THE KEY NODE METHOD:
A HIGHLY-PARALLEL ALPHA-BETA ALGORITHM[1]

by

Gary Lindstrom

UUCS 83-101

March 1983

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

ABSTRACT

A new parallel formulation of the alpha-beta algorithm for minimax game tree searching is presented. Its chief characteristic is incremental information sharing among subsearch processes in the form of "provisional" node value communication. Such "eager" communication can offer the double benefit of faster search focusing and enhanced parallelism. This effect is particularly advantageous in the prevalent case when static value correlation exists among adjacent nodes. A message-passing formulation of this idea, termed the "Key Node Method", is outlined. Preliminary experimental results for this method are reported, supporting its validity and potential for increased speedup.

[1] This work was supported in part by the National Science Foundation under grant MCS 78-03832.

1. The alpha-beta algorithm

1.1. Review

The alpha-beta strategy is a familiar method for economizing on the cost of minimax searching on game trees. Under this strategy, move generation at a node is "cut-off" or abandoned whenever it is determined from nearby node values that the node's ultimate value cannot possibly rise to the root of the tree. Such cuts may be "shallow" (due to a superior sibling), or "deep" (due to a superior ancestor sibling). Although the alpha-beta strategy has at times been called a heuristic, it is rather an optimization admitting no possibility of error in top-level minimax move selection.

A number of studies have estimated the savings obtained by the alpha-beta strategy under various conditions (e.g. [13]). In sum, these findings indicate that the alpha-beta strategy significantly slows (but does not eliminate) the exponential cost of searching to increasing game tree depths. In view of its ease of implementation in ordinary (i.e. recursive) depth-first searching, the method has seen wide application.

1.2. Interest here

With the prospect that large-scale physical parallelism will become increasingly common in computer systems of the future, considerable attention is being devoted to the problem of adapting sequential algorithms to multi-processor form. The alpha-beta method has served as a challenging case study in this regard, for a number of reasons:

- Parallel algorithms:

  * In sequential form, the method exploits a powerful optimization effect that is equally desirable in parallel versions. However, this effect does not directly generalize to parallel form, since it generally assumes a sequential left-to-right bottom-up node evaluation order.

  * A satisfactory parallel version must strike a balance between computational aggressiveness and caution, so that available physical parallelism is utilized, but in a "focused" manner (i.e. so that tasks likely to sharpen the search are favored to run early).

  * The alpha-beta method is broadly representative of an important class of problems in operations research, namely branch and bound problems.

- Parallel computer architectures:

  * An effective parallel version of the algorithm would necessarily exploit asynchronous information sharing among processes. This is the sine qua non of distributed control, captured in an extremely simple application setting.

  * This information sharing would involve unpredictable patterns of communication traffic, but within a known general process structure, i.e. a logical tree. This provides a concrete yet nontrivial framework for investigations of communication throughput on particular network topologies.

  * The alpha-beta method is computationally intensive and easily scalable to create any desired load for testing purposes.

2. Existing parallel approaches

Efforts to generalize the classical depth-first formulation of the alpha-beta method generally fall into two classes: logically parallel (i.e. coroutine oriented), and concurrent (i.e. exploiting true multi-processing).

2.1. Logically parallel approaches

This class may be viewed as a bridge between recursive and concurrent formulations, in that depth-first visitation order is broken through the use of "retentive" (i.e. non stack-based) sequential control. Two basic approaches of this kind are generally known:

1. The SSS* method of Stockman, which maps the alpha-beta method onto a state space search problem. A natural "best-first" node expansion order results [14]. The motivation is simply minimization of visited node count; however, the guaranteed attainment of this goal has recently been challenged [11].
2. The evolving tree search (ets) method of this author, which is a framework for exploiting cutting opportunities while doing node expansion in an arbitrary (e.g. heuristically-driven) order [9]. In particular, move generation from nodes can electively be suspended and resumed, with full maintenance of node cutting relationships (including move generation restarting when cutting values are weakened).

2.2. Concurrent approaches

Within the true multi-processing realm, three approaches have been proposed, each providing some measure of computational speedup[2]:

1. The earliest approach to alpha-beta concurrency appears to

[2] Mono-processing run time divided by multi-processing run time.

be the parallel aspiration search of Baudet [2]. In this method, the range of possible root node values is partitioned into "windows", which are assigned to individual processors as fictitious initial alpha-beta values. The processors then concurrently search the entire game tree by the classical recursive algorithm, but with the attention of each somewhat individually directed by its assigned "window."

