Single-Agent and Game-Tree Search †

T.A. Marsland

Computing Science Department
University of Alberta
Edmonton
Canada T6G 2H1

E-Mail: tony@CS.UAlberta.CA
Fax: (403)-492-1071

Technical Report
TR91.16
August 1991

† Draft for the Encyclopedia of Computer Science and Technology

Single-Agent and Game-Tree Search

T.A. Marsland

Computing Science Department
University of Alberta
Edmonton
Canada T6G 2H1

E-Mail: tony@CS.UAlberta.CA
Fax: (403)-492-1071

1. Introduction

Problem solving by exhaustive enumeration is a common computational technique that often relies on a decision tree framework to ensure that all combinations are considered. This approach is helped by a wealth of powerful tools for supporting tree searches. A related but slightly more general model is based on a state-space approach in which, from a given state of the system and a set of actions (that is, given a description vector), the successor states are expanded until a specified goal is reached. Selecting an action transforms one state of a system into another, where perhaps a different set of actions is possible. A variety of general methods may be posed this way, for example, to find a sequence of actions that convert a system from an original to a final state with a pre-specified set of properties (in other words to seek a goal), or to find all such sets of actions (exhaustive enumeration), or to find the fewest actions needed to reach a goal (find a minimal cost solution), and so on.

Because these state transition problems can be described by graphs, which in turn are supported by a substantial mathematical theory, efficient methods for solving graph-based problems are constantly sought. However, many of the most direct classical methods for finding optimal solutions, e.g., dynamic programming, have a common fundamental failing: they cannot handle large problems (whose solution requires many transitions), because they must maintain an exponentially increasing number of partially expanded states (nodes) as the search front grows. Since storage space for intermediate results is often a more serious limitation than inadequate computing speed, heuristics and algorithms that trade space for time have practical advantages, rendering solutions that are otherwise unattainable.

To illustrate these points, and to provide insights into widely applicable and generally useful techniques that can be used to improve many optimization methods, we will consider the subdomains of single agent (one-person) and adversary (two-person) games. In both cases solutions can be found by traversing a decision tree that spans all the possible states in the “game.” Since the order in which the decisions are made is not necessarily important, it is common for identical states to exist at different places in the decision tree. Under these circumstances such trees might be viewed as graphs. Because a tree is an intrinsically simpler structure, as well as being more regular than a graph, we will temporarily ignore such duplications, but later we will introduce methods that explicitly recognize and eliminate duplicates, and so reduce the effective size of the search space.

2. Single Agent Search

As an example of a single agent search problem consider the popular N-puzzle game, which typified by N distinct tiles on a rectangular grid plus a single ‘‘empty tile’’ space. The object of the game is to slide the tiles until all are in specified positions (a goal state). Humans can be adept at this problem, even when N is large, but solve it without regard to optimality (least tile movement). For computers a simple optimal algorithm exists, one which is general and can be applied to a wide variety of state-space search applications. Called A* [Nilsson, 1971], it is guaranteed to find an optimal solution, but because of its high memory requirements it can handle only small problems (e.gg., 3x4 puzzle or smaller). A more recent variation, Iterative Deepening A* (IDA*) [Korf, 1985] draws effectively on the notion of successive refinement and uses an interesting technique that can be generally incorporated in tree searches. As we show later the iterative deepening idea has been around for more than two decades in the computer chess community, where it is highly refined and enjoys great popularity. In IDA* the iterative technique controls elegantly the growth of memory needed in the expansion of a one-person game tree, but in such a way that an optimal solution is still guaranteed.

The essence of A* is the use of a heuristic evaluation function to guide the search by ordering successor states according to estimated cost of the path (set of transitions) from the start to the goal state. This is possible by using an evaluation function of the form:

$$f(n) = g(n) + h(n),$$

where g(n) is the measured cost of the path from the start state (say node 0) to the current state, node n, and h(n) is an estimate of the cost of the path from n to the goal state. If h(n) never overestimates the remaining cost, A* is guaranteed to find an optimal (least cost) solution. The properties of g(n) and h(n) are easily seen from the simple N-puzzle example. Here g(n) is exactly equal to the number of tile movements taken so far to convert the start state to the current state. Further, if h(n) measures the sum of the Manhattan distances (that is, the sum of the vertical and horizontal displacements of each tile from its current square to its goal state square), then it never overestimates the number of tile movements required. It is comforting to have an algorithms guaranteeing an optimal solution but, as with most state-space search methods, even an almost perfect evaluation function excessively produces partially expanded states. By analogy with a technique pioneered in computer chess programs to keep the time cost of search within reasonable bounds, Korf developed a simple mechanism to control a single agent search based on the A* evaluation function, and so find an optimal solution by ensuring that no solution of lesser cost exists. Korf’s [1985] iterative deepening version of A* eliminates the need to maintain open/closed lists of node state vectors, and has linear space complexity.

2.1. Iterative Deepening

Iterative Deepening A* is interesting and, as Korf shows, is more powerful than A*, in that it can find optimal solutions to some bigger problems, since its memory management costs are negligible, and space requirements are linear with depth. Even so IDA* is not universally successful, it can behave especially poorly on the traveling salesman problem. Again, illustrating with the N-puzzle, if node 0 represents the start state, a lower bound on the cost from the start position is:

$$c = f(0) = h(0),$$

since g(0) is zero, so at least c tile movements are needed. Thus during the first iteration solutions of length c are sought. As soon as the condition

$$g(n) + h(n) > c$$

holds, the search from node n is discontinued. In problems of this type g(n) increases monotonically, so that unless h(n) decreases by an amount equal to g's increase the search stops quickly. Thus during each iteration a minimal expansion is done. If the goal state is not found, the depth of search is increased to the smallest value of g(n) + h(n) attained in the previous iteration (always an increase of 2 for the N-puzzle), and the next iteration is started. The last iteration is usually the most expensive, especially if all the minimal cost solutions are sought. Even though more nodes may be expanded than for A*, the simplicity of the method makes IDA* the more useful algorithm in practice, because it eliminates costly mechanisms for maintaining the ordered list of states that remain to be expanded.

3. Min-Max Search

So far we have considered how expansion of a game tree can be controlled by an evaluation function, and how the major shortcomings (excessive memory requirement) of a best-first state-space search can be overcome with a simple iterative depth-first search.

Iterative deepening is a powerful and general method whose effect can be further improved if used with some other refinements. These advantages can be seen better through the study of methods for searching two-person game trees, which represent a struggle between two opponents who move alternately. Because one side's gain (typically position or material in board games) usually reflects an equivalent loss for the opponent; these problems are often modeled by an exhaustive minimax search, so called because the first player is trying to maximize the gains while the second player (the hostile opponent) is minimizing them. In a few uninteresting cases the complete game tree is small enough that it can be traversed and every terminal (tip or leaf) node examined to determine precisely the value for the first player. The results from the leaf nodes are fed back to the root using the following back-up procedure. Given an evaluation function f(n) which can measure the value of any leaf node from the first player's view:
For a leaf node n

$$\text{MinMax}(n) = f(n) = \text{Evaluate}(n)$$

For any interior node, n, with successor nodes n_i

$$\text{MinMax}(n) = \underset{i}{Max}\ (-\text{MinMax}\ (n_i)).$$

Note that this formulation, referred to by Knuth and Moore [1975] as *Negamax*, replaces the opponent's minimization function by an equivalent maximization of the negation of the successor values, thus achieving a more symmetric definition. Here Evaluate(n) is a function that computes the merit value of a leaf node, n, from the root node's viewpoint. For a true leaf (no successors) the merit value will be thought of as exact or accurate, and without error. Building exhaustive minimax enumeration trees for difficult games like chess and Go is impractical, since they would contain about 10^{40} or 10^{100} nodes, respectively. Evaluate(n) can also be used at pseudo-leaf (frontier or horizon) nodes, where it computes a value that estimates the merit of the best successor. Again the value will be designated as true or accurate, even though it is only an estimate (in some more sophisticated search methods an attempt is made to account for the uncertainly in the leaf values). Under the negamax backing up rule, the sequence of branches from the root to the best pseudo leaf node is referred to as the *Principal Variation*, and the merit value of the leaf node at the end of the path is the one that is backed up to the root and becomes the value of the tree.

