Heuristic Search in One and Two Player Games[*]

T. Anthony Marsland
University of Alberta
Computing Science Dept.
Edmonton T6G 2H1
Canada
<tony@CS.UAlberta.CA>

Alexander Reinefeld
Paderborn Center for Parallel Computing
Warburger Str. 100
4790 Paderborn
Germany
<ar@Uni-Paderborn.DE>

October 4, 1992

[*]Revised version for the ACM Computing Surveys, Salvatore T. March (editor in chief), University of Minnesota, 271 19th Ave. S, Minneapolis, MN 55455. For review only, please do not cite or copy.

1 Abstract

With the continuing price-performance improvement of small computers there is growing interest in looking again at some of the heuristic techniques developed for problem-solving and planning programs, to see if they can be enhanced or replaced by more algorithmic methods. The application of raw computing power, while and anathema to some, often provides better answers than is possible by reasoning or analogy. Thus brute force techniques form a good basis against which to compare more sophisticated methods designed to mirror the human deductive process. One source of extra computing power comes through the use of parallel processing on a multicomputer, an so this aspect is also covered here.

Here we review the development of heuristic algorithms for application in single-agent and adversary games. We provide a detailed study of iterative deepening A* and its many variants, and show how effective various enhancements, including the use of refutation lines and a transposition table, can be. For adversary games a full review of improved versions of the alpha-beta algorithm (e.g. Principal Variation Search) is provided and various comparisons made to SSS*, Aspiration Search and Scout. The importance of memory functions is also brought out. The second half of the paper deals exclusively with parallel methods not only for single-agent search, but also with a variety of parallelizations for adversary games. In the latter case there is an emphasis on the problems that pruning poses in unbalancing the work load, and so the paper covers some of the dynamic tree-splitting methods that have evolved.

This survey will be of interest to those concerned with fundamental issues in computing, but should be especially appealing to experimentalists who want to explore the limitations of theoretical models and to extend their utility. Hopefully this will lead to the development of new theories for dealing with the search of "average trees".

2 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. Exhaustive approaches can benefit from a wealth of powerful heuristics to eliminate unnecessary combinations that do not affect the final outcome. On the other hand, a related but slightly more elegant computational model is the *state-space* approach in which, from a given initial state of the system and a set of actions (that is, given a description vector), the successor states are expanded until solution criteria, such as reaching a specified goal, are satisfied. By this means exhaustive enumeration may be avoided, since selecting an action transforms one state of a system into another, where perhaps a different set of actions is possible, ultimately reaching a goal state. Thus a solution is a sequence of actions that converts an initial state into a goal state having a pre-specified set of properties. A variety of tasks may be posed in this framework. For example, we can ask for a solution, all solutions, or a minimal cost solution.

Because these state transitions 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* [Bellman and Dreyfus, 1962], 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 of the graph) 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 by finding solutions that are otherwise unattainable.

To illustrate these points, and to provide insights into the use of a principal variation, a transposition table and other generally useful aids that can improve search strategies, 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. However, different kinds of "solutions" are sought in the two subdomains: For single agent games, a solution is described by a path leading to a goal state, while in the multi-agent case a solution subtree (including all possible opponent reactions) is needed. Most work has been done for the two-person case. Although there is a growing interest in multi-agent games [Luckhardt and Irani, 1986; Korf, 1991], this paper will not deal with that generalization.

In single agent and two-person games the order in which the decisions (moves) are

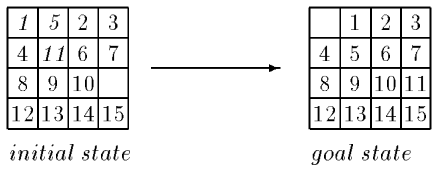


Figure 1: A sample problem instance of the 15-puzzle

made is not necessarily important, so 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, but (to capitalize on a tree's intrinsically simpler and more regular structure) we will initially ignore such duplicated states. Later we will introduce methods that explicitly recognize and eliminate duplicates, and so reduce the effective size of the search space.

3 Single Agent Search

As an example of a single agent search problem consider the N-puzzle game, which has been a popular test bed for search methods over the last few decades. It is easy to implement and yet it offers a combinatorially large problem space of $16!/2 \approx 10^{13}$ states (for the $N = 15$ case). The N-puzzle consists of 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). Figure 1 shows a sample 15-puzzle in which only the highlighted tiles *1, 5* and *11* are out of place.

Humans can be adept at this problem, even when N is large, but solve it without regard to optimality (least tile movement). Finding optimal solutions is difficult since no polynomial-time algorithm is known. In fact Ratner and Warmuth [1986] proved that the problem is NP-hard. A simple (exponential) 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.g., 3×4 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 employed 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 developed and enjoys great popularity. In IDA* the iterative technique controls elegantly the growth of memory needed in the expansion of a single-agent game tree, but in such a way that an optimal solution is still guaranteed.

3.1 A*

The essence of A* [Hart *et al.*, 1968; Nilsson, 1971] 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 to node n (i.e., to the current state) 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 said to be *admissible*: it is guaranteed to find an optimal (least cost) solution, and with minimal node expansions. Dechter and Pearl [1985] proved that: A* never examines a node that can be skipped by some other algorithm having access to the same heuristic information that A* uses.

The computation of $g(n)$ and $h(n)$ is easily seen from the simple 15-puzzle example given later in Figure 3. Here $g(n)$ is the number of tile movements taken so far to convert the start state to the current state, and $h(n)$ estimates the distance to the final state by measuring 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). The initial "bound" in Figure 3 comes from the h value of the root node, and this in turn is $5 = 1 + 1 + (2 + 1)$, being the displacements from their goal position of the highlighted tiles *1*, *5* and *11*, respectively. Clearly, the Manhattan distance never overestimates the number of tile movements required. Hence, optimal solutions are guaranteed with this heuristic, but, as with most state-space search methods, even an almost perfect evaluation function will lead to excessive production of partially expanded states.

3.2 Iterative Deepening A*

By analogy with a technique pioneered in computer chess programs to keep the time cost of search within reasonable bounds [Scott, 1969], 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 lesser-cost solution exists. Korf's [1985] *Iterative Deepening A*, IDA** for short, eliminates the need to maintain open/closed lists of state descriptors, and so has linear space overhead with negligible memory management costs.

IDA* performs a series of depth-first searches with successively increased cost-bounds. As in A*, the total cost $f(n)$ of a node n is made up of the cost already spent in reaching that node $g(n)$, plus the estimated cost of the path to a goal state $h(n)$. At the beginning, the cost bound is set to the heuristic estimate of the initial state, $h(root)$. Then, for each iteration, the bound is increased to the minimum path value that exceeded the previous bound:

```
procedure $IDA^*$ $(n)$;
$bound := h(n)$;                      { initial cost bound is $h(root)$ }
while not $solved$ do                 { iterate until solved ... }
    $bound := DFS(n, bound)$;         { ... with increased cost bound }
output $bound$ as optimal solution cost;


function $DFS$ $(n, bound)$;
if $f(n) > bound$
    then return $f(n)$;               { path cost exceeds bound }
if $h(n) = 0$
    then return $solved$;
return lowest value of $DFS(n_i, bound)$ for all successors $n_i$ of $n$
```

While this high-level description illustrates the general search scheme, Figure 2 gives a refined version and shows precisely how the actual node expansion is performed by the depth-first-search *DFS* routine. If successful, *DFS* returns either the cost of the optimal solution path, or an updated *new_bound* for the next iteration; otherwise only nodes with cost less than the specified *bound* are expanded in the current iteration. When the estimated completion cost, $c(n, n_i) + h(n_i)$, of a path going from node n via successor n_i to a (yet unknown) goal node does not exceed the current cost *bound*, the search is recursively deepened by calling *DFS* on the successor node n_i. Otherwise, n_i is cut off and the node expansion continues with the next successor n_{i+1}. Here $c(n, n_i)$ is the cost of moving a tile from n to n_i.

Again, illustrating with the N-puzzle, if *root* represents the start state, a lower bound on the total solution cost is:

$$bound = f(root) = h(root),$$

since $g(root)$ is zero at the start position, so at least *bound* tile movements are needed. Thus during the first iteration, solutions of cost $h(root)$ are sought. As soon as the condition

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

```
function DFS (n: node; bound: integer): integer;
begin                                              {returns next cost bound}
    if $h(n) = 0$ then begin
        $solved \leftarrow true$; return 0;        {goal state found}
    end;
    $new\_bound \leftarrow \infty$;
    for each successor $n_i$ of $n$ do begin
        $merit \leftarrow c(n, n_i) + h(n_i)$;
        if $merit \leq bound$ then begin           {search deeper}
            $merit \leftarrow c(n, n_i) + DFS(n_i,\ bound - c(n, n_i))$;
            if solved then return merit;
        end;                                       {path cost reached bound: cutoff}
        if $merit < new\_bound$ then
            $new\_bound \leftarrow merit$;         {cost bound for next iteration}
    end;
    return new_bound;                              {return next iteration's bound}
end;
```

Figure 2: Cost bounded depth-first search (refined version)

holds, the search from node n is stopped. 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 no goal state is found, the cost bound is increased to the smallest of all the $g(n) + h(n)$ values that exceed the current cost bound. Figure 3 illustrates the first two iterations of an IDA* search on a simple problem instance. Since none of the nodes in the first iteration is a goal node, a second iteration with an increased cost bound (here 7) is needed. As it turns out, it is not until the fifth iteration (not shown here) that a solution is found, and then only after a total of 215 node generations.

