update(w);
   else Const_Node1(v,0);
```

Remark 4 *By induction on the depth of the tree it is easy to prove that the leaves which are labelled with* 0 *in the initial tree have to be examined independent of the leaf values and independent of the ordering of the tree. If only these leaves have to be examined, we have a linear speedup with constant* 1. *If all leaves have to be examined we have also a linear speedup with constant* 1. *Therefore the main problem for the analysis is to find the worst case.*

Remark 5 *For the definition of the algorithm it is not necessary that the tree is uniform.*

References

[1] I. Althöfer,
On the complexity of searching game trees and other recursion trees,
J. Algorithms 9 (1988) 538–567.

[2] I. Althöfer, Data compression using an intelligent generator ... ,
Artificial Intelligence 52 (1991), 109–113.

[3] I. Althöfer and B. Balkenhol,
A game tree with distinct leaf values which is easy for the alpha–beta algorithm.
Artificial Intelligence 52 (1991), 183–190.

[4] B. Balkenhol, Diplom Thesis, University of Bielefeld,
Faculty of Mathematics (1992).

[5] A.Z. Broder, A.R. Karlin, P. Raghavan and E. Upfal,
On the parallel complexity of evaluating game trees,
Preprint (1990).

[6] J. Harting, Analyse eines parallelen Spielbaum–Suchverfahrens mit linearem Speed–up,
Diplomarbeit, Universität Bielefeld, Fakultät für Mathematik, 1991.

[7] F.–H. Hsu, Large scale parallelization of alpha–beta search:
an algorithmic and archirectural study with computer chess,
Ph.D. Thesis, Carnegie–Mellon University, Pittsburgh, PA (1990).

[8] R.M. Karp and Y. Zhang, On parallel evaluation of game trees,
in: Proceedings First ACM Symposium on Parallel Algorithms and Architectures (1989) 409–420.

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

[10] G. Lin and N. Pippenger, Parallel Evaluation of Game Trees,
Preprint, University of British Columbia, 1992.

[11] T.A. Marsland, Computer chess methods,
in: S.C. Shapiro, ed., Encyclopedia of Artificial Intelligence
((Wiley, New York, 1987) 159–171.

[12] J. Pearl, Heuristics:
Intelligent Search Strategies for Computer Problem Solving
(Addison–Wesley, Reading, MA, 1984).

[13] I. Roizen and J. Pearl,
A minimax algorithm better than alpha–beta ? Yes and No.
Artif. Intell. 21 (1983) 199–220.

[14] G. Schrüfer, Presence and absence of pathology on game trees,
in: D.F. Beal, ed., Advances in Computer Chess 4
(Pergamon, Oxford, 1986) 101–113.

[15] G. Stockman, A minimax algorithm better than alpha–beta ?
Artif. Intell. 12 (1979) 179–196.

[16] Y. Zhang, Parallel algorithms for combinatorial search problems,
Ph.D. Thesis, University of California, Berkeley, CA (1989).