3.1. Fail-Soft Alpha-Beta

One early paper on computer chess [Newell, Shaw and Simon, 1958] recognized that a full minimax search was not essential to determine the value of the tree. Some years later a little known work by Brudno [1963] provided a theoretical basis for pruning in minimax search. From all these observations, the alpha-beta pruning algorithm, was developed, and it remains today the mainstay for game-tree search. Of course many improvements and enhancements have been added over the years, and some of these will be explored here. An important point about the alpha-beta algorithm is that it is a simple branch and bound method, where the bound for all Max nodes (including the root) is named Alpha, and the bound for the Min nodes is named Beta. In effect, search can be viewed as taking place within a window or range of integer values Alpha to Beta with the underlying assumption that the integer value, V, of the tree lies in that range (that is, Alpha < V and V < Beta). Clearly if the initial values of Alpha and Beta are $-\infty$ and $+\infty$, respectively, the merit value of the tree will fall within that infinite range. In contrast, one popular enhancement to the alpha-beta algorithm, called *aspiration search*, artificially narrows these bounds, hoping to reduce the search space by cutting out more of the tree, and gambling that the true merit will still be found. To be most effective, aspiration search should include Fishburn's *fail-soft* idea [Fishburn, 1981]. The important theoretical point behind that idea is presented in Figure 1, which shows pseudo code for `ABSearch` (a basic version of the alpha-beta algorithm). In Figure 1, the integer parameters `Alpha` and `Beta` represent lower and upper bounds, and `Height` is the remaining distance (in ply) to the search frontier. Also, `Position[0]` represents a pointer to the current node (state), and `Position[N]` is a pointer to the N^{th} successor of the current node. ABSearch returns the merit value of the subtree by using a recursive backing up process. One important feature of this skeleton is that the maximum depth of search is limited to Height from the root. Clearly this depth parameter can be manipulated to allow selective extensions of the search depth.

```
function ABSearch (Position, Alpha, Beta, Height);
  if Height ≡ 0
    then return (Evaluate(Position[0]));          {frontier node}
  N = SelectNextNode (Position[0]);              {get first move}
  if N ≡ null
    then return (Evaluate(Position[0]);     {leaf, no successors}
  Best = −∞;
  while N ≠ null do begin
    Merit = − ABSearch (Position[N], −Beta, −Max(Alpha,Best), Height−1);
    if Merit > Best then begin
      Best = Merit;                              {improved value}
      if Best ≥ Beta then return (Best);                {cut-off}
    end;
    N = SelectNextNode (Position[N]);             {get next move}
  end while;
  return (Best);                       {return the subtree value}
end ABSearch;
```

Figure 1: Fail-soft alpha-beta algorithm

3.2. Aspiration Search

The essence of the fail-soft approach is the initialization of `Best` to $-\infty$ instead of Alpha, as seems natural. Thus, even if the initial bounds (Alpha, Beta) are too narrow, after the search is complete we will know whether V ≤ Best ≤ Alpha, or whether Beta ≤ Best ≤ V. That is, not only determine whether the search failed low or high, but also provide an upper/lower bound of the tree's true value. In the case of failure, the tree must be re-searched with correct bounds of either $-\infty$ to Best, or Best to $+\infty$, as appropriate. So far we have considered only a basic minimax pruning process incorporating a fail-soft mechanism, so that the best available merit value is returned when inappropriate Alpha-Beta bounds are chosen. Inclusion of this feature makes possible a useful variation, called aspiration search or narrow window search, which initially restricts the Alpha-Beta bounds to a narrow range around an expected value, V_0, for the tree. Thus at the root node an aspiration search might be invoked by setting Alpha = $V_0 - \varepsilon$ and Beta = $V_0 + \varepsilon$ and

```
    Best = ABSearch (Position, Alpha, Beta, Height)
```

As a result, if the condition Alpha < Best < Beta is not met a re-search is necessary, as detailed in the following code:

```
if Best ≥ Beta then
    Best = ABSearch (Position, Best, +∞, Height);
if Best ≤ Alpha then
    Best = ABSearch (Position, -∞, Best, Height);
```

The advantages of working with narrow bounds can be significant, especially for games where it is easy to estimate V_0. However, there is ample experimental evidence [Marsland, 1983; Musczycka and Shinghal, 1985; Kaindl, 1990] to show that use of heuristics to estimate the search window in Aspiration Search still does not usually yield a performance comparable to the *Principal Variation Search (PVS)* algorithm [Marsland and Campbell, 1982]. The main disadvantage of Aspiration Search is that the estimate of V_0

is made strictly before the search begins, while for PVS the value of V_0 is continually refined during the search. Thus PVS benefits more from application-dependent knowledge that provides a good move ordering, and so almost guarantees that the value of the first leaf will be a good estimator of the tree's merit value. Nevertheless a problem remains: no matter how narrow the initial bounds, nor how good the move ordering, the size of the minimal game tree still grows exponentially with depth.

3.3. Approximating Game Trees

In practice, because game trees are so large, one must search a series of approximating subtrees of length `Height`, as the code in Figure 1 shows. Thus, instead of true leaf nodes, where the value of the node is known exactly, we have pseudo-leaf nodes or frontier nodes where the value of the unexplored subtree beyond this horizon is estimated by the evaluation function. In the simplest case the approximating tree has a pre-specified fixed depth, so that all the frontier nodes are at the same distance from the root. This model is satisfactory for analytical and simulation studies of searching performance, but it does not reflect the current state of progress in application domains. For example, a typical chess program builds its approximating tree with three distinct phases. From the root all moves are considered up to some fixed depth, d (usually a constant), but if a node has only one or two legal successors—e.g. after a checking move in chess—it is not counted towards the depth, so the effective length of some paths could be d + d/2 (since in practice only one side at a time administers a series of checks). Once the nominal depth of the first phase is reached, a second phase extends the search by another constant amount (again forced nodes are not counted in this extension), but at every new node only a selection of the available moves is considered. This heuristic is the dangerous and discredited practice of forward pruning. It works here because the exhaustive search layer finds short term losses that lead to long term gains (obvious sacrifices), while the next stage uses forward pruning to eliminate immediately losing moves and seemingly inferior short term continuations, thus reducing the demands on the third (quiescence search) phase. Although not *ad hoc*, this approach is ill-defined (although clearly some programmers have superior methods), but as we shall see it leads naturally to several good possibilities for a probabilistic way of controlling the width of the search.

The third (quiescent) phase of search is more dynamic. It is called a quiescence search, because its purpose is to improve the evaluation estimate of critical frontier nodes involving dynamic terms that cannot be measured accurately by the static evaluation function. In chess these terms include captures, checks, pins and promotions. It is essential that these quiescence trees be severely restricted in width, only containing moves that deal with the non-quiescent elements. There have been several studies of desirable properties of quiescence search, but most noteworthy is the work of Kaindl [1983, 1989], the method of singular extensions by Anantharaman *et al.* [1988], and the formalization of the null-move heuristic [Beal, 1989].