Note that the pseudo-code in Figure 2 also includes the *cost revision* idea of Chakrabarti *et al.* [1989], according to which the current estimate $f(n)$ is revised to the smallest $f(m)$ amongst all tip nodes m that are descendants of n. The revised cost values are collected in the variable *new_bound* and are recursively backed up to the parent node. The motivation behind this scheme is to increase the information horizon and therefore to tighten the bound on the true solution cost $f^*(n)$. When memory space is available (e.g. to hold node information in a hash access table), the revised cost values can be preserved and reused later to improve the node expansion order and/or to cut off subtrees that

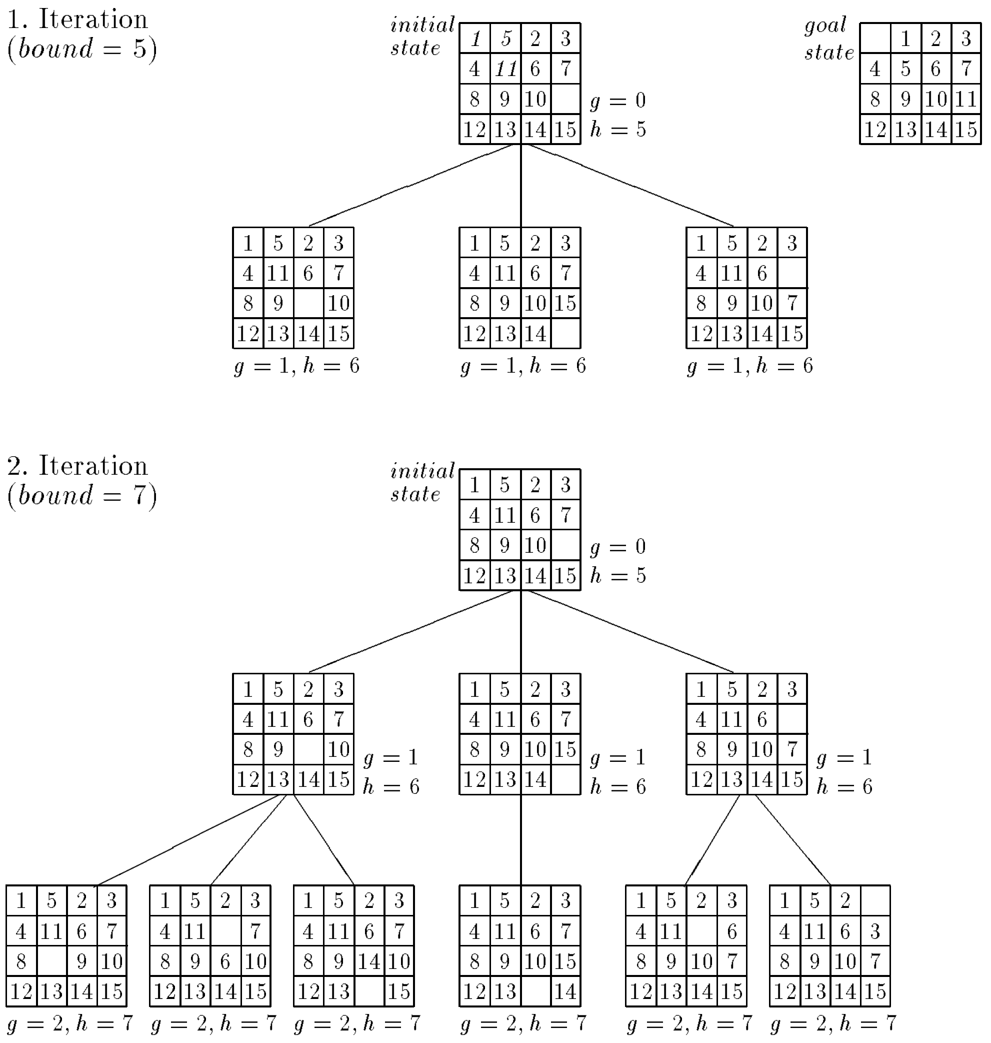


Figure 3: First two iterations on a 15-puzzle problem instance

must otherwise be expanded. The pseudo-code in Figure 2, however, shows the simple scheme without a memory table. Here the revised values are only used to determine the new cost bound for the next iteration.

3.3 Domains of Applicability

Despite the computation wasted in earlier iterations, IDA* is effective when the search space grows exponentially with increasing cost bounds. This growth rate is called the *heuristic branching factor*, b_h, and is defined as the average node ratio of two consecutive iterations. Its magnitude depends primarily on the characteristics of the application (e.g. tree width), and secondarily on the quality of the heuristic estimate and the effectiveness of the search method. For the 15-puzzle, we determined $b_h \approx 6.7$ [Reinefeld and Marsland, 1991], although the tree has an *edge branching factor* of only $b_e \approx 2$, because there are an average of two successors to each node. Hence, the N-puzzle is one of the rare applications with $b_h > b_e$, making it especially amenable for the iterative-deepening approach. The cost bound is constantly increased by the constant 2, which ensures that *all* frontier nodes are expanded by at least one extra tree level, and some of them much farther. Therefore most of the work is done in the last iteration (especially if all the minimal cost solutions are sought), and the overhead spent in re-examining the shallow tree levels does not affect the asymptotic growth rate of the total search effort. This observation is confirmed by Korf [1985], and Stickel and Tyson [1985] showed analytically that iterative deepening needs only a small constant factor $w/(w-1)$ more node expansions than breadth-first search, when breadth-first does not eliminate possible duplicate nodes.

Care must be taken in applications with low heuristic branching factors $b_h < 2$, where iterative deepening might not pay for the increased overhead. In such applications, the operator costs and/or the heuristic estimate values are typically drawn from a large domain, so that the cost bound increases only by a small amount. The worst case is reached when all f-values are unique and monotone. Under such circumstances IDA* is no longer asymptotically optimal[1]. It then expands $N(N+1)/2$ nodes [Patrick *et al.*, 1992], where N is the number of node expansions of A*. Whether such extreme cases occur in practice depends mainly on the domain of the operator costs. As an example, some traveling salesman problems require real valued operator costs, which make it very likely that only one frontier node is expanded in each new iteration. Clearly, iterative-deepening is not efficient and other search schemes should be applied instead. Rao *et al.* [1991] identify characteristics of problem domains in which one of *iterative-deepening*, *best-first* or *depth-first branch-and-bound* performs the best.

However there also exist highly refined variants of IDA*, that have been especially designed for applications with low heuristic branching factors. Those enhancements are based on either a more liberal increase of the cost bound, or on the usage of previously stored node information in later iterations. The first class includes an early proposal of Stickel and Tyson [1985], named *evenly bounded depth-first search*, an iterative deepening

[1]The proof of IDA*'s asymptotic optimality [Korf, 1985] has proved to be incomplete – even under Korf's original requirements of finite precision and non-exponential node costs. Mahanti *et al.* [1992a] provide a new set of necessary and sufficient conditions.

with controlled re-expansion, named *IDA*_CR* by Sarkar *et al.* [1991], and the hybrid iterative-deepening depth-first branch-and-bound variants *DFS** [Rao *et al.*, 1991] and *MIDA** [Wah, 1991]. All these schemes attempt to reduce the search overhead by increasing the cost bound by more than the minimal value. Although some of the early methods do not guarantee the first solution to be optimal, usually a worst case error can be given. If optimality is still required, the search can be resumed with the solution cost as a cost bound until the whole iteration is completed. (In domains, where the cost bound increments are known *a priori* it would be better to use the next lower cost bound.)

3.4 Memory Variants

The second class of IDA* enhancements includes memory-bounded search methods, that use available memory to preserve node information from one iteration to the next. They make flexible use of all available space, rather than relying on a fixed memory size (like A*). Of the various proposals, two early schemes are noteworthy: *MREC* by Sen and Bagchi [1989] and *MA** by Chakrabarti *et al.* [1989]. While differing in detail, both algorithms operate on an explicit search graph that maintains as much node information from the last iteration as possible. When the memory space is exhausted, the node expansion strategy changes from best-first to depth-first, either by recursively re-expanding all nodes starting at the root (MREC), or by further extending the tip nodes of the explicit search graph (MA*). This slight difference makes MREC more time efficient (since it does not need an open list), and MA* more efficient in terms of node expansions (since it never re-visits the shallow nodes). In practice, the node savings of MA* do not compensate for the time spent in the more complex memory management, especially since the node savings originally reported by Chakrabarti *et al.* [1989] could not be verified by other researchers [Mahanti *et al.*, 1992b].

In summary, these methods – and also the later proposals like Russell's [1992] *SMA** and *IE* – maintain an explicit search graph, which is often too expensive. Hashing tables [Reinefeld and Marsland, 1991] are much preferred in practice, because of their simpler structure and faster access time. Their main purpose is to eliminate unnecessary search caused by *move transpositions* (i.e. different paths ending in the same position), but they are also useful in improving the node expansion order. In the 15-puzzle, when a table of size 256k is used to eliminate transpositions and move cycles, IDA* search is reduced to about half of the nodes, see Trans in Table 1. Move pre-sorting according to the table information (Trans+Move) saves an additional 7% of the expansions. (Note,

Search Method	*Nodes [%]*		*Time*
	mean	*std*	*[%]*
IDA*	100		100
Sort	99	42	114
PV	86	52	85
History	94	48	107
Trans	53	6	82
Trans+Move	46	28	64
minimal IDA*	54	26	–

Table 1: Performance of the IDA* memory variants on the 15-puzzle

here the transposition table entries can hold a variety of information, but most useful is the best choice, Move, found earlier.) This improvement is more than can be achieved with a heuristic sorting scheme (Sort), because the transposition table contains revised cost values (from deeper tree levels) that are usually better estimates and thus more valuable.

Another obvious enhancement is to start a new iteration on the most promising path achieved before. The *principal variation* in adversary games is the move sequence actually chosen if both opponents adhere to the minimax principle. In single-agent search, the principal variation (PV) corresponds to the path that ends in a leaf node with the lowest h- and largest g-value, that is, the path that has greatest potential to be nearest to a goal. As can be seen in Table 1, PV reduces the node count by 14%, but exhibits an undesirably high standard deviation. The *history heuristic* [Schaeffer, 1989b], which also proved useful in the domain of two-player games, achieves its performance by maintaining a score table for every move seen in the search tree. All moves that are applicable in a given position are then examined in order of their previous success. In the 15-puzzle, it seems that the history heuristic is not successful (achieving only a meager 6% reduction), because there exist only weak criteria to define the merit of a move.