By the nature of the alpha-beta method, each processor can independently report whether the root value falls within its window (or to which side of it). Selection of processor windows then becomes an adaptive search problem in its own right. After considerable analysis, Baudet concludes that this method offers speedup limited by a small constant (e.g. 5 or 6), independent of the degree of available physical concurrency.

2. The tree splitting approach of Finkel and Fishburn [3, 5] maps the game tree homomorphically onto a physical tree of processors. Leaf processors search by the classical algorithm. Asynchronous remote procedure calls are used to post updates of narrowing alpha-beta bounds as sibling values are reported. Deep cut-offs can occur. An earlier pseudo-functional formulation of this asynchronous window narrowing technique may be found in [7].

   Speedup on the order of at least the square root of the number of processors is claimed.

3. In the mandatory work first (mwf) approach of Akl, Barnard and Doran [1, 4], the game tree is initially hypothesized to be perfectly best-first ordered. The subset of the game tree which would be visited in this case under the classical method is then concurrently searched. If the best-first hypothesis is confirmed, the algorithm terminates. Otherwise, search is resumed (again using mwf) at those incompletely evaluated nodes whose initial results contradict their subordinate ranking. Deep cut-offs are unexploited in this method, but are known to be statistically insignificant in general.

   Although the simulation results presented in [1] seem to suggest that speedup levels off at a relatively low number of processors, subsequent analytical study in [4] indicates a much greater potential speedup.

A recent article by Marsland and Campbell [10] surveys these approaches and others in the slightly more general setting of "strongly ordered game trees."

3. The Key Node Method

We now present our new parallel alpha-beta method. As will become clear, it combines

- the focusing effect of *mwf*, with
- the node restarting capability of *ets*.

3.1. Overall strategy

The Key Node Method employs a very simple global strategy. The ingredients of this strategy are:

1. A notion of key node, i.e. those nodes which must obtain provably correct lower bounds on node strength.
2. A policy of eager value reporting from key nodes as their provisional minimax values change, and
3. A message passing control regime to communicate such value changes, as well as to propagate changes in key node status.

The result is a dynamically shifting *mwf* tree, as shaped by the accumulation of partial results from ongoing subsearches. Thus eagerness in the form of provisional value communication is exploited to channel the concurrent search to parallel paths in a changing, but eventually stabilizing, *mwf* tree.

3.2. Details

Given this search strategy, the details of the method are rather straightforward, involving a precise definition of key node, the contents of messages, the local data associated with each node, and algorithms for _up_ and _down_ message processing.

3.2.1. Key nodes

A _Key Node_ is defined as follows:

1. The _root_ of a minimax game tree is a _key node_, and is considered to be a _first descendant_ of a fictitious superroot node.
2. _All_ descendants of a _first descendant key node_ are _key nodes_.
3. Only the _first descendant_ of a _nonfirst descendant key node_ is a _key node_.

In terminology of [4],

1. key nodes are the union of type 1 and type 2 nodes, with
2. type 1 nodes being first descendant key nodes.

Figure 3-1 depicts this definition. The reader should bear in mind, however, that descendant ordering is dynamic best-first ordering, rather than static move generation order.

It is easy to prove that nonkey node values can be ignored in minimaxing without loss of root value correctness. Moreover, all first descendant key nodes have correct values, and the values at nonfirst descendant key nodes are valid lower bounds on their correct strength.

```
            /                      \                        \
           /                        \                        \
       are_key=T                are_key=T                are_key=F
      are_first=T              are_first=F              (don't care)
           /\                       /\                       /\
          /  \                     /  \                     /  \
         /....\                   /....\                   /....\
        /      \                 /      \                 /      \
       /        \               /        \               /        \
      /          \             /          \             /          \
  are_key=T   are_key=T    are_key=T   are_key=F     are_key=F  are_key=F
are_first=T are_first=F  are_first=T are_first=F
```

Figure 3-1: Definition of "key node."