In summary, the three layer search employs algorithmic backward pruning which is at first exhaustive, then uses limited forward pruning of seemingly obvious losing moves, and finally a highly directed selective search. Thus the use of heuristics increases with the depth of search, thereby introducing more uncertainty but extending the depth (frontier/horizon) along lines of greatest instability, thereby clarifying the outcome. This

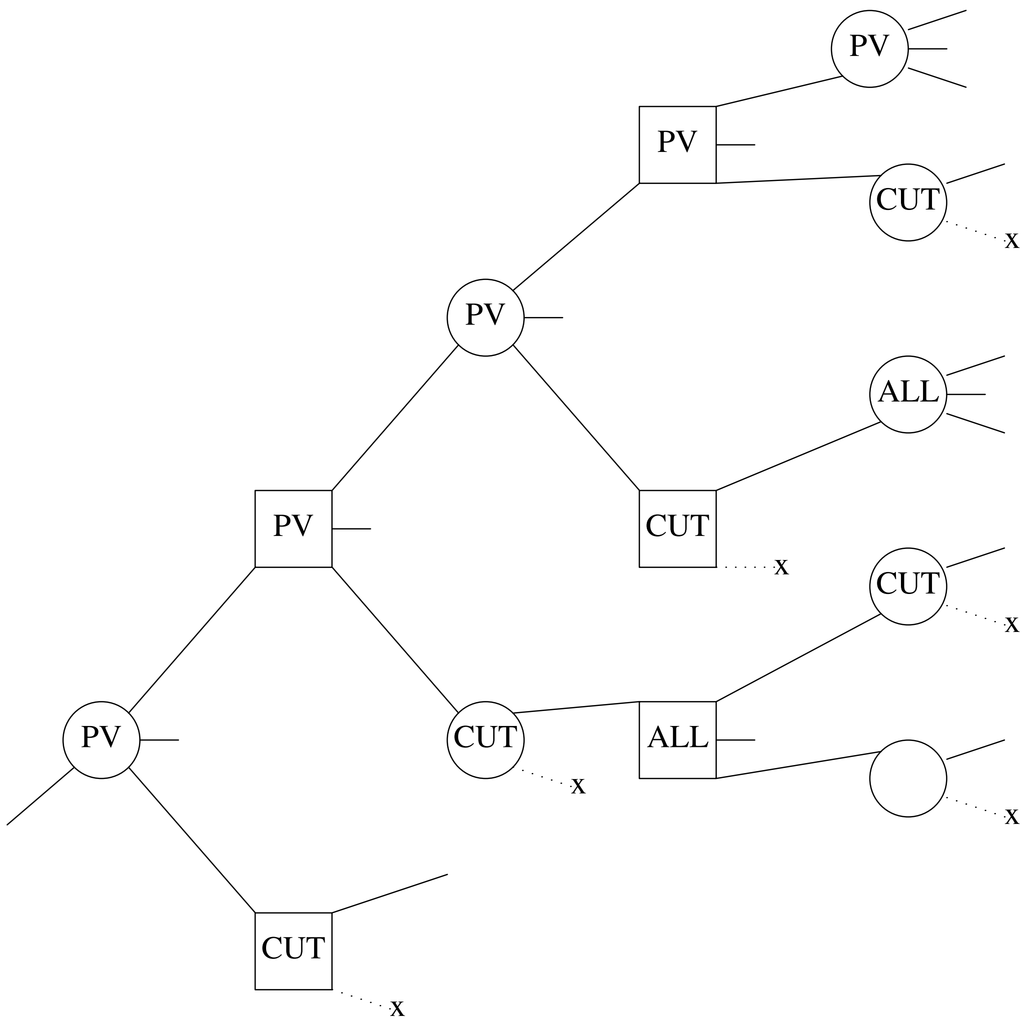

Figure 2: Structure of a minimal game tree

approach has many practical advantages and can be used equally effectively in many decision tree applications.

There is no theoretical model for these variable depth search processes. Analytical studies usually restrict themselves to the use of uniform trees (trees with exactly W successors at each node and fixed depth, D). The most commonly quoted result is the figure for the size of the minimal (optimal) game tree, which has

$$W^{\left\lceil \frac{D}{2} \right\rceil} + W^{\left\lfloor \frac{D}{2} \right\rfloor} - 1$$

leaf nodes. In Knuth and Moore’s [1975] terminology, the minimal game tree is made up of type 1, type 2 and type 3 nodes. Marsland and Popowich [1985] call these PV, CUT and ALL nodes to make it clearer where cut-offs may occur, as Figure 2 shows.

3.4. Principal Variation Search

An important reason for considering fail-soft alpha-beta is that it leads naturally to more efficient implementations, specifically Principal Variation Search (PVS), which in turn uses a *Null Window Search* (NWS). The fundamental idea here is that as soon as a better move (and bound) is found, an attempt is made to prove that the remaining alternatives are inferior. A null window is used so that no integer value can fall between these two adjacent bounds. Thus all remaining searches with that window will fail, hopefully low, proving the inferiority of the move. If the null window search fails high, then the move is superior to the previously best and the search will have to be repeated with the correct bounds, to find the proper path and value, as Figure 3 shows. Figure 4, on the other hand, illustrates exactly how the bounds are set and how the tree's merit value is backed up in a small example.

```
function PVS (Position, Alpha, Beta, Height);
   if Height ≡ 0 then return (Evaluate(Position[0]);
   N = SelectNextNode (Position[0]);
   if N ≡ null then return (Evaluate(Position[0]));
   Best = − PVS (Position[N], −Beta, −Alpha, Height−1);
   while SelectNextNode(Position[N]) ≠ null do
      if Best ≥ Beta then return (Best);              {CUT node}
      N = SelectNextNode (Position[N]);
      Alpha = Max(Alpha, Best);
      Merit = − NWS (Position[N], −Alpha, Height−1);
      if (Merit > Best) then
        if (Merit ≤ Alpha) or (Merit ≥ Beta)
          then Best = Merit
          else Best = − PVS (Position[N], −Beta, −Merit, Height−1);
   end;
   return (Best);                                      {PV node}
end PVS;

function NWS (Position, Beta, Height);
   if Height ≡ 0 then return (Evaluate(Position[0]);
   N = SelectNextNode (Position[0]);
   if N ≡ null then return (Evaluate(Position[0]));
   Best = −∞;
   while N ≠ null do
      Merit = − NWS (Position[N], −Beta+1, Height−1);
      if Merit > Best then Best = Merit;
      if Best ≥ Beta then return (Best);              {CUT node}
      N = SelectNextNode (Position[N]);
   end;
   return (Best);                                     {ALL node}
end NWS;
```

Figure 3: Principal variation (null window) search

Thus the fundamental reason for the form of Figure 3 is now clear, it reflects the the structure of a game tree in that at PV nodes an alpha-beta search (PVS) is used, while

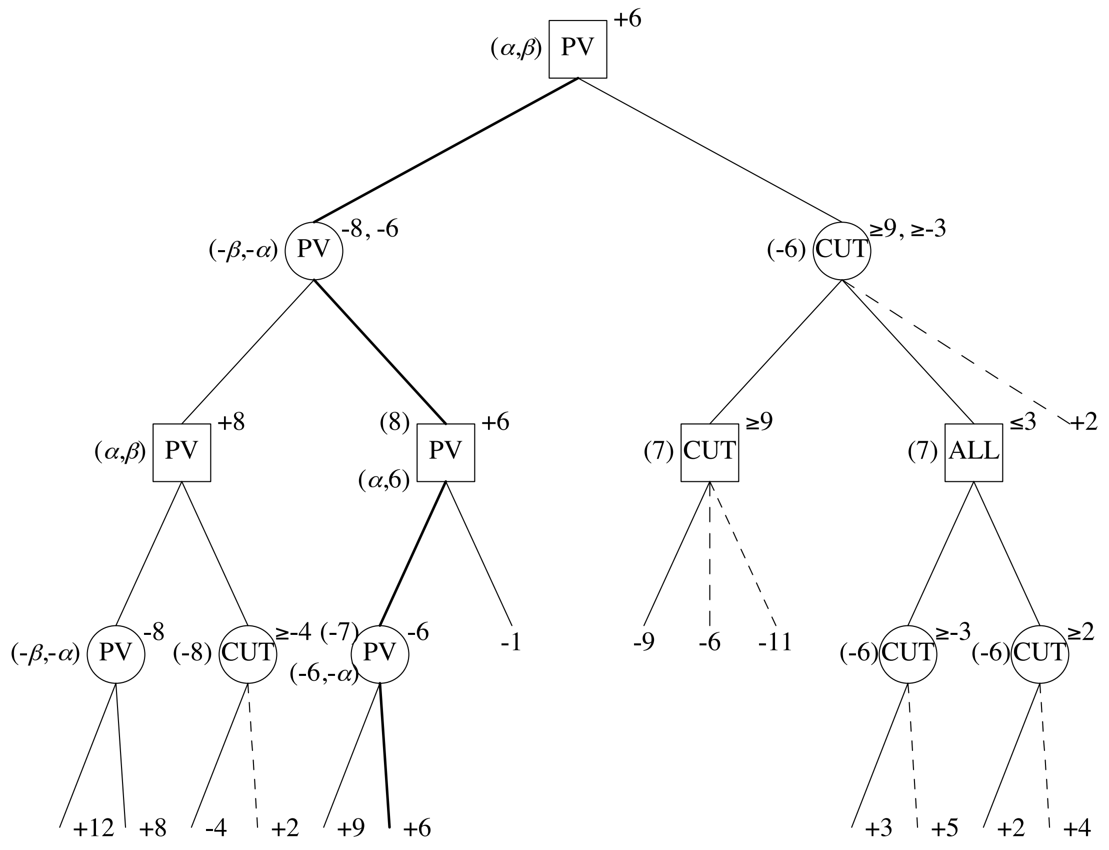

Figure 4: Sample pruning of a minimax tree by PVS/NWS

CUT and ALL nodes are initially visited by NWS.