In summary, two categories of IDA* refinements have been identified: heuristics that improve the node expansion order (Sort, PV, History), and mechanisms to avoid the multiple re-expansion of the same nodes (Trans, Trans+Move). While the former help only in the last iteration, the latter always reduce the search effort – at the cost of some additional memory space. As a form of lower bound we include in Table 1 an entry for "minimal IDA*", a nondeterministic optimal version that finds the goal

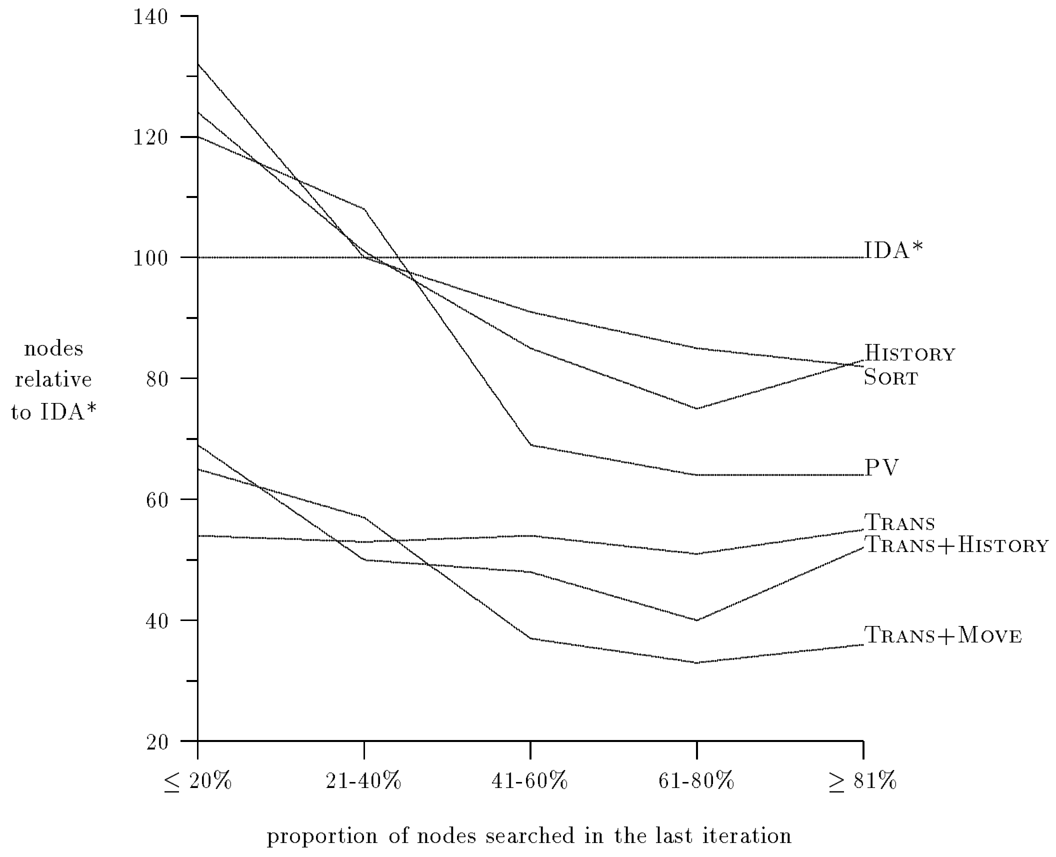


Figure 4: Relative performance of IDA* enhancements on the 15-puzzle

node immediately at the start of the final iteration. Interestingly, all variants involving Trans examine fewer nodes than this optimal IDA* search.

Figure 4 shows the performance of the memory variants in a graphical form. Here, Korf's [1985] hundred random problem instances are grouped into five sets, defined by the ratio of the nodes searched by IDA* in the last iteration to the total nodes. The problems of the first set (0-20%) are already well ordered for IDA* and it seems hard to achieve further savings with any of the move ordering heuristics. On the contrary: in their attempt to further improve the expansion order, History, Sort and PV often expand more nodes in the end. Only with decreasing tree order (e.g. when more than 40% of the node expansions occur in the last iteration), do these techniques become more effective than pure IDA*.

On the other hand, schemes involving a hash table are almost equally effective over the whole range of problems. Use of a basic transposition table (Trans) reduces the node count by one half, while additional savings are achieved with the successor ordering techniques MOVE and HISTORY, especially when the tree is poorly ordered. In practice, one would use the combined version TRANS+MOVE, as Figure 4 shows.

The use of memory functions has a long history [Doran and Michie, 1966], but sometimes they don't work as well as they should. Consider, for example, bi-directional search [Pohl, 1971] which expands the solution space simultaneously from both the start state and the goal state, and tries to match states at the search frontiers. In principle this should work well, but as Pohl points out "both search trees grow almost complete but separate solution paths, with intersection occurring near one or the other of the end points" [Pohl, 1971].

3.5 Coping with Incomplete Information

Even though the IDA* memory variants are time efficient and flexible in their space requirements, complex problems spawning large search spaces can only be solved with approximate methods. In an early paper, Pohl [1970] introduced a weight factor w to the evaluation function $f(n) = (1 - w) \cdot g(n) + w \cdot h(n)$ that allows the "focus" of the search to vary from pure breadth-first ($w = 0$) to pure hill climbing ($w = 1$). One important intermediate value, $w = 0.5$, gives an admissible A* search. The relation between efficiency of search and the accuracy of the heuristic function is discussed and formally analyzed in Pearl's book on Heuristics [1984, p. 86ff].

While weighted evaluation functions focus the search process by pruning branches that are thought to be irrelevant, the tree might still be too large for a complete expansion up to the leaf nodes. In two-player games, it is customary to stop the lookahead search at some point and back up estimate values. Research in single agent search, in contrast, has concentrated primarily on finding optimal solutions. One exception is Korf's [1990] *Real-Time A* (RTA*)* that commits to a decision within a pre-defined time allotment. It does so by performing a lookahead up to a limiting search depth and backing up the minimum of the frontier node values $f(n)$ to the root. Then the apparently best move is (physically) made, and the planning process starts again on the new configuration. Fortunately, not all frontier nodes must be expanded. With a monotonic cost function f (that never decreases along a path from the root) all nodes whose f-value is higher than the lowest f-value encountered on the search frontier so far can be cut off. This is analogous to a pruning technique used in two-player games, so Korf calls it *alpha pruning.* In the 15-puzzle, alpha pruning allows the search horizon to be doubled from 18 to 40 moves. Moreover, the search horizon achievable with alpha pruning actually increases with increasing branching factor [Korf, 1990, p. 195].

Weaker solutions can be obtained even faster by applying a sequence of *macro operators* [Korf, 1985] that incrementally solve one subgoal after the other, while leaving the previously solved subgoals intact. For the 15-puzzle, a table of macros that describes the primitive moves required to shift a given tile from any initial position to its destination is needed, but without impairing the position of the remaining tiles (temporary disorderings are allowed, though). Note, that no search is required at problem solving time, since all knowledge is contained in the macro table. Thus, problem solving with macro operators might be viewed as an extreme form of pre-processing.

4 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 approach. In practice, the memory variants of IDA* are perhaps the most practical schemes. They combine the advantages of a depth-first search with that of a best-first search, by saving a limited amount of node information to improve the node expansion order and to eliminate irrelevant subtrees.

The advantages of iterative deepening 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 recursive back-up procedure. Given an evaluation function $f(n)$ which can measure the value of any leaf node from the first player's view, we get for a leaf node n

$$MinMax(n) = f(n) = Evaluate(n),$$

and for any interior node n with successor nodes n_i

$$MinMax(n) = \begin{cases} \max\limits_i \; (MinMax(n_i)) & \text{if } n \text{ is a Max node} \\ \min\limits_i \; (MinMax(n_i)) & \text{if } n \text{ is a Min node} \end{cases}$$

A more concise 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 symmetric definition:

$$NegMax(n) = \max_i \,(-NegMax(n_i)).$$

Here, all node values are taken from the players point of view, and $f(n)$ is a function that computes the merit value of a leaf node n for the last player. For a true leaf (with 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^{73} or 10^{173} nodes, respectively. In these applications, the evaluation function $Evaluate(n)$ 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 [McAllester, 1988; Rivest, 1988; Anantharaman, 1991] an attempt is made to account for the uncertainty in the leaf values). 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 minimax value of the tree. Ballard [1983] on the other hand considers pruning in probabilistic games, where the issue is whether a branch in the game tree will be taken, rather than any uncertainty in the leaf values. This interesting study is long overdue for follow-up work.

4.1 Alpha-Beta

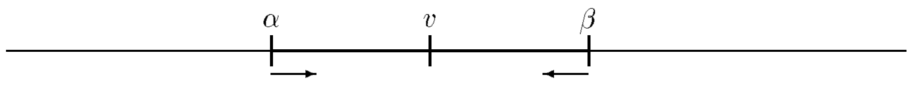
One early paper on computer chess [Newell *et al.*, 1958] recognized that a full minimax search was not essential to determine the value of the tree. Some years later a little known work by Brudno [1963] provided a theoretical basis for pruning in minimax search. From these observations, the alpha-beta 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.

As pointed out in Knuth and Moore's classical paper [1975], the alpha-beta algorithm is a refined branch-and-bound method that uses a special backing up rule (the minimax rule) and employs two bounds rather than one – i.e. one bound for each player. The bound used at all Max nodes (including the root) is named *Alpha*, and the bound for the Min nodes is named *Beta*. *Alpha* represents the least value player Max is guaranteed (no matter how good Min's moves are), while *Beta* is the value Min can expect, at the very least.

```
function $ABSearch$ ($n$: node; $\alpha$, $\beta$, $height$: integer): integer;
begin                                                  {returns minimax value of $n$}
    if $height = 0$ or $n$ is a leaf then
        return $Evaluate(n)$;                          {leaf or frontier node}
    $next \leftarrow SelectSuccessor(n)$;              {generate first successor}
    $estimate \leftarrow -\infty$;
    while $next \neq NULL$ do
        $merit \leftarrow -ABSearch\ (next,\ -\beta,\ -max(\alpha, estimate),\ height - 1)$;
        if $merit > estimate$ then begin
            $estimate \leftarrow merit$;               {improved value}
            if $estimate \geq \beta$ then return $estimate$; {cut-off}
        end;
        $next \leftarrow SelectBrother(next)$;         {generate brother}
    end while;
    return $estimate$;                                 {return the subtree value}
end;
```

Figure 5: Fail-soft alpha-beta algorithm

In effect, the search can be viewed as taking place within a *window* or range of integer values *Alpha* to *Beta* with the underlying assumption that the value, v, of the tree lies in that range:



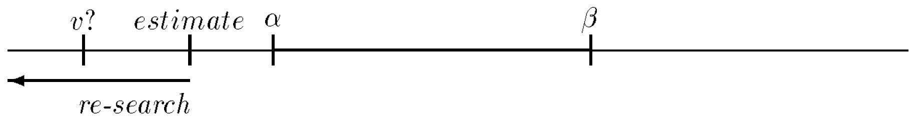
Clearly if the initial values of *Alpha* and *Beta* are $-\infty$ and $+\infty$ respectively, the merit value of the tree will fall within that infinite range. However, the window will not stay so wide; it automatically reduces as the move choices are explored and the merits of the alternatives become clear. This change occurs as Max raises the *Alpha* bound in an attempt to maximize its gain, whereas Min reduces *Beta* when it tries to minimize Max's outcome. "The final game value is pulled from two sides", as von Neumann and Morgenstern [1944] observe in their historical paper that put game theory on a formal basis.