3.2.2. Node contents

The local data associated with each game tree node is defined as a Pascal record type in figure 3-2.

```
    node =
        RECORD pos:     position;          {game position}
               parent:  ^node;             {parent node in tree}
               nrdesc:  descnr;            {number of descendants;
                                                     0 if terminal}
               desc:    ARRAY [1..desclim] OF   {desc. node records}
                                 RECORD  dname:  {desc. name}
                                            ^node;
                                         dvalue: {value last reported}
                                            nodeval
                                 END;
               amkey,            {is this node currently key?}
               amfirst,          {"   "    "        "   · a first desc.?}
               visited:          {has "    "   been visited yet?}
                        Boolean;
        END;
```

Figure 3-2: Game tree node data record.

For simplicity, we assume here that the entire game tree pre-exists in "latent" form, with value initializations for pos, onmove, parent, nrdesc, desc (dvalues = "worst" value reportable), and visited (false). In a more realistic formulation, of course, the node records would be created upon demand during tree search.

3.2.3. Message formats

As suggested in section 3.1, two types of messages are exchanged among our search processes. Messages flowing downward in the game tree convey node status changes, while messages flowing upward report new (possibly provisional) values. Figure 3-3 captures these requirements, again as a Pascal type definition.

```
    msg =       RECORD  dest:   ^node;
                        CASE msg_type:
                                (down, up) OF
                            down:       (are_key,
                                         are_first:    Boolean);
                            up:         (sender:       ^node;
                                         val:          nodeval);
                END;
```

Figure 3-3: Message formats.

3.2.4. Message processing logic

Operations for processing down and up messages are specified in figures 3-4 and 3-5, respectively. Note that search at a node is initiated by receipt of its first down message (which will necessarily convey are_key=T). While these operations assume FIFO message arrival order, this requirement is not essential, and will be addressed in section 5.2.

```
{install updates, temporarily saving old values}
amkey_current := amkey;         amkey := are_key;
amfirst_current := amfirst;     amfirst := are_first;

IF      nrdesc = 0
        THEN    {terminal node case}
        BEGIN   IF      {first down message to node}
                        NOT visited
                THEN    BEGIN   visited := true;
                                send_up_msg(
                                    {dest =}    parent,
                                    {sender =}  dest, {i.e. me}
                                    {val =}     statval(pos));
                        END
                ELSE    {ignore subsequent down messages}
        END
ELSE    {nonterminal node case}
        BEGIN   IF      {change in key status for 1st desc}
                        amkey_current <> are_key
                THEN    notify_descs(1, 1);
                IF      {change in key status for other descs}
                        (amkey_current AND amfirst_current) <>
                                (are_key AND are_first)
                THEN    notify_descs(2, nrdesc);
        END;


                        - - - - -


        PROCEDURE notify_descs(low, high: descnr);

        BEGIN   (* inform desc[low ... high] of new status *)
                FOR i := low TO high DO
                        send_down_msg(
                            {dest =}
                                desc[i].dname,
                            {are_key =}
                                amkey AND (amfirst OR (i=1)),
                            {are_first =}
                                i=1)
        END;    (* notify_descs *)
```

<u>Message contents</u>: are_first, are_key, dest;
<u>Node attributes</u>: amfirst, amkey, desc, nrdesc, parent, pos, visited;
<u>Temporary variables</u>: amkey_current, amfirst_current, i.

Figure 3-4: <u>Down</u> message processing logic.

```
{save current rank 1 descendant name & value}
v := desc[1].dvalue;  dfirst := desc[1].dname;

{find current rank of descendant now reporting}
k := find_desc(sender);

{install new value reported and re-sort descendants}
desc[k].dvalue := val;  sort_desc;

IF {rank 1 descendant has changed and node is key}
       (desc[1].dname <> dfirst) AND amkey
THEN   BEGIN   {inform two descendants involved of new ranking}
               send_down_msg({dest =}           desc[1]^.dname,
                             {are_key =}        amkey,
                             {are_first =}      true);
               send_down_msg({dest =}           dfirst,
                             {are_key =}        amkey AND amfirst,
                             {are_first =}      false);
       END;

IF {rank 1 value has changed}
       desc[1].dvalue<>v
THEN   send_up_msg({dest =}     parent,
                   {sender =}   dest,
                   {val =}      desc[1].dvalue);
```

- - - - -

<u>Message</u> <u>contents</u>: dest, sender, val;
<u>Node</u> <u>attributes</u>: amfirst, amkey, desc;
<u>Temporary</u> <u>variables</u>: dfirst, k, v.

Figure 3-5: <u>Up</u> message processing logic.