Note that use of NWS is not essential, since in the PVS code of Figure 3 the line

```
        Merit = -NWS (Position[N], -Alpha, Height-1)
```

can be replaced by

```
    Merit = -PVS (Position[N], -Alpha-1, -Alpha, Height-1)
```

to produce a compact fully recursive alternative. This more compact implementation also illustrates better the notion of a null window. It encapsulates everything into one routine and is precisely the approach taken in an early description [Marsland, 1983] and in NegaScout [Reinefeld, 1983]. The use of NWS serves two purposes: first it makes possible a direct comparison with Scout [Pearl, 1980] and also, as we shall see later, this separation helps the design of parallel game-tree search algorithms. Although Scout is a depth-first search it does not seem to be used in practice, perhaps, as Kaindl [1990] points out, because it does not gain from the benefit that fail-soft alpha-beta provides.

Figure 3 explains how a "fail high" null-window search at ALL and CUT nodes is converted into a PVS re-search of a PV node. It also shows that the status of a node changes to PV whenever its value increases, so that it is re-searched by PVS. Reinefeld and Marsland [1987] built on this model and developed some results for an *average* game tree, based of the notion of a re-search rate, and developed the theoretical conditions under which PVS is better than pure alpha-beta.

3.5. NegaScout and Scout

Fully recursive versions of PVS have been produced [Marsland, 1983], but particularly interesting is Reinefeld's [1983] NegaScout model, which Kaindl [1990] shows to be a more efficient implementation of Scout [Pearl, 1980]. NegaScout introduced an admissible (without error) pruning technique near the frontier, in contrast to the more speculative *razoring* method of Kent and Birmingham [1977], and the notion of a *futility cutoff*, best described by Schaeffer [1986]. The essential idea behind razoring is that at the last move before the frontier the side to move will usually be able to improve the position, and hence the value of the node. In effect we assume that there is always at least one move that is better than simply passing, i.e., not making a move. Therefore if the current node merit value already exceeds the Beta bound, a cut-off is inevitable and the current node cannot be on the Principal Variation. This heuristic is widely applicable, but it is prone to serious failure. For example, in chess, where passing is not allowed, razoring will fail in zugzwang situations, since every move there causes the value for the moving player to deteriorate. More commonly, when the pieces are already on ‘‘optimal squares’’ most moves will appear to lead to inferior positions. This is especially true when the side to move has a piece newly under attack. The futility cutoff, on the other hand, is a little safer. Again at the layer before the frontier, if the current node value is less than Alpha, only moves that have the potential to raise the node value above Alpha are of interest. This will include appropriate captures and all checking moves. It may be futile to consider the rest unless the current node value is close to Alpha. Abramson [1989] provides an accessible review of razoring and other control strategies for two-player games.

3.6. Other Search Methods

As IDA* has shown, depth first searches have modest storage needs and can benefit from iterative deepening. For the two-person games there are several best-first searches, but they all suffer from the same excessive demands on memory and heavy overhead in maintenance of supporting data structures. Nevertheless, the state space searches are interesting on theoretical grounds and contain ideas that carry over into other domains. For example, Berliner's [1979] best first B* algorithm returns a two-part evaluation range with pessimistic and optimistic bounds. Since the real aim is often to find the best choice or move (with only secondary interest in the expected value), B* uses its bounds to identify that move. The best move is the one whose pessimistic value is at least equal to the largest optimistic value of all the alternatives. Note that it is not necessary to search intensely enough to reduce the range intervals to a single point, just enough to find the best move, thus some search reduction is theoretically possible. Later Palay [1983] developed an algorithm called PB* to introduce probability distributions into the evaluation function.

SSS*, a best-first algorithm developed by Stockman [1979], is also of special interest. Closely related to A*, SSS* dominates the alpha-beta algorithm in the sense that it never visits more leaf nodes. Also, with the change proposed by Campbell [1981] to alter the order in which the tree is traversed, SSS* expands a subset of the nodes visited by a normal alpha-beta search, e.g., ABSearch (Figure 1). But more efficient depth-first search algorithms, like PVS, exist and they too dominate ABSearch. Statistically, most

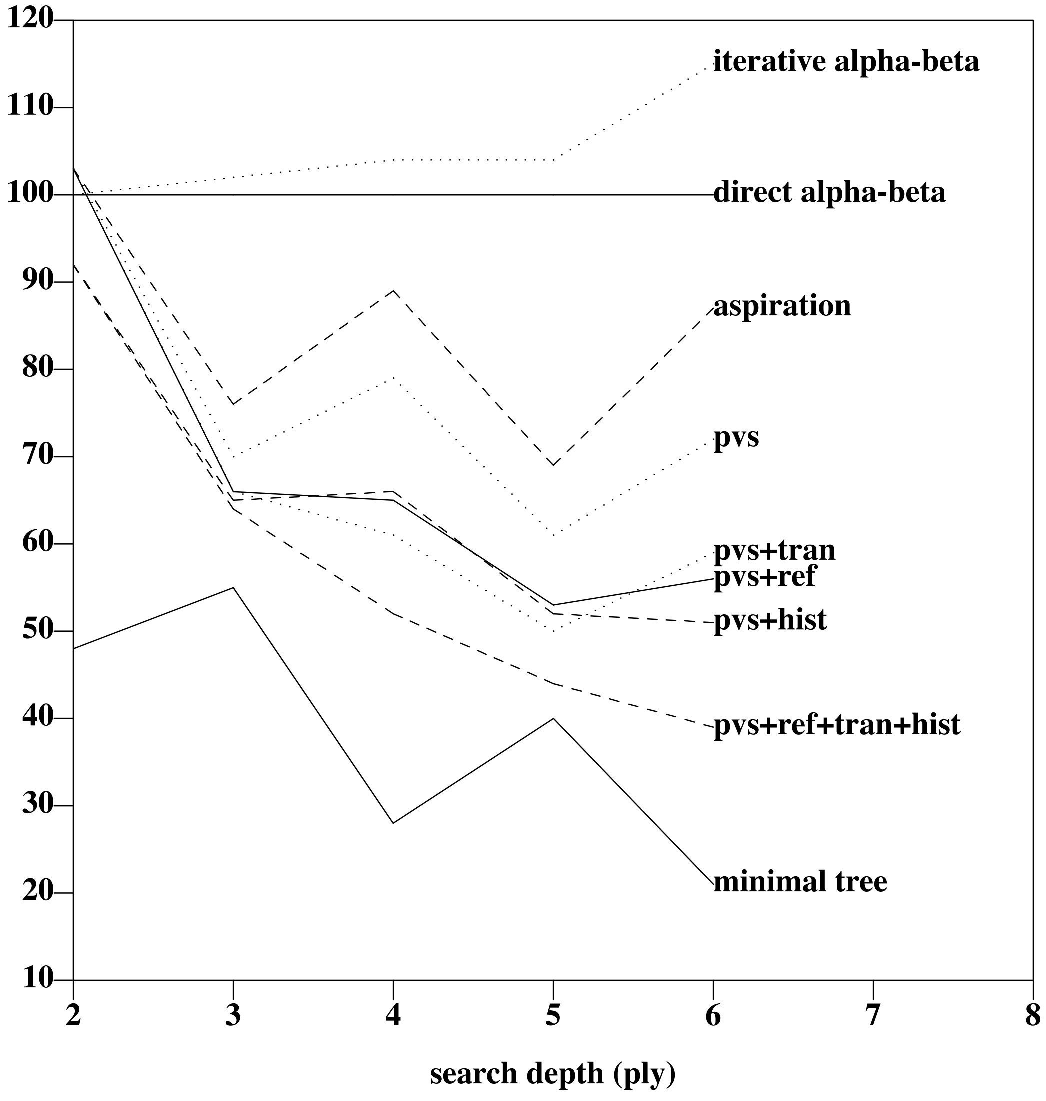

Figure 5: Search of strongly ordered uniform trees (D = 5 and W = 20)