In *ABSearch* (see Figure 5) the integer parameters α and β represent lower and upper bounds, and *height* is the remaining distance (in ply) to the search frontier. Also, n represents a pointer to the root node of the (sub-)tree, and the functions $SelectSuccessor(n)$ and $SelectBrother(n)$ return the first successor and the next brother of n, respectively. If the minimax value v lies within the (α, β)-window, *ABSearch* returns the merit value of the subtree by using a recursive backing up process.

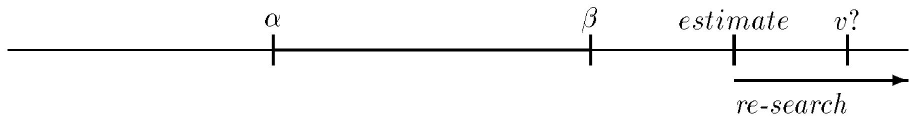
4.2 Aspiration Search and the Fail-Soft Improvement

One popular enhancement to the alpha-beta algorithm, called *aspiration search* [Slate and Atkin, 1977; Baudet, 1978], artificially narrows the (α, β)-bounds, hoping to reduce the search space right from the beginning, 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 [1981] *fail-soft* idea, which returns an approximation of the true minimax value when outside the (α, β)-window. This is achieved by initializing *estimate* to $-\infty$, as Figure 5 shows, instead of to the expected lower bound α. Thus, even if the initial (α, β)-bounds are too narrow, so that the search can fail, we will know whether $v \leq estimate \leq \alpha$,



or whether $\beta \leq estimate \leq v$,



The initial search not only determines whether the search fails low or high, but also provides an upper or lower bound (*estimate*) for the true minimax value. Aspiration search builds on the fail-soft mechanism. It initially restricts the (α, β)-bounds to a narrow range $(v_0 - \epsilon, v_0 + \epsilon)$ around an expected tree value, v_0, and conducts re-searches with a properly enlarged window $(-\infty, estimate)$ or $(estimate, +\infty)$, as required.

The advantages of working with narrow bounds can be significant, especially for games where it is easy to estimate v_0. An aspiration search gambles that the time saved by successful searches with narrow windows will more than pay for the re-searches that follow failure. 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

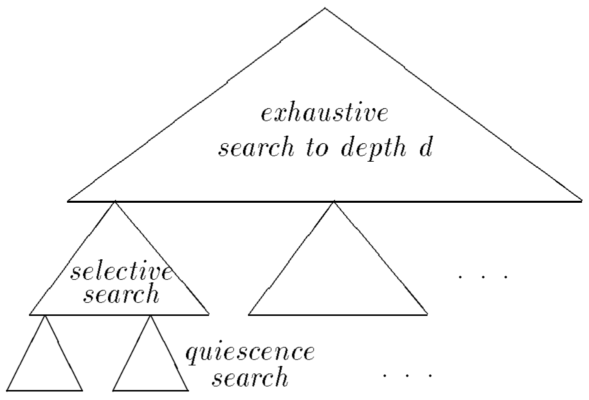


Figure 6: Staged search of game trees

provides a good move ordering, and with iterative deepening 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.

4.3 Approximating Game Trees

In practice, because game trees are so large, one must search a series of approximating subtrees of length *height*, based on the code in Figure 5. Thus, instead of true leaf nodes, where the value of the node is known exactly, we have pseudo-leaf or frontier nodes where the value of the unexplored subtree beyond this horizon is estimated by a heuristic 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 stages or phases, as Figure 6 suggests. From the root all moves are considered up to some fixed depth d (usually a constant), but if a node has only a few (one or two) legal successors (e.g. after a checking move in chess) the length of the path is extended by one ply. Thus 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 cause a search extension), but at every new node only a selection of the available moves is considered. This heuristic incorporates the dangerous and discredited practice of *forward pruning*. It works here because the exhaustive search layer finds short term losses (obvious sacrifices)

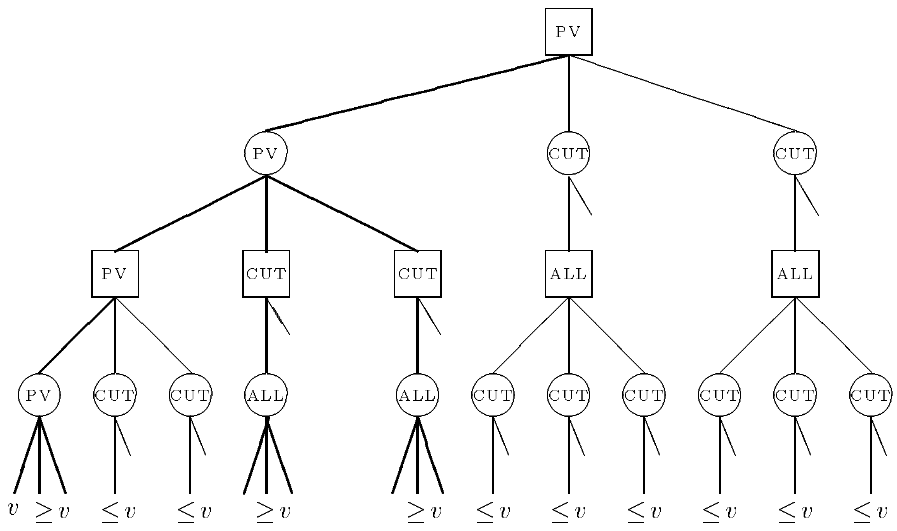


Figure 7: Structure of a critical game tree

that lead to long term gains, while the next stage uses forward pruning to eliminate not only moves that persist in losing, but also 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 the search is more dynamic. It is called a *quiescence search*, because its purpose is to improve the evaluation estimate of critical frontier nodes that involve dynamic terms which 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 approach has many practical advantages and can be used equally effectively in other decision tree applications.

There is as yet no theoretical model for these variable depth search processes. Previous analytical studies usually restricted 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 that the critical (optimal) game tree has

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

leaf nodes [Slagle and Dixon, 1969]. This is the fewest nodes that must be examined by *any* search algorithm to determine the minimax value. In Knuth and Moore's [1975] terminology, the minimal or critical game tree is made up of type 1, type 2 and type 3 nodes, although Marsland and Popowich [1985] call these PV, CUT and ALL nodes, respectively, to make it clearer where cut-offs may occur. A more recent study considered the properties of an average game tree, presented formula for the tree size, and showed how the different nodes types are distributed [Reinefeld and Marsland, 1987].

Figure 7 illustrates the general structure of a critical game tree, where the Principal Variation (the PV nodes) is located in the leftmost path and all alternatives are refuted with minimal effort (the CUT nodes). Figure 7 also helps to count the leaf nodes in the critical game tree. To prove that the minimax value of a game tree is v, at least $W^{\lfloor \frac{D}{2} \rfloor}$ leaf node expansions are needed to show that there exists one subtree (the highlighted one in Figure 7) with a value $\geq v$, and at least $(W^{\lceil \frac{D}{2} \rceil} - 1)$ expansions to show the inferiority of the remaining $W - 1$ alternatives at Max nodes along the PV, which have value $\leq v$. Together these two quantities yield the Slagle and Dixon result quoted earlier.

4.4 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 uses a *Null Window* to eliminate other contenders, or to provide a lower bound for a new principal variation. The fundamental idea 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 of size $(-\alpha - 1, -\alpha)$ is used so that no integer value can fall between the two adjacent bounds. Thus all remaining searches with that window will fail, hopefully low, proving the inferiority of the move with minimal effort. If the null window search fails high, then the current move is superior to the previously best and the search must be repeated with the correct bounds, to find the proper path and value, along the lines of Figure 8.

Although correct and efficient, the definition of PVS given here is fully recursive, so it is not easy to distinguish between PV, CUT and ALL nodes. It also provides a version of the algorithm that is difficult to parallelize at the places where most work is to be done. This disadvantage can be overcome by replacing the line

```
**function** $PVS$ ($n$ : **node**; $\alpha$, $\beta$, $height$: **integer**): **integer**;   {for PV nodes}
    **if** $height = 0$ **or** $n$ is a leaf **then**
        **return** $Evaluate(n)$;                                    {leaf or frontier node}
    $next \leftarrow SelectSuccessor(n)$;
    $best \leftarrow -PVS\ (next,\ -\beta,\ -\alpha,\ height - 1)$;      {PV node}
    $next \leftarrow SelectBrother(next)$;
    **while** $next \neq NULL$ **do**
        **if** $best \geq \beta$ **then**
            **return** $best$;                                       {CUT node}
        $\alpha \leftarrow max(\alpha,\ best)$;
        $merit \leftarrow -PVS\ (next,\ -\alpha - 1,\ -\alpha,\ height - 1)$;
        **if** $merit > best$ **then**
            **if** $merit \leq \alpha$ or $merit \geq \beta$
                **then** $best \leftarrow merit$                      {improved value}
                **else** $best \leftarrow -PVS\ (next,\ -\beta,\ -merit,\ height - 1)$;  {re-search}
        $next \leftarrow SelectBrother(next)$;
    **end while**;
    **return** $best$;                                               {return the subtree value}
**end**;
```

Figure 8: Principal Variation Search

$$merit \leftarrow -PVS\ (next,\ -\alpha - 1,\ -\alpha,\ height - 1);$$

by a call to the *Null Window Search* function given in Figure 9, as follows:

$$merit \leftarrow -NWS\ (next,\ -\alpha,\ height - 1);$$

Figure 10 can now be used to show the structure of a typical game tree and provides a small example of the use of the negamax framework to illustrate exactly how the bounds are set and how the tree's merit value is backed up. Here the CUT and ALL nodes are searched only by NWS, while the two PV nodes leading immediately to the node with the minimax value of +6 are searched first by NWS and then re-searched by PVS. Thus the figure also shows both the occurrence of a fail-high re-search, and a case where the leftmost (first) branch of a cut node does not produce a cut-off (e.g, at the CUT node that is the right successor of the root node).