4. Performance assessment

4.1. Correctness and general observations

Given this sketch of the Key Node Method, we make the following general observations:

1. No matter how the mwf tree shifts, there is complete information transfer upward in the tree, and descendant value information is never discarded until superceded. Hence an intuitive termination argument can be made, based on monotonicity of information gathering.

2. Since the mwf is continually shifted as new values are reported upward, when the method terminates the correct root value must be indicated.

3. The time to process each message is at worst proportional to d, the number of descendants of a node. This is obvious for down messages; it is also true for up messages, despite the apparent log d additional factor implied by descendant re-ordering. In fact perfect re-ordering is not necessary; instead, only the new rank 1 and 2 descendants must be found when new values are installed. These can easily be determined in the same linear sweep used to associatively retrieve the reporting descendant's current rank.

4. For perfectly ordered trees, the Key Node Method visits exactly the same nodes as does the original mwf method. Like the mwf method, it fails to exploit deep cut-off opportunities.

4.2. Empirical studies

One may fairly question whether the Key Node Method simply exchanges node expansion costs for message processing costs. To address this issue, the method was implemented in a simulated multi-processing setting, and preliminary performance measurements were gathered.

4.2.1. Static value correlation

It is common practice for game playing programs to do move preordering, i.e. commencing node expansion by ranking all possible moves in terms of the static values associated with their immediately resulting game positions. The rationale for this practice is an implicit assumption of correlation between the static value at a node and its ultimate minimax value, however deep the search involved in that ultimate value's calculation. A more explicit reliance on this assumption is evident in iterative deepening [6]. Under this technique, a game tree is repeatedly searched to increasing ply depths, with the results at each cycle being used to length the "horizon" for move preordering at the next cycle. Interestingly enough, this phenomenon has largely been ignored in most analytical studies of alpha-beta performance, and may be responsible for recently argued fundamental flaws in the minimax approach as a whole [12].

In our judgment, this static value correlation effect is both prevalent and economically exploitable. In particular, this effect adds considerable credence to the utility of provisional value reporting as done in the Key Node Method. Indeed, the method might be viewed as a concurrent variation of iterative deepening (more on this in section 5.3).

On this belief, our empirical studies were conducted on a class of artificially generated game trees in which static values at each node were obtained by adding a uniformly drawn random number to the static value of its parent node (the root having a static value of zero). We defend this choice by observing that while catastrophic static value changes can occur (e.g. loss of a queen in chess), these are infrequent in comparison to small changes reflecting minor variation in static value components (e.g. relative board control and material balance). Moreover, we conjecture that this effect is particularly evident in actual play by skillful competitors (i.e. "principal variations").

4.2.2. Architectural setting

Seeking a general architectural model for our performance investigation, we assumed a fixed number of identical processors sharing a common memory and drawing messages in round robin order from a pooled queue. For timing, each message was assumed to be processed within unit delay. At the beginning of each time unit, a "ply" of k messages is removed from the front of the message queue and processed, where k equals the minimum of the number of messages waiting and the total number of processors.

Care was taken, however, to ensure that potential memory contention among the processors was accounted for at least crudely. This was accomplished by a simulated "locking" effect, whereby at most one message is processed at each distinct node within a message ply. The second and later messages destined for the same node within a ply are deferred to the next ply (with one or more processors consequently left idle for the current time unit).

4.2.3. Simulation results

The findings of our preliminary simulation studies are given in Tables 4-1 and 4-2. In each case, the columns report game tree degree (f), game tree depth (q), total node count, total terminal node count, number of nodes visited by the classical alpha-beta algorithm, number of terminal nodes visited by the classical alpha-beta algorithm, time required by a monoprocessor version of the Key Node Method (i.e. number of messages generated), number of nodes visited by the Key Node Method, number of terminal nodes visited by the Key Node Method, number of processors simulated, and observed speedup factor.

f	q	nodes total	terms total	nodes class.	terms class.	mono. time	nodes KNM	terms KNM	nr. procs.	speed up
3	5	364	243	101.50	52.40	347.70	124.60	67.60	2	1.83
							126.60	68.60	5	4.06
							124.60	67.60	10	7.11
							124.60	67.60	20	10.65
3	6	1093	729	175.20	80.50	881.60	272.40	146.20	2	1.84
							272.40	146.20	5	4.22
							277.00	148.60	10	7.61
							277.00	148.80	20	13.06
4	4	341	256	74.90	39.50	214.80	87.80	52.00	2	1.67
							87.80	52.00	5	3.66
							89.60	53.20	10	6.25
							89.60	53.20	20	9.42
4	5	1365	1024	194.90	110.10	711.10	256.50	153.10	2	1.73
							256.50	153.10	5	3.93
							256.50	153.10	10	7.11
							258.60	154.60	20	12.57

Table 4-1: Experimental measurements: with move preordering.