efficient of all is a variation of SSS*, named DUAL* by Marsland *et al.* [1987], which is formed by complementing the actions at the min and max nodes. The duality has the effect of doing a directed (left to right) search at the root node and SSS* below that. Thus DUAL* has lower memory requirement (since it uses SSS* to search a 1-ply shallower tree), but otherwise shares the same burdensome overheads. Although Reinefeld [1989, Table 3.1, page 102] has established the dominance over a normal alpha-beta search on theoretical grounds, the statistical performance of these algorithms varies widely. In particular, SSS* does not do well on bushy trees (average width > 20) of odd depth, as Figure 5 illustrates for strongly ordered trees [Marsland & Campbell, 1982]. Such trees are intended to have properties similar to the game trees that arise during a typical application like chess, yet permit a better comparison than is possible with random data. SSS* does not perform well here because the trees used were not random, rather the best move was searched first more than 60% of the time. DUAL* is best because of the directional search at the root. However, both SSS* and DUAL* share A*'s problem, namely that the CPU overhead to maintain the active states is more than five times that required for a depth-first search [Marsland *et al.*, 1987]. Thus, lower leaf node counts for SSS* and DUAL* do not normally translate into lower CPU utilization, quite the contrary. The idea for DUAL* came from a study of parallel SSS* algorithms [Kumar and Kanal, 1984] and opens a whole new world of game tree search studies.

Of the other new techniques, McAllister's [1988] so called conspiracy number search is especially interesting. Although this method also makes heavy demands on computer memory, it is one of the class of probabilistic algorithms that attempt to measure the stability of search. A tree value is more secure (unlikely to change) if several nodes would have to ''conspire'' (all be in error) to change the root value. Application of this method is still in its infancy, although Schaeffer [1990] has provided some working experiences and Allis *et al.* [1991] make a comparison between SSS*, alpha-beta, and conspiracy number search for random trees. Since many game-tree applications require the search of bushy trees (e.g., chess and Go) some form of probabilistic basis for controlling the width of search would be of great importance

3.7. Memory Functions for Iterative Deepening

The main problem with direct searches to pre-specified minimal depth is that they provide inadequate control over the CPU needs. Since CPU control can be important in human-computer play, an iterative deepening method was introduced by Scott [1969]. In its early form, rather than embark on a search to depth N-ply (and not knowing how long it might take), a succession of searches of length 1-ply, 2-ply, 3-ply etc. were used until the allotted time is reached. The best move found during one iteration is used as the first move for the start of the next and so on. Over the following years this idea was refined and elaborated, notably by Slate and Atkin [1977], until by the late 1970s several memory functions were in use to improve the efficiency of successive iterations. It is this increased efficiency that allows an iterative deepening search to pay for itself and, with memory function help, to be faster than a direct D-ply search. The simplest enhancement is the use of a *refutation table*, as presented by Akl and Newborn [1977]. Here, during each iteration, a skeletal set of paths from the root to the limiting frontier is maintained. One of those paths is the best found so far, and is called the Principal Variation (or Principal Continuation). The other paths simply show one way for the opponent to refute them, that is, to show they are inferior. As part of each iteration these paths are used to start the main alternatives, with the intention of again proving their inferiority. The overhead for the refutation table is best described in a new book by Levy and Newborn [1990].

3.8. Transposition Table Management

More general than the refutation table is the *transposition table*, which in its simplest form is a large hash table for storing the results from searches of nodes visited so far. The results stored consist of: (a) the best available choice from the node, (b) the backed up value (merit) of the subtree from that node, (c) whether that value is a bound, (d) the length of the subtree upon which the value is based. As with all hash tables, a key/lock entry is also required to confirm that the entry corresponds to the node being searched. The space needed for the key/lock field depends on the size of the hash table, but 48 bits is common. Problems with entry conflict error were initially dealt with by Zobrist [1970] when he proposed a hashing method for Go. Much later, the application to computer chess was reviewed [Marsland, 1986], with further insights by Nelson [1985] and by Warnock and Wendroff [1989]. The importance of the transposition table is two-fold. Like a refutation table, it can be used to guide the next iteration, but being bigger it also contains information about the refutations (killer moves) in subtrees that are not part of the main continuation. Perhaps of greater importance is the benefit of information sharing during an iteration. Consider the case when an entry corresponding to the current subtree is found in the transposition table. If the depth field entry is not less than the remaining depth of search, it is possible to use the merit value stored in the entry as the value of the subtree from the node. This circumstance arises often, since transposition of moves is common in many two-person games. As a result, use of a transposition table reduces the effective size of the tree being searched; in extreme cases not only enabling a search of less than the minimal game tree, but also extending the search of some variations to almost double the frontier distance. More common, however, is use of the ‘‘move’’ from the transposition table. Typically that move was put there during a null window search, having caused a cut off, and is re-used to guide the research down the refutation line.

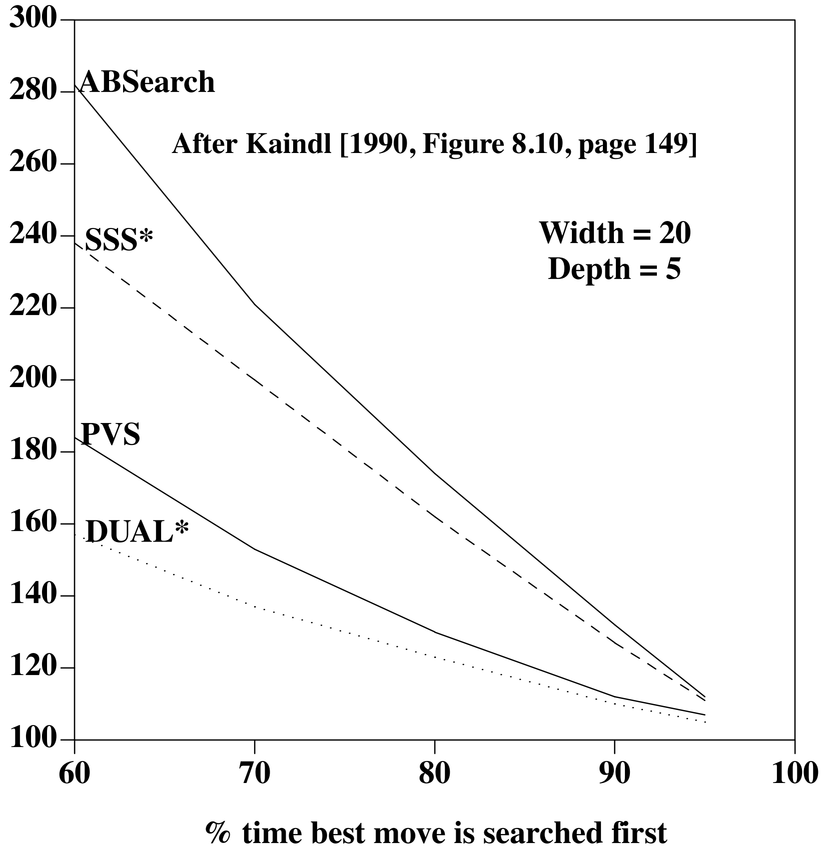

Figure 6: Leaf node comparison of alpha-beta enhancements

Another memory function is the *history heuristic table*. This is a general method for identifying “killer moves,” that is choices that have cut-off the search at other places in the tree [Schaeffer, 1983]. The method is especially suited to cases where the choices (moves) at any node are drawn from a fixed set. For instance, without regard to the pieces, all moves on a chess board can be mapped into a 64x64 table (or even two tables, one for each player). Stored in that table would be a measure of how effective each move had been in causing cut-offs. Schaeffer found that simply using the frequency of prior pruning success is a more powerful means of ordering moves, than using application dependent heuristic knowledge. Move ordering in turn dramatically improves the efficiency of directional searches like `ABSearch` and `PVS`.