The original compact formulation 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 the additional NWS routine serves two purposes: first this

```
function $NWS$ ($n$ : node; $\beta$, $height$: integer): integer; {for ALL and CUT nodes}
    if $height = 0$ or $n$ is a leaf then
        return $Evaluate(n)$;                          {leaf or frontier node}
    $next \leftarrow SelectSuccessor(n)$;
    $estimate \leftarrow -\infty$;
    while $next \neq NULL$ do
        $merit \leftarrow -NWS\ (next,\ -\beta + 1,\ height - 1)$;
        if $merit > estimate$ then
            $estimate \leftarrow merit$;               {improved value}
        if $merit \geq \beta$ then
            return $estimate$;                         {CUT node}
        $next \leftarrow SelectBrother(next)$;
    end while;
    return $estimate$;                                 {ALL node}
end;
```

Figure 9: Null window search function for use with PVS

separation helps in the design of a parallel algorithm, since the parallelization can now be done at the PV and ALL nodes, and second it makes possible a direct comparison with Scout [Pearl, 1980].

4.5 Scout and its Variants

PVS [Marsland, 1983], NegaScout [Reinefeld, 1983] and their predecessor P-Alphabeta [Fishburn, 1981] emerged from a slightly inferior search scheme, the *Scout* algorithm [Pearl, 1980], which was invented to simplify the proof of the asymptotic optimality of *ABSearch*. The original formulation [Pearl, 1980] is clumsy and difficult to implement, because it requires three separate routines: an evaluation routine to determine the exact minimax value of a PV node and two boolean test routines to prove alternative subtrees inferior. Despite Scout's complex structure, and although it re-expands certain subtrees several times, Scout proved surprisingly efficient in test cases. Consequently, fully recursive versions of Scout have been designed [Marsland, 1983] but particularly interesting is the *NegaScout* model [Reinefeld, 1983], which Kaindl [1990] shows to be more efficient than Scout in practice. Theoretical investigations [Reinefeld and Marsland, 1987; Reinefeld, 1989] confirm, that NegaScout (NS) expands fewer nodes than *ABSearch* in well ordered trees, which are often found in practical applications. Conditions are fa-

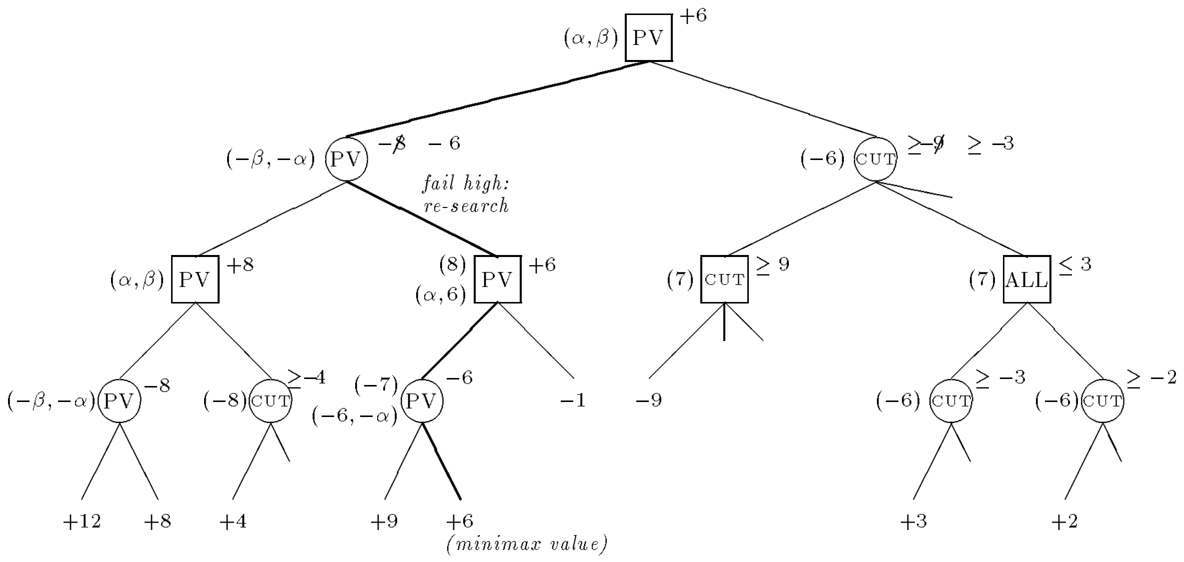


Figure 10: Sample pruning of a minimax tree by PVS/NWS

vorable for NegaScout when only few re-searches are needed in PV nodes (that is, when the principal variation does not change) and when the best successor in CUT nodes is found early, as Figure 11 illustrates. In practice, domain-dependent heuristic knowledge is used to order the node successors before expansion. In chess trees, for example, only 1.5 CUT node successors are expanded on average before a cut-off occurs [Marsland and Popowich, 1985]. This gives NegaScout and PVS a clear advantage over *ABSearch*.

NegaScout also introduced an admissible (without error) pruning technique near the frontier, in contrast to the more speculative *razoring* method of Birmingham and Kent [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 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 β-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

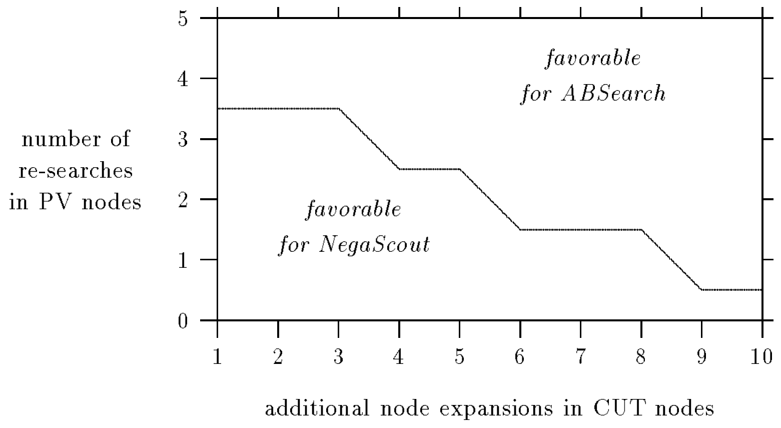


Figure 11: NegaScout vs. *ABSearch* (width= 20, depth= 7)

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 should 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.

4.6 Best-First Search Methods

For 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 [1985] developed an algorithm called *PB** to introduce probability distributions into the evaluation process.

*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

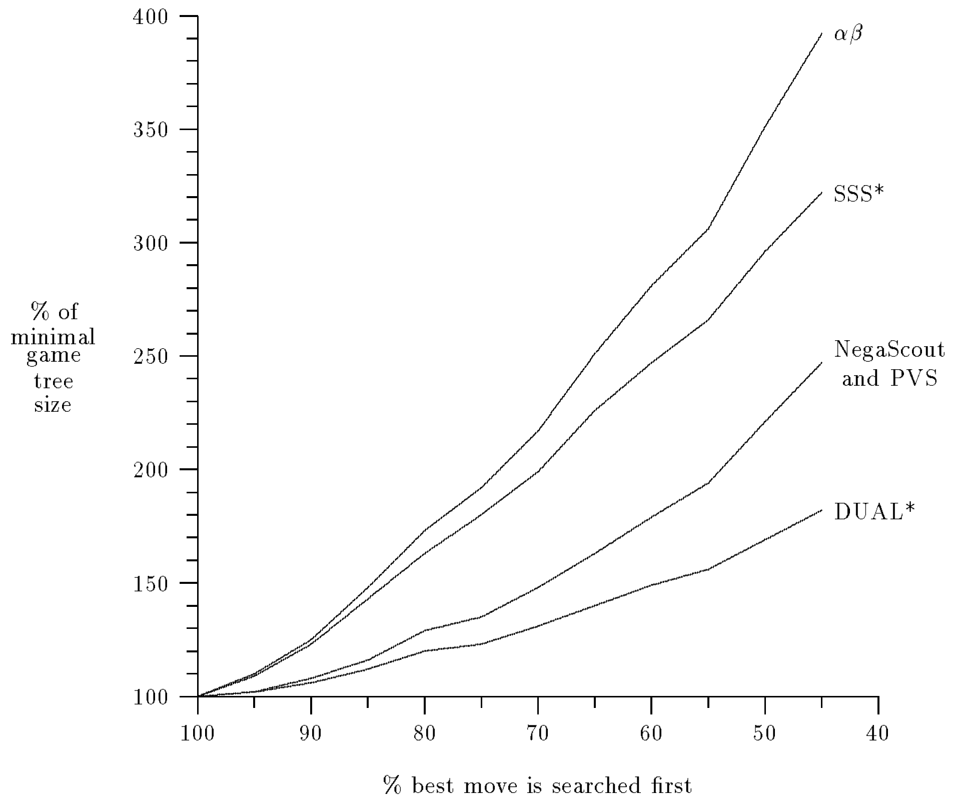


Figure 12: Search of strongly ordered trees (width= 20, depth= 5)

never visits more leaf nodes. Also, with the change proposed by Campbell in his 1981 thesis (reported later [Campbell and Marsland, 1983]) 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 5). Because SSS* dominates a simple alpha-beta search hopes were raised that this reflects a reduced exponential growth rate. Such hopes, however, were refuted when Roizen and Pearl [1983] showed that SSS* and Alpha-Beta are asymptotically equivalent. Only in moderate search depths $d = 2, \ldots, 20$ and moderate widths $w = 2, \ldots, 20$ is SSS* of advantage. But this advantage is limited and reaches a peak at depth $d = 8$, where Alpha-Beta needs a maximum of three times as many node expansions as SSS*.