In each case ten trials were averaged to smooth the numbers obtained. Random static value increments for each descendant (see section 4.2.1)

f	q	nodes total	terms total	nodes class.	terms class.	mono. time	nodes KNM	terms KNM	nr. procs.	speed up
3	5	364	243	152.20	86.00	1284.90	227.00	137.60	2	1.88
							227.00	137.40	5	4.36
							223.80	136.20	10	8.17
							222.00	134.60	20	14.66
3	6	1093	729	358.50	196.40	3989.10	615.40	369.80	2	1.90
							612.40	368.00	5	4.47
							611.00	366.80	10	8.51
							609.00	365.60	20	16.04
4	4	341	256	140.60	88.60	1064.80	209.60	142.90	2	1.83
							210.20	143.20	5	4.19
							208.10	141.70	10	7.76
							200.60	136.60	20	13.29
4	5	1365	1024	384.10	247.30	4072.60	695.70	470.80	2	1.85
							701.70	474.10	5	4.33
							699.00	471.70	10	8.16
							701.10	473.50	20	15.45

Table 4-2: Experimental measurements: without move preordering.

were integers drawn from [-1000, 999). All final root node values were checked for minimax correctness.

We make the following observations on the results obtained.

1. Significant speedup is obtained, despite the simulated memory contention and the generous number of processors (20) in the largest case tested.

2. Speedup increases appear comparable for both increased tree breadth (f) and increased tree depth (q).

3. Finally, the apparently greater speedup of the method on the non move preordering case is interesting but must be discounted as being unrealistic. For example, in the f=4 q=5 case, the 20-processor average run time is 4072.6/15.45 = 263.6, under nonpreordering, while the corresponding average run time under preordering is 711.10/12.57 = 56.6. Given the low cost of move preordering, the creater speedup in the former case must therefore be considered illusory.

5. Optimizations and extensions

Clearly, further testing of this method on minimax trees from "real" game players must be made before any firm conclusions on its merit can be made. Beyond this, several other areas of continued development are suggested by the results obtained thus far.

5.1. Buffered nonkey node values

A minor optimization can be obtained by suppressing <u>up</u> messages from nonkey nodes.[3] Such messages occur when a node has been downgraded to nonkey status, but the downgrading of nodes in its descendant tree has not yet occurred due to message latency. This optimization gives a simple priority effect to are_key=F messages, without assuming direct architectural support for message priority.

5.2. Removal of message FIFO assumption

In general, FIFO message processing is physically difficult to achieve in real multi-processing architectures without serious concurrency obstruction. Hence we ultimately wish to remove this assumption from the Key Node Method.

Fortunately, non-FIFO message order can easily be accommodated; indeed, in a certain sense, the phenomenon can be salutary. The problem is of course one of message overtaking, whereby "old" messages are received after "new" ones. Due to the eager information premise of

[3] The simulation results summarized in tables 4-1 and 4-2 in fact reflect this optimization.

the Key Node Method, receiving "new" messages "early" is generally beneficial, as long as "old" messages are recognized as such and are ignored. This discrimination is easily achieved through the following simple device. Each node marks all sent messages with a "time" stamp (a local serial number suffices), and records the latest time stamp of messages received from its parent and each of its descendants. Out of order messages are then easily detected and ignored.

5.3. Iterative deepening

As mentioned in section 4.2.1, the Key Node Method can be viewed as a concurrent variation of the iterative deepening strategy used in sequential game players. This effect can be amplified through the augmentation of messages to convey search depth information. That is, down messages could specify a "search to" depth, and up messages could specify a "valid to" depth.

The key node logic can easily be extended to accommodate this extra information as follows.

- Down messages with "search to" depth greater than 1 are propagated with depth decrementation.

- Up messages are generated in the following manner.

  * Upon receipt of a "search to 1" down message, a node behaves as a terminal node, responding with its static value "valid to 1."