3.9. Combined Enhancements

The relative efficiencies of these various alpha-beta enhancements are adequately captured in Figure 6, which presents data from a model chess program (Parabelle) searching a suite of test positions. A direct N-ply alpha-beta search is taken as the 100% basis for comparison, based on frontier nodes visited. Figure 6 shows that under reasonable assumptions PVS is more efficient than Aspiration Search (although optimal aspiration windows will necessarily do better). Further the memory function assists of transposition table (`+trans`), refutation table (`+ref`) and history table (`+hist`) for re-ordering the moves are additive and make a significant improvement in performance. The worsening result for the 6-ply search by PVS with transposition table (`pvs+trans`) may be attributed to overloading of the small (only 8K entries) transposition table. For comparison, a lower bound is provided by estimating the size of the minimal uniform game tree that approximates the average size of the trees that were generated during the search of the test suite (for Figure 6, the average width of each node in the tree traversed was 34 branches). The oscillatory nature of these graphs can be attributed to the higher fraction of frontier nodes that must be evaluated in odd-depth trees.

4. Parallel Game-Tree Search

In recent years the increased availability of small low-cost computers has led to an upsurge of interest in parallel methods for traversing trees and graphs. In the game-tree case, experience has been gained with a variety of practical implementations. Although many of the theoretical studies in parallelism focused on a dataflow model, by and large that model could not account for pragmatic factors like communication and synchronization delays that inevitably plague physical systems.

The main problems faced by the designers of parallel tree-search systems are

(a). How best to exploit the additional resources (e.g. memory and i/o capability) that may become available with the extra processors.

(b). How to distribute the work across the available processors.

(c). How to avoid excessive duplication of computation.

Some simple combinatorial problems have no difficulty with point (c) and so, if work distribution is not a problem, ideal or even anomalously good speedup is possible [Lai and Sahni, 1984].

In game-tree search the necessary information communicated is the improving estimates of the tree value. But, since uniprocessor solutions strive to minimize the size of the tree traversed by maximizing the pruning efficiency, parallel systems face the problem of unpredictable size of the subtrees searched (e.g., pruning may produce an unbalanced workload) leading to potentially heavy synchronization (waiting for more work) losses. The standard basis for comparison is speedup, defined by

$$speedup \;=\; \frac{time\ taken\ (or\ nodes\ visited)\ by\ a\ uni-processor}{time\ taken\ (or\ nodes\ visited)\ by\ an\ N-processor\ system}$$

Although simple, this speedup measure can often be misleading, because it is dependent on the efficiency of the uniprocessor implementation. Also use of node counts does not help measure the communication and synchronization overheads. Thus good speedup may merely reflect a comparison with an inefficient uniprocessor design. On the other hand, poor speedup clearly identifies an ineffective parallel system.

4.1. Single Agent Search

IDA* [Korf, 1985] has proved to be an effective depth-first method for single agent games. Not surprisingly it has also been a popular algorithm to parallelize. Rao *et al.* [1987] proposed PIDA*, an almost linear algorithm whose speedup is about 0.93N when N processors are used, even when solving the 15-puzzle with its trivial node expansion cost. It was thought to be even more efficient on the Traveling Salesman Problem, which entails a more expensive node generation process, but there the really efficient branch and bound algorithms are sequential (but are not amenable to parallelization), so comparisons often overstate the success of the parallel methods. Powley and Korf [1989] propose a parallel window search for IDA*, which ‘‘can be used to find a nearly optimal solution quickly, improve the solution until it is optimal, and then finally guarantee optimality, depending on the amount of time available.’’ At the same time Huang and Davis [1989] proposed a distributed heuristic search algorithms (PIA*) which they compare to A*. On a uniprocessor, PIA* expands the same nodes as A*. Although they claim that ‘‘this algorithm can achieve almost linear speedup on a large number of processors’’ [Huang and Davis, 1989], it has the disadvantage that its memory requirements are the same as for A*, and therefore is of doubtful practical value.

4.2. Adversary Games

In the area of two-person games, early simulation studies with a *Mandatory Work First* (MWF) scheme [Akl *et al.*, 1982], and the PVSplit algorithm [Marsland and Campbell, 1982], showed that a high degree of parallelism was possible, despite the work imbalance introduced by pruning. Those papers recognized that in many applications, especially chess, the game-trees tend to be well ordered because of the wealth of move ordering heuristics that have been developed [Slate and Atkin, 1977; Gillogly, 1972]; thus the bulk of the computation occurs during the search of the first subtree. The MWF approach recognizes that there is a minimal tree that must be searched. Since that tree is well-defined and has regular properties, it is easy to generate and search. Also, nodes where all successors must be considered can be searched in parallel, albeit with reduced benefit from improving bounds. The balance of the tree can be generated algorithmically and searched quickly through simple tree splitting. Fishburn and Finkel [1982] also favored this method and provided some analysis. The first subtree of the minimal game tree has the same properties as the whole tree, but its maximum height is one less. This so called principal variation can be recursively split into parts of about equal size for parallel exploration. PVSplit, an algorithm based on this observation, was proposed [Campbell, 1981] and simulated [Marsland and Campbell, 1982]. Independently Monroe Newborn built the first parallel chess program, and later presented performance results [Newborn, 1985] [Newborn, 1988]. For practical reasons the tree was only split down to some pre-specified common depth from the root (typically 2), where the greatest benefits from parallelism can be achieved. This use of a common depth has been taken up by Hsu [1990] in his proposal for large-scale parallelism. Limiting depths are also an important part of changing search modes and in managing transposition tables.

4.3. Advanced Tree-splitting Methods

Results from fully recursive versions of PVSplit were presented for the Parabelle chess program [Marsland and Popowich, 1985]. These results confirmed the earlier simulation results and offered some insight into a major problem: In this N-processor system, however, it was common for all but one of the processors to be idle for an inordinate amount of time. This led to the development of variations that dynamically assign processors to the search of the principal variation. Notable is the work of Schaeffer [1989], which uses a loosely coupled network of workstations, and Hyatt *et al.*'s [1989] implementation for a shared-memory computer. That dynamic splitting work has attracted growing attention with a variety of approaches. For example, the results of Feldmann *et al.* [1990] show a speedup of 11.8 with 16 processors (far exceeding the performance of earlier systems) and Felten and Otto [1988] measured a 101 speedup on a 256 processor hypercube. This latter achievement is noteworthy because it shows an effective way to exploit the 256 times bigger memory that was not available to the uniprocessor. Use of the extra transposition table memory to hold results of search by other processors provides a significant benefit to the hypercube system, thus identifying clearly one advantage of systems with an extensible address space.

These results show a wide variation not only of methods but also of apparent performance. Part of the improvement is accounted for by the change from a static assignment of processors to the tree search (e.g. PVSplit), to the dynamic processor re-allocation schemes of Hyatt *et al.* [1989], and also Schaeffer [1989]. These later systems tried to identify dynamically the ALL nodes of Figure 3 (where every successor must be searched) in the game tree, and search them in parallel, leaving the CUT nodes (where only a few successors might be examined [Marsland and Popowich, 1985]) for serial expansion. The MWF approach first recognized the importance of dividing work at ALL nodes and did this by a parallel search of the minimal game tree. In a similar vein Ferguson and Korf [1988] proposed a "bound-and-branch" method that only assigned processors to the left-most child of the tree-splitting nodes where no bound (subtree value) exists. Their method is equivalent to the static PVSplit algorithm, and yet realizes a speedup of 12 with 32 processors for Othello-based alpha-beta trees! More recently

Steinberg and Solomon [1990] also addressed this issue with their *ER* algorithm, and also considered the performance of different processor tree architectures. Their 10-fold speedup with 16 processors was obtained through the search of 10-ply trees generated by an Othello program. They did not consider the effects of iterative deepening, nor exploit the benefits of transposition tables. As with similar studies, the fundamental flaw with speedup figures is their reliance on a comparison to a particular (but not necessarily best) uniprocessor solution. If that solution is inefficient the speedup figure will look good (for example, by omitting the important node-ordering mechanisms). For that reason comparisons with a standard test suite from a widely accepted game is often done and should be encouraged. Most of the working experience with parallel methods for two-person games has centered on the alpha-beta algorithm. Parallel methods for more node-count efficient sequential methods, like SSS*, have not been successful [Vornberger and Monien, 1987], although the use of hashing methods to replace linked lists has not been fully exploited.