In an attempt to find a version of SSS* that achieves a better storage efficiency, two hybrid algorithms, *PS** by Marsland and Srimani [1986] and *ITERSSS** by Bhattacharya and Bagchi [1986] were invented. They break SSS* into a series of k-partitions which are searched one at a time. Although the original idea was to form some kind of iterative broadening [Ginsberg and Harvey, 1992] for adversary game trees, the end result was a spectrum of search algorithms whose performance matched the range from SSS* ($k = 1$) to Alpha-Beta ($k = W$).

While impressive node reductions and catchy publications like "A minimax algorithm better than Alpha-Beta?" [Stockman, 1979] attracted the attention of the game-playing community, it was often overlooked that simple depth-first search algorithms, like PVS and NegaScout also dominate Alpha-Beta. Moreover, these other algorithms are usually faster than SSS*, because they are based on a simple recursive depth-first expansion scheme borrowed from Alpha-Beta. Even though the structure of these new algorithms is similar to Alpha-Beta, SSS* does not dominate PVS or NegaScout (there even exist cases where SSS* examines more nodes than either), as Reinefeld [1989] proves.

Statistically, most 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 performing a directed (left to right) search at the root node and an SSS* search below that. Thus DUAL* has a lower memory requirement (since it uses SSS* to search a 1-ply shallower tree), but otherwise shares the same burdensome overheads. Its search characteristic resembles that of PVS and NegaScout, with the nice distinction that there is no need for a re-search, because DUAL* simply resumes its best-first node expansion based on the previously acquired information. Although Reinefeld [1989] 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 12 illustrates for *strongly ordered trees* [Marsland and 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 do well here because the trees used were not random; in fact, 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 faster execution, quite the contrary.

This state of affairs has recently been alleviated by *RecSSS**, a recursive SSS* formulation of Bhattacharya and Bagchi [1990]. RecSSS* needs only twice as much time as Alpha-Beta in searching synthetic game trees of the type described by Marsland *et al.* [1987]. It does so by allocating all the needed memory right at the beginning of the search. All memory cells are directly addressed, so that no time-consuming list, queue or heap management is needed. Thus, the design of RecSSS* follows the lines of *INS*

[Reinefeld *et al.*, 1985], an "informed" version of NegaScout, that also allocates memory space in larger chunks. INS turned out to be even faster, and it can reasonably compete with Alpha-Beta on a CPU-time basis, while allowing flexible use of additional memory. The memory requirements of RecSSS*, in contrast, are of the same order of magnitude as that of SSS*.

Of the newer techniques, McAllester's [1988] so called *conspiracy number search* is especially interesting. Although this method also makes heavy demands on computer memory, it is in 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] compare 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 also be of great importance.

4.7 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 70s 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 (the tip nodes) 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 bookkeeping required for the refutation table is best described in the book by Levy and Newborn [1990].

4.8 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 search domain, 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 revived [Marsland and Campbell, 1981], with further insights by Nelson [1985] and by Warnock and Wendroff [1988].

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. When the length field 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, this not only enables a search of less than the critical game tree, but also extends the search of some variations to almost double the frontier distance. More common, however, is use of the stored "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 re-search down the refutation line.

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

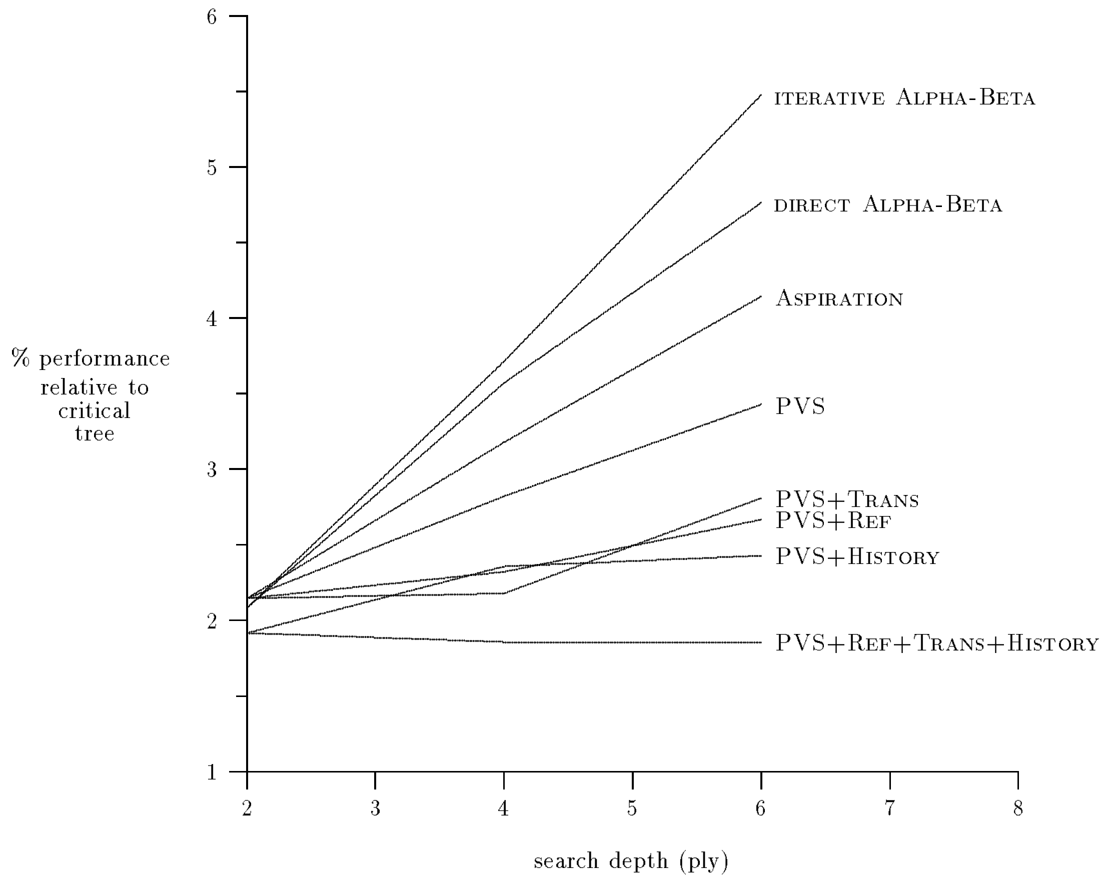


Figure 13: Node comparison of alpha-beta enhancements (even depths only)

frequency of prior pruning success is a more powerful means of ordering moves than using application dependent heuristic knowledge. In turn, move ordering dramatically improves the efficiency of directional searches like *ABSearch* and *PVS*.

4.9 Combined Enhancements

The relative efficiencies of these various alpha-beta enhancements are adequately captured in Figure 13, which presents data from a model chess program (Parabelle) searching a suite of test positions. The size of the critical game tree (having $W^{\lceil \frac{D}{2} \rceil} + W^{\lfloor \frac{D}{2} \rfloor} - 1$ nodes, with $W = 34$, in this case) is taken as the 100% basis for comparison. Figure 13 shows that under reasonable assumptions PVS is more efficient than ASPIRATION search (although optimal aspiration windows will necessarily do better [Shams *et al.*, 1991]). Further, the memory function assistance of transposition table (+TRANS), refutation table (+Ref) and history table (+History) for re-ordering the moves are additive and make a significant improvement in performance. Note that the performance graph of the fully enhanced PVS (PVS+Ref+Trans+History) is linear to the size of the minimal tree, indicating an exponential efficiency gain with increasing search depths. The worsening result for the 6-ply search by PVS with transposition table (PVS+Trans) has been attributed to overloading of the small (only 8K entries) transposition table [Marsland, 1987].

Although Figure 13 clearly points out the merits of the iterative-deepening enhancements used in adversary game tree searches, it is also instructive to compare this data to the single-agent searches, shown in Figure 4. Using iterative alpha-beta and iterative-deepening A* as a base method for adversary and single-agent search, respectively, the enhancements (transposition, history and refutation tables) can be evaluated and their performance compared. In both domains, the best savings are achieved with a transposition table (+Trans), which yields a search reduction of almost 50% in the larger trees (depth ≥ 3). The history heuristic (+History) is more of advantage in adversary game tree searching, because there exists a clear notion of the "goodness" of a move, which is missing in single-agent searches. Also, the refutation tables (+Ref) used in adversary game tree searches are slightly more effective than the PV line used in single-agent search, because they contain information not only about the best move continuation, but also about the best alternatives to the PV. This slight advantage becomes more significant in larger trees, where longer subtrees can be pruned.

Combining the enhancements, node reductions of up to 60% can be achieved in both domains. For single-agent search, this is better than can be achieved with an optimal IDA* search, where one finds the goal at the beginning of the last iteration (see Table 1).

5 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 focus on a dataflow model, by and large that model does not account for pragmatic factors like communication and synchronization delays that inevitably plague physical systems, and so addresses few of the difficult issues. The main problems faced by the designers of parallel tree-search systems are:

1. How best to exploit the additional resources (e.g. memory and i/o capability) that may become available with the extra processors.
2. How to distribute the work across the available processors.
3. How to avoid excessive duplication of computation.

Some simple combinatorial problems have no difficulty with point (3) and so, if work distribution is not a problem, ideal or even anomalously good speedup is possible [Lai and Sahni, 1984].

For best performance in game-tree search it is necessary to communicate the improving estimates of the tree value. This can lead to an "acceleration anomaly" when the tree value is found earlier than is possible with a sequential algorithm. However, uniprocessor systems have compensating advantages in that the algorithm can be optimized for best pruning efficiency, while a competing parallel system may not have the right information to achieve the same degree of pruning, hence do more work (higher search overhead). Further, the very fact that pruning occurs makes it impossible to determine in advance how big any piece of work (subtree to be searched) will be, leading to a potentially serious work imbalance and heavy synchronization (waiting for more work) delays.

The standard basis for comparison is *speedup*, defined by

$$\textit{speedup} = \frac{\textit{time taken by a sequential single-processor algorithm}}{\textit{time taken by an N-processor system}}$$

Although simple, this measure can often be misleading, because it depends on the efficiency of the uniprocessor implementation. Thus good speedup may merely reflect a comparison with an inefficient sequential method. Using node count instead of elapsed time, as is common practice in the sequential case, does not help, because it neglects the important communication and synchronization overheads. On the other hand, poor speedup clearly identifies an ineffective parallel system.