  * When an up message is sent from a nonterminal node n, the "valid to" level is computed in a manner generalizing the value reporting logic of fig. 3-5. That is, we extend the desc element records of the node datatype (see fig. 3-2) to include a field validto, recording the "valid to" level last reported by that

descendant (initially 0, reflecting no validity). The <u>up</u> message sent by n is marked "valid to" a level computed by the following case analysis:

1. n.amfirst=F:

   - 0, if n.desc[1].validto=0;

   - k+1, if n.desc[1].validto=k>0.

2. n.amfirst=T:

   - 0, if the minimum of n.desc[i].validto is 0, for 1<=i<=n.nrdesc;

   - k+1, if the minimum of n.desc[i].validto is k>0, for 1<=i<=n.nrdesc.

In addition to providing the customary benefits of iterative deepening (better time management in tournament situations, and fewer node expansions, due to the "high terminal" effect discussed in [9]), two other advantages accrue here:

- A simple local test for root value finality is obtained, and
- Redundant downward messages (to nodes already "valid to" the desired depth) can be suppressed.

5.4. Distributed implementations

Given its message-passing basis, the Key Node Method is naturally well-suited to distributed computing systems. However, as discussed in section 4.2.2 the simulation results reported in Tables 4-1 and 4-2 were obtained for a centralized memory model, in which each processor has equal access to the data representing each node.

In a truely distributed architecture, the memory (and node records) would be partitioned among the processors, forming processing element (PE) pairs. Two possibilities would then exist for PE message processing:

1. Messages could be routed to the PE containing the destination node record, or

2. Messages could be processed by any PE, at the cost of extra message traffic (node attribute read/writes) to and from the PE possessing the node record.

The first method resembles distributed database systems, while the latter resembles applicative multi-processing architectures such as AMPS [8]. Further study of the Key Node Method and related approaches under both these regimes is clearly warranted.

REFERENCES

[1] S. G. Akl, D. T. Bernard, R. J. Doran.
Simulation and analysis in deriving time and storage requirements for a parallel alpha-beta algorithm.
In Proc. 1980 Int'l. Conf. on Par. Proc., pages 231-234. IEEE, August, 1980.

[2] Gerard Baudet.
The design and analysis of algorithms for asynchronous multiprocessors.
PhD thesis, Dept. of Computer Science, Carnegie-Mellon Univ., April, 1978.
Report CMU-CS-78-116.

[3] Raphael A. Finkel and John P. Fishburn.
Parallelism in alpha-beta search.
Artificial Intelligence 18, 1982.

[4] Raphael A. Finkel and John P. Fishburn.
Improved speedup bounds for parallel alpha-beta search.
Undated.

[5] J. P. Fishburn, R. A. Finkel and Sharon A. Lawless.
Parallel alpha-beta search on Arachne.
In Proc. Int'l. Conf. on Parallel Proc., pages 235-243. IEEE Computer Society, 1980.

[6] James J. Gillogly.
Performance analysis of the Technology Chess Program.
Technical Report CMU-CS-78-189, Dept. of Computer Science, Carnegie-Mellon Univ., March, 1978.

[7] R.M. Keller.
An approach to determinacy proofs.
Technical Report UUCS-78-102, University of Utah, Dept. of Computer Science, March, 1978.

[8] R.M. Keller, G. Lindstrom, and S. Patil.
A loosely-coupled applicative multi-processing system.
In AFIPS, pages 613-622. AFIPS, June, 1979.

[9] Gary Lindstrom.
Alpha-beta pruning on evolving game trees.
March, 1979.
Univ. of Utah Tech. Rpt. UUCS 79-101.

[10] T. A. Marsland and M. Campbell.
Parallel search on strongly ordered game trees.
Computing Surveys 14(4):533-551, 1982.

[11] Robert J. McGlinn.
Is SSS* better than alpha-beta?
1982.
Unpublished report, Dept. of Computer Science, Southern Illinois Univ., Carbondale.

[12] D. S. Nau.
The last player theorem.
Artificial Intelligence 18:53-65, 1982.

[13] Judea Pearl.
The solution of the branching factor of the alpha-beta pruning algorithm.
Technical Report UCLA-ENG-CSL-8019, University Of California, Los Angeles, Cognitive Systems Lab., April, 1980.

[14] G. C. Stockman.
A minimax algorithm better than alpha-beta?
Artificial Intelligence 12:179-196, 1979.

Table of Contents

List of Figures

List of Tables