4.4. Recent Developments

Although there have been several successful implementations involving parallel computing systems [Guiliano *et al*., 1990], significantly better methods for NP-hard problems like game-tree search remain elusive. Theoretical studies often concentrate on showing that linear speedup is possible on worst order game trees. While not wrong, they make only the trivial point that where exhaustive search is necessary, and where pruning is impossible, then even simple work distribution methods yield excellent results. The true challenge, however, is to consider average game trees, or the strongly ordered model, where extensive pruning occurs, leading to unsymmetric trees and a significant work distribution problem.

For the game-tree case, many people consider the minimal or optimal tree model, in which the best successor is considered first at every node. Although idealistic, this model presumes the search of a highly structured tree, one which the iterative deepening searches approximate. Akl *et al*. [1982] considered the search of minimal trees in their simulation of the Mandatory Work First method. Intuitively this is a nice idea, and yet it has not led to practical methods for the search of game trees. In practice average trees differ significantly from minimal trees, and so the underlying assumption behind MWF is undermined. Thus static processor allocation schemes like MWF and PVSplit cannot achieve high levels of parallelism, although PVSplit does very well with up to 4 processors. MWF in particular ignored the true shape of the minimal game tree under optimal pruning, and so was better with shallow game trees, where the pruning imbalance from the so called "deep cutoffs" has less effect.

Many people have recognized the intrinsic difficulty of searching game trees under pruning conditions, and one way or another try to recognize dynamically when the minimal game tree assumption is being violated, and hence to re-deploy the processors. Powley *et al*. [1990] presented a distributed tree search scheme, which has been effective for Othello. Similarly Feldmann *et al*. [1990] introduced the concept of making ‘‘young brothers wait’’ to reduce search overhead. Both of these systems have yielded impressive speedup results, but they may be overstated. The first system used a hypercube, so that the N-fold increase in processors was accompanied by an N-fold increase in memory not available to the uniprocessor. The second system used slow (8088) processors, so that the

I/O time may have been significant in the uniprocessor case.

Generalized depth-first searches [Korf, 1989] are fundamental to many AI problems, and Kumar and Rao [1990] have fully explored a method that is well-suited to doing the early iterations of IDA*. The unexplored part of the trees are marked and are dynamically assigned to any idle processor. In principle this method could be used for deterministic game trees too. Finally we come to the issue of scalability and the application of massive parallelism. None of the work discussed so far for game tree search seems to be extensible to arbitrarily many processors. Nevertheless there have been claims for better methods and some insights into the extra hardware that may be necessary to do the job. Perhaps most complete is Hsu's recent thesis [1990]. His project for the re-design of the Deep Thought chess program is to manufacture a new VLSI processor in large quantity. The original machine had 2 or 4 processors, but two new prototypes with 8 and 24 processors have been built as a testing vehicle for a 1000 processor system. That design was the major contribution of the thesis [Hsu, 1990], and with it Hsu predicts, on the basis of some simulation studies, a 350-fold speedup. No doubt there will be many inefficiencies to correct before that comes to pass, but in time we will know if massive parallelism will solve our game-tree search problems.

References

[Abr89] B. Abramson, ‘‘Control Strategies for Two-Player Games,’’ *ACM Computing Surveys* **21**(2), 137-162 (1989).

[AkN77] S. G. Akl and M. M. Newborn, "The Principal Continuation and the Killer Heuristic," *1977 ACM Ann. Conf. Procs.*, (New York: ACM), Seattle, Oct. 1977, 466-473.

[ABD82] S. G. Akl, D. T. Barnard and R. J. Doran, ‘‘Design, Analysis and Implementation of a Parallel Tree Search Machine,’’ *IEEE Trans. on Pattern Anal. and Mach. Intell.* **4**(2), 192-203 (1982).

[AMH91] L. V. Allis, M. Meulen and H. J. Herik, ‘‘$\alpha\beta$ Conspiracy Number Search’’ in D.F. Beal (ed.), *Advances in Computer Chess 6*, Ellis Horwood, 1991, 73-95.

[ACH88] T. Anantharaman, M. Campbell and F. Hsu, ‘‘Singular Extensions: Adding Selectivity to Brute-Force Searching,’’ *Int. Computer Chess Assoc. J.* **11**(4), 135-143 (1988). Also in *Artificial Intelligence* **43**(1), 99-110 (1990).

[Bea89] D. Beal, ‘‘Experiments with the Null Move’’ in D. Beal (ed.), *Advances in Computer Chess 5*, Elsevier, 1989, 65-79. Revised as ‘‘A Generalized Quiescence Search Algorithm’’ in *Artificial Intelligence* **43**(1), 85-98 (1990).

[Ber79] H. J. Berliner, ‘‘The B* Tree Search Algorithm: A Best First Proof Procedure,’’ *Artificial Intelligence* **12**(1), 23-40 (1979).

[BiK77] J. A. Birmingham and P. Kent, ‘‘Tree-searching and Tree-pruning Techniques’’ in M. Clarke (ed.), *Advances in Computer Chess 1*, Edinburgh University Press, Edinburgh, 1977, 89-107.

[Bru63] A. L. Brudno, ‘‘Bounds and Valuations for Abridging the Search of Estimates,’’ *Problems of Cybernetics* **10**, 225-241 (1963). Translation of Russian original in *Problemy Kibernetiki* **10**, 141-150 (May 1963).

[Cam81] M. S. Campbell, Algorithms for the Parallel Search of Game Trees, Tech. Rep. 81-9, Computing Science Dept., University of Alberta, Edmonton, Canada, August 1981.

[FMM90] R. Feldmann, B. Monien, P. Mysliwietz and O. Vornberger, ‘‘Distributed Game Tree Search’’ in V. Kumar, P.S. Gopalakrishnan and L. Kanal (eds.), *Parallel Algorithms for Machine Intelligence and Vision*, Springer-Verlag, New York, 1990, 66-101.

[FeO88] E. W. Felten and S. W. Otto, "A Highly Parallel Chess Program," *Procs. Int. Conf. on 5th Generation Computer Systems*, (Tokyo: ICOT), Nov. 1988, 1001-1009.

[FeK88] C. Ferguson and R. E. Korf, "Distributed Tree Search and its Application to Alpha-Beta Pruning," *Proc. 7th Nat. Conf. on Art. Intell. (vol 1)*, (Los Altos: Kaufmann), Saint Paul, Aug. 1988, 128-132.

[Fis84] J. P. Fishburn, Analysis of Speedup in Distributed Algorithms, UMI Research Press, Ann Arbor, Michigan, 1984. See earlier PhD thesis (May 1981) Comp. Sci. Tech. Rep. 431, University of Wisconsin, Madison, 118pp.

[Gil72] J. J. Gillogly, ‘‘The Technology Chess Program,’’ *Artificial Intelligence* **3**(1-4), 145-163 (1972). Also in D. Levy (ed.), *Computer Chess Compendium*, Springer-Verlag, 1988, 67-79.

[GKM90] M. E. Guiliano, M. Kohli, J. Minker and I. Durand, ‘‘PRISM: A Testbed for Parallel Control’’ in V. Kumar, P.S. Gopalakrishnan and L. Kanal (eds.), *Parallel Algorithms for Machine Intelligence and Vision*, Springer-Verlag, New York, 1990, 182-231.

[Hsu90] F. Hsu, Large Scale Parallelization of Alpha-Beta Search: An Algorithmic and Architectural Study with Computer Chess, CMU, Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, Feb. 1990.

[HuD89] S. Huang and L. R. Davis, "Parallel Iterative A* Search: An Admissible Distributed Search Algorithm," *Procs. 11th Int. Joint Conf. on AI (vol 1)*, (Los Altos: Kaufmann), Detroit, 1989, 23-29.

[HSN89] R. M. Hyatt, B. W. Suter and H. L. Nelson, ‘‘A Parallel Alpha/Beta Tree Searching Algorithm,’’ *Parallel Computing* **10**(3), 299-308 (1989).