The exponential growth rate of the solution space combined with the dispersion of the solutions over a wide range of leaf nodes make parallel search algorithms especially susceptible to anomalous speedup behavior. Clearly, acceleration anomalies are among the welcome properties, whereas slowdown behavior can be reduced (or completely avoided) by an appropriate algorithm design. In the following, we speak of *superunitary speedup*[2]

[2]Note that some authors use the term "super*linear*" when they really mean "super*unitary*". We distinguish between *linear superunitary speedup* (when $\lim_{n\to\infty} S(n)/n > 1$ and finite), and *superlinear* speedup (when $\lim_{n\to\infty} S(n)/n = \infty$). Definitions for slowdown behavior (i.e. subunitary and sublinear speedups) can be derived analogously.

when the speedup is greater than the number of additionally employed processors, and of *subunitary speedup* when a larger multiprocessor system takes more time to execute than a smaller one. Let $t(n)$ be the execution time of an algorithm on n processors, and let $n_1 < n_2$. We define

$$\begin{aligned} \textit{superunitary speedup} &= \frac{t(n_1)}{t(n_2)} > \frac{n_2}{n_1} \\ \textit{subunitary speedup} &= t(n_1) < t(n_2) \end{aligned}$$

With a careful algorithm design, superunitary speedup should occur more often than subunitary speedup. Certain implementations do not exhibit serial-to-parallel slowdown, although superunitary speedup is still possible [Rao and Kumar, 1987, p. 487]. Moreover, Kalé and Saletore [1990] present a method that completely avoids any kind of slowdown behavior in parallel state-space search. This is achieved with a prioritizing scheme that keeps the node expansion sequence close to that of a corresponding sequential search. Such methods may yield satisfactory performance for some cases, but in general more liberal parallel implementations, which exploit better the natural parallelism of the problem domain, are preferred.

In game playing programs, parallelism is not primarily employed to obtain the answer any faster, but to get a more reliable result (e.g. based on a deeper search). Here, the emphasis lies on *scalability* instead of speedup. While speedup holds the problem size constant and increases the system size to get the result earlier, scalability measures the ability to grow both the problem size and the system size at the same time:

$$\textit{scale-up} = \frac{\textit{time taken to solve a problem of size } s \textit{ by a single-processor}}{\textit{time taken to solve a problem of size } (N \times s) \textit{ by an N-processor system}}$$

Ideally, this equation evaluates to 1. The scale-up is then said to be linear.

5.1 Parallel Single Agent Search

Single agent game tree search seems to be more amenable to parallelization than the techniques used in adversary games, because a large proportion of the search space must be fully enumerated – especially when optimal solutions are sought. This can safely be done in parallel without the risk of missing a cut off. Although move ordering also saves nodes expansions, it does not play the same crucial role as in multi-agent game-tree search, where huge parts of the search space are often pruned near the root. For this reason, parallel single agent search techniques usually achieve better speedups than their counterparts in adversary game trees.

Most parallel single agent searches are based on A* or IDA*. As in the sequential case, parallel A* outperforms IDA* on a node count basis, while parallel IDA* needs only linear storage space and runs faster. In addition, cost effective methods exist (e.g. parallel window search in Sec. 5.1.2) that determine non-optimal solutions with even less computing time.

5.1.1 Parallel A*

Given N processors, the simplest way to parallelize A* is to let each processor work on one of the currently best states of the Open list. This approach minimizes the search overhead, as confirmed theoretically by Irani and Shih [1986] and in practice by Kumar *et al.* [1988]. The practical experiments have been run on a shared memory BBN-Butterfly machine with 100 processors, where a search overhead of less than 5% was observed for the traveling salesman problem.

But elapsed time is more important than the node expansion count, because the global Open list is accessed before and after each node expansion, and so memory contention becomes a serious bottleneck. It turns out, that the centralized strategy is only useful in domains where the node expansion time, T_{exp}, is large compared to the time needed for an Open list access, T_{acc}. One such application is the LMSK algorithm (named after its inventors Little, Murty, Sweeney and Karel [1963]) for solving the traveling salesman problem. Almost linear time speedups were achieved [Kumar *et al.*, 1988] only with up to T_{exp}/T_{acc} processors. The saturation point lies well below 20 when the Open list entries are maintained in a global linked list structure. A more sophisticated concurrent heap data structure speeds the Open access, T_{acc}, which allows to increase the parallelism to about 50 processors. Similar results were derived much earlier by Mohan [1982], who ran the LMSK algorithm on the hierarchical Cm* multiprocessor system [Jones *et al.*, 1977]

Distributed strategies using local Open lists alleviate the memory contention problem. But again some communication must be provided to allow processors to share the most promising state descriptors, so that no computing resources are wasted in expanding inferior states. For this purpose Kumar *et al.* use a global "Blackboard" table that holds state descriptors of the currently best nodes. After selecting a state from its local Open list, each processor compares its f-value to that of the states contained in the Blackboard. If the local state is much better or much worse than those stored in the Blackboard, node descriptors are sent or received, so that all processors are concurrently exploring states of almost equal heuristic value. With this scheme, a 69-fold speedup was achieved on an 85-processor BBN Butterfly [Kumar *et al.*, 1988].

Although a Blackboard is not accessed as frequently as a global Open list, it still induces memory contention with increasing parallelism. To alleviate this problem, Huang and Davis [1989] proposed a distributed heuristic search algorithm, *PIA**, which works solely on local data structures. On a uniprocessor, PIA* expands the same nodes as A*, while in the multiprocessor case, it performs a parallel best-first node expansion. The search proceeds by repetitive synchronized iterations, in which processors working on inferior nodes are stopped and reassigned better nodes. To avoid unproductive waiting at the synchronization barriers, the processors are allowed to perform speculative processing. Although Huang and Davis [1989] claim that "this algorithm can achieve almost linear speedup on a large number of processors," it has the same disadvantage as the other parallel A* variants, namely excessive memory requirements.

5.1.2 Parallel IDA*

IDA* (Fig. 2) has proved to be effective, when best-first schemes cannot be used because of excessive memory requirements. Not surprisingly it has also been a popular algorithm to parallelize. Rao *et al.* [1987] proposed *PIDA**, an algorithm with almost linear speedup, even when solving the 15-puzzle with its trivial node expansion cost. PIDA* splits the search space into disjoint parts, so that each processor performs a local cost-bounded depth-first search on its private portion of the state space. When a process has finished its job, it tries to get an unsearched part of the tree from other processors. When no further work can be obtained, all processors detect global termination and compute the minimum of the cost bounds, which is used as a new bound in the next iteration. Note, that superunitary speedup ($\geq N$) is possible when a processor finds a goal node early in the final iteration. In fact, Rao *et al.* [1987] report an *average* superunitary speedup of 9.24 with 9 processors on the 15-puzzle. While such impressive results could not be replicated by others (Rao *et al.* took a non-representative data set of 13 instances only), our PIDA* implementation gave also high (but non-superunitary) speedups on a Transputer system. Perhaps more important is the all-solution-case where no superunitary speedup is possible. Here, Rao *et al.* obtained an average speedup of $0.93N$ with up to $N = 30$ processors on a bus-based multiprocessor architecture (Sequent Balance 21000), which suggests only low multiprocessing overheads (locking, work transfer, termination detection and synchronization).

PIDA* employs a task attraction scheme (illustrated in Fig. 14) for distributing the work among the processors. When a processor becomes idle, it asks a neighbor for a

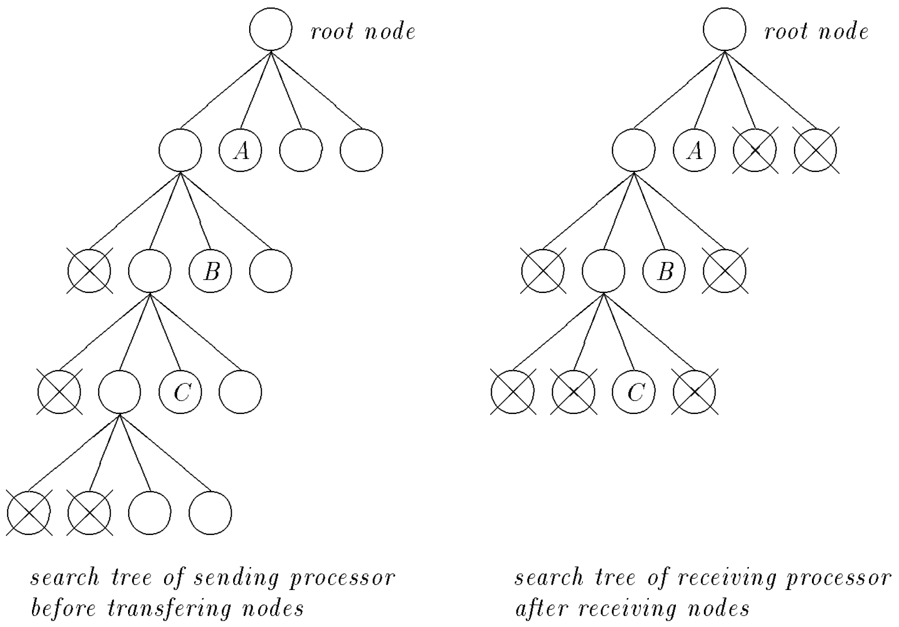


Figure 14: Work distribution scheme of PIDA*

piece of the search space. The donor then splits its depth-first search stack and transfers to the requester some nodes (subtrees) for parallel expansion. The best splitting strategy depends on the regularity (uniform width and height) of the search tree, although short subtrees should never be given away. When the tree is most regular (like in the 15-puzzle) a coarse grained work transfer strategy can be used (e.g. transferring only nodes near the root), otherwise a slice of nodes (e.g. nodes A, B and C in Fig. 14) should be transferred. The left graph in Figure 15 shows the single speedup results of PIDA* executed on a Transputer system with the coarse grained work distribution scheme. We took Korf's [1985] fifty smallest 15-puzzle problem instances and sorted them according to the relative search effort spent in the last iteration (where the goal is found). The straight lines give a quadratic least square fit of the data points. When the sequential version returns a solution after only few node expansions in the last iteration, the processor that expands the "left" tree part in the parallel implementation also finds the goal node in about the same time. Hence speedups are consistently less than linear in the left part of the graph. On the right hand side of the graph, the each game tree is poorly ordered for sequential node expansion, and one of the parallel processors searching a "right" subtree will first find a solution. Since the search is stopped before all of the left tree parts are examined superlinear speedup is common. Overall, we achieved a speedup of 1.78 with two processors, 3.47 with four processors and 4.93 with six proces-