[Kai83] H. Kaindl, "Searching to Variable Depth in Computer Chess," *Procs. 8th Int. Joint Conf. on Art. Intell.*, (Los Altos: Kaufmann), Karlsruhe, Germany, Aug. 1983, 760-762.

[Kai89] H. Kaindl, Problemlosen durch Heuristische Suche in der Artificial Intelligence, Springer-Verlag, Vienna, 1989.

[Kai90] H. Kaindl, "Tree Searching Algorithms" in T.A. Marsland and J. Schaeffer (eds.), *Computers, Chess, and Cognition*, Springer-Verlag, New York, 1990, 133-158.

[KnM75] D. E. Knuth and R. W. Moore, "An Analysis of Alpha-beta Pruning," *Artificial Intelligence* **6**(4), 293-326 (1975).

[Kor85] R. E. Korf, "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search," *Artificial Intelligence* **27**(1), 97-109 (1985).

[Kor89] R. E. Korf, "Generalized Game Trees," *Procs. 11th Int. Joint Conf. on AI (vol 1)*, (Los Altos: Kaufmann), Detroit, 1989, 328-333.

[KuK84] V. Kumar and L. Kanal, "Parallel Branch and Bound Formulations for AND/OR Tree Search," *IEEE Trans. on Pattern Anal. and Mach. Intell.* **6**(6), 768-778 (1984).

[KuR90] V. Kumar and V. N. Rao, "Scalable Parallel Formulations of Depth-First Search" in V. Kumar, P.S. Gopalakrishnan and L. Kanal (eds.), *Parallel Algorithms for Machine Intelligence and Vision*, Springer-Verlag, New York, 1990, 1-41.

[LaS84] T. Lai and S. Sahni, "Anomalies in Parallel Branch-and-Bound Algorithms," *Comm. ACM* **27**, 594-602 (1984).

[LeN90] D. N. L. Levy and M. M. Newborn, How Computers Play Chess, W.H. Freeman & Co., New York, 1990.

[MaC82] T. A. Marsland and M. Campbell, "Parallel Search of Strongly Ordered Game Trees," *Computing Surveys* **14**(4), 533-551 (1982).

[Mar83] T. A. Marsland, "Relative Efficiency of Alpha-beta Implementations," *Procs. 8th Int. Joint Conf. on Art. Intell.*, (Los Altos: Kaufmann), Karlsruhe, Germany, Aug. 1983, 763-766.

[MaP85] T. A. Marsland and F. Popowich, "Parallel Game-Tree Search," *IEEE Trans. on Pattern Anal. and Mach. Intell.* **7**(4), 442-452 (July 1985).

[Mar86] T. A. Marsland, "A Review of Game-tree Pruning," *Int. Computer Chess Assoc. J.* **9**(1), 3-19 (1986).

[MRS87] T. A. Marsland, A. Reinefeld and J. Schaeffer, "Low Overhead Alternatives to SSS*," *Artificial Intelligence* **31**(2), 185-199 (1987).

[McA88] D. McAllister, "Conspiracy Numbers for Min-Max Search," *Artificial Intelligence* **35**(3), 287-310 (1988).

[MuS85] A. Musczycka and R. Shinghal, "An Empirical Study of Pruning Strategies in Game Trees," *IEEE Trans on Systems, Man and Cybernetics* **15**(3), 389-399 (1985).

[Nel85] H. L. Nelson, "Hash Tables in Cray Blitz," *Int. Computer Chess Assoc. J.* **8**(1), 3-13 (1985).

[New85] M. M. Newborn, "A Parallel Search Chess Program," *Procs. ACM Ann. Conf.*, (New York: ACM), Denver, Oct 1985, 272-277. See also (March 1982) Tech. Rep. SOCS 82.3, Computer Science, McGill University, Montreal, Canada, 20pp.

[New88] M. M. Newborn, "Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search," *IEEE Trans. on Pattern Anal. and Mach. Intell.* **10**(5), 687-694 (1988).

[NSS58] A. Newell, J. C. Shaw and H. A. Simon, "Chess Playing Programs and the Problem of Complexity," *IBM J. of Research and Development* **4**(2), 320-335 (1958). Also in E. Feigenbaum and J. Feldman (eds.), *Computers and Thought*, 1963, 39-70.

[Nil71] N. Nilsson, Problem Solving in Artificial Intelligence, McGraw-Hill, 1971.

[Pal85] A. J. Palay, Searching with Probabilities, Pitman, 1985. See earlier Ph.D. Thesis (1983), Computer Science, Carnegie-Mellon University, Pittsburgh, 152pp.

[Pea80] J. Pearl, "Asymptotic Properties of Minimax Trees and Game Searching Procedures," *Artificial Intelligence* **14**(2), 113-138 (1980).

[PoK89] C. Powley and R. E. Korf, "Single-Agent Parallel Window Search: A Summary of Results," *Procs. 11th Int. Joint Conf. on AI (vol 1)*, (Los Altos: Kaufmann), Detroit, 1989, 36-41.

[PFK90] C. Powley, C. Ferguson and R. E. Korf, "Parallel Heuristic Search: Two Approaches" in V. Kumar, P.S. Gopalakrishnan and L. Kanal (eds.), *Parallel Algorithms for Machine Intelligence and Vision*, Springer-Verlag, New York, 1990, 42-65.

[RKR87] V. N. Rao, V. Kumar and K. Ramesh, "A Parallel Implementation of Iterative-Deepening A*," *Procs. 6th Nat. Conf. on Art. Intell.*, Seattle, July 1987, 178-182.

[Rei83] A. Reinefeld, "An Improvement of the Scout Tree-Search Algorithm," *Int. Computer Chess Assoc. J.* **6**(4), 4-14 (1983).

[ReM87] A. Reinefeld and T. A. Marsland, "A Quantitative Analysis of Minimal Window Search," *Procs. 10th Int. Joint Conf. on Art. Intell.*, (Los Altos: Kaufmann), Milan, Italy, Aug. 1987, 951-954.

[Rei89] A. Reinefeld, Spielbaum-Suchverfahren, IFB 200, Springer-Verlag, Heidelberg, 1989.

[Sch83] J. Schaeffer, "The History Heuristic," *Int. Computer Chess Assoc. J.* **6**(3), 16-19 (1983).

[Sch86] J. Schaeffer, Experiments in Search and Knowledge, Ph.D. thesis, University of Waterloo, Waterloo, Canada, Spring 1986. Also Tech. Rep. 86-12, Computing Science, University of Alberta, July 1986.

[Sch89] J. Schaeffer, "Distributed Game-Tree Search," *J. of Parallel and Distributed Computing* **6**(2), 90-114 (1989).

[Sch90] J. Schaeffer, "Conspiracy Numbers," *Arificial Intelligence* **43**(1), 67-84 (1990).

[Sco69] J. J. Scott, "A Chess-Playing Program" in B. Meltzer and D. Michie (eds.), *Machine Intelligence 4*, Edinburgh University Press, 1969, 255-265.

[SlA77] D. J. Slate and L. R. Atkin, "CHESS 4.5 - The Northwestern University Chess Program" in P. Frey (ed.), *Chess Skill in Man and Machine*, Springer-Verlag, 1977, 82-118.

[StS90] I. Steinberg and M. Solomon, "Searching Game Trees in Parallel," *Procs. Int. Conf. on Parallel Processing (vol 3)*, University Park, PA, Aug. 1990, 9-17.

[VoM87] O. Vornberger and B. Monien, "Parallel Alpha-Beta versus Parallel SSS*," *Procs. IFIP Conf. on Distributed Processing*, (Amsterdam: North Holland), Oct. 1987, 613-625.

[WaW88] T. Warnock and B. Wendroff, "Search Tables in Computer Chess," *Int. Computer Chess Assoc. J.* **11**(1), 10-13 (1988).

[Zob70] A. L. Zobrist, A New Hashing Method with Applications for Game Playing, Tech. Rep. 88, Computer Sciences Dept., University of Wisconsin, Madison, April, 1970. Also in *Int. Computer Chess Assoc. J.* **13**(2), 169-173 (1990).