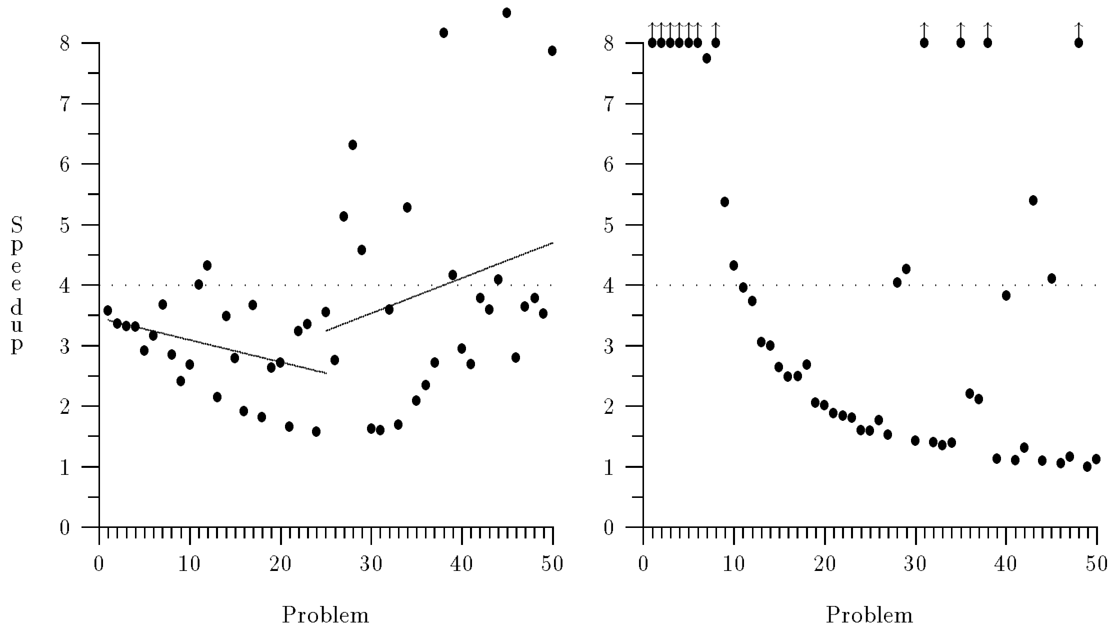


Figure 15: Speedup of PIDA* (left) and PWS (right) with 4 processors

sors. Although this is by no means the maximum achievable performance (little effort was put into tuning our algorithm), this example demonstrates that PIDA* works for small systems. The communication cost is neglectable, even though our algorithm runs only on a uni-directional processor ring (where each message makes a full round through all processors).

Another parallel IDA* approach borrows from Baudet's [1978] parallel window method for searching adversary games (Section 5.2.1). Powley and Korf [1991] adapted this method to single agent search and called it *Parallel Window Search (PWS)*. Their basic idea is to simultaneously start as many iterations as there are processors. This works for a small number or processors, which either expand the tree up to their given thresholds until a solution is found (and the search is stopped), or they completely expand their search space. A global administration scheme then determines the next larger search bound and node expansion starts over again. Clearly, the solution time is limited by the time to complete the goal iteration. The better the node ordering, the faster the execution time. In the extreme case, superunitary speedup can be achieved (e.g., the first ten problems in Fig. 15(b)) with a good heuristic pre-sorting scheme that shifts the goal to the "left" part of the tree. While such domain-dependent sorting schemes are commonly used in sequential heuristic search, they seem to be almost mandatory for PWS.

Note that the first solution found by PWS need not necessarily be optimal. Suboptimal solutions are often found in searches of poorly ordered trees. There a processor working with a higher cut-off bound finds a goal node in a deeper tree level, while other processors are still expanding shallower tree parts (that might contain cheaper solutions). But according to Powley and Korf, PWS is not primarily meant to compete with IDA*, but it "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" [Powley and Korf, 1991]. Compared to PIDA, the degree of parallelism is limited, and it remains unclear, how to apply PWS in domains with variable cost-bound increases.

In summary, PWS and PIDA* complement each other almost ideally. It seems natural to combine both methods to a single search scheme that runs PIDA* on groups of processors administered by a global PWS algorithm.

	PWS	**PIDA***
Scalability	limited	high
Synchronization Overhead	none	between iterations
Communication Overhead[3]	negligible	little
Solution-Quality	limited non-opt.	optimal
Superunitary Speedup	sometimes	sometimes
Subunitary Speedup	very seldom	very seldom

5.1.3 Heuristic Search on SIMD Machines

The largest parallel computers that exist today have a single instruction stream, multiple data stream (SIMD) architecture allowing thousands of simple processing elements to be connected in an economical way. Hillis' *Connection Machine* [Hillis, 1986], as an example, employs $2^{16} = 65536$ bit-serial processing elements that operate synchronously under a single control regime. At any instance, all processors execute the same instruction (possibly on a different data item), or no instruction at all. SIMD machines are designed for "data parallel" applications, where operations are concurrently applied to all elements of a large data set. Clearly, tree searching tasks involve processing a group of data items (=nodes), and this can be done in parallel, but the number of active nodes

[3]The amount of communication needed depends on the work distribution scheme. More communication is needed with a fine-grained work distribution, while a coarse grained work distribution requires fewer messages (but may induce unbalanced work load). Note that the choice of the work distribution scheme also affects the probability of acceleration anomalies.

varies dynamically during search. Hence, load balancing poses a major challenge in the design of efficient tree searching applications for SIMD machines.

There exist two iterative-deepening searches for SIMD-machines: *SIDA** by Powley *et al.* [1992; 199x] and *IDPS* by Mahanti and Daniels [1992]. While differing in detail, both schemes work in a similar way. They initially distribute a set of frontier nodes to the processors and carry out a series of IDA* iterations on these nodes thereafter. Each processing element searches its assigned subtree independently in a depth-first fashion. When a specified number of processors finish their subtree and become idle, the search is stopped and the remaining nodes are redistributed in a load-balancing phase. Autonomous node expansion phases alternate with load-balancing phases. When all subtrees have been searched up to the given cost bound, some additional global load balancing is performed and the next iteration is started with an increased cost bound. This continues until one processor finds a goal node.

The two schemes differ only in subtle details, like: how to best obtain an equally balanced initial work distribution (e.g. by node contraction), how to do the load balancing (e.g. which nodes to share) and when to start a load balancing phase (e.g. static vs. dynamic triggering). Powley *et al.* report speedup factors of 5400 and 9300 on a Connection Machine with 8K and 16K processors, which correspond to efficiencies (=speedup/processors) of 67% and 57%. Mahanti and Daniels achieved even more favorable overall efficiencies of 92% and 76% with their IDPS implementation on the same machine. But what has been achieved? A speedup factor of 9300 sounds impressive, but in practice the result is delivered only twelve times faster than on a SUN SparcStation 1 – at the cost of 65535 additional processing elements. This state of affairs will eventually change when the algorithms have been successfully implemented on faster SIMD machines, since then the efficiency figures (57% and 76%) become more important.

Another SIMD search scheme, Parallel-Retracting-A*, *PRA** [Evett *et al.*, 1990], builds on A* rather than IDA*. It maps the search space to the processing elements, so that each node is sent to a unique processor (determined by a hashing function) for further expansion. By this means, each processor autonomously expands its dedicated nodes, and also eliminates duplicates without further interrogation. When the private storage space is exhausted, a node retraction procedure is invoked to reduce the active nodes. While PRA* examines significantly fewer nodes than IDA* [Evett *et al.*, 1990, Table 1], the search is slowed down by the frequent need for communication and by time-consuming node-retraction operations.

A brute-force depth-first search for a "blocks in a box" puzzle has also been implemented on the Connection Machine [Frye and Myczkowski, 1990]. It rebalances the work-load when the ratio of active to idle processors drops below 2/3. Since the re-balancing phases take about 50% of the execution time, the overall efficiency inevitably drops to less than 50%. Unfortunately the paper of Frye and Myczkowski contains no other speedup measures or efficiency data.

5.2 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 critical tree that must be searched. Since that tree is well-defined and has regular properties (see Fig. 7), it is easy to generate. In their simulation of the MWF method Akl *et al.* [1982] consider the merits of searching the critical game tree in parallel, with the balance of the tree being generated algorithmically and searched quickly by simple tree splitting. Fishburn and Finkel [1982] also favor this method and provide some analysis. The first subtree of the critical 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 and simulated [Marsland and Campbell, 1982], and later tested [Marsland and Popowich, 1985]. Even so, 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 ignores the true shape of the average game tree, and so is at its best with shallow searches, where the pruning imbalance from the so called "deep cutoffs" has less effect. Other working experience includes the first parallel chess program by Newborn, who later presented performance results [Newborn, 1985] [Newborn, 1988]. For practical reasons Newborn only split the tree 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 is also an important part of changing search modes and in managing transposition tables.

5.2.1 Parallel Window Search

In an early paper on parallel game-tree search, Baudet [1978] suggests partitioning the range of the alpha-beta window rather than the tree. In his algorithm, all processors search the whole tree, but each with a different, non-overlapping, alpha-beta window. The total range of values is subdivided into p smaller intervals (where p is the number of processors), so that approximately one third is covered. The advantage of this method is that the processor having the true minimax value inside its narrow window will complete more quickly than a sequential algorithm running with a full window. Even the unsuccessful processors return a result: They determine whether the true minimax value lies below or above their assigned search window, providing important information for re-scheduling idle processors until a solution is found.

Its low communication overhead and lack of synchronization needs are among the positive aspects of Baudet's approach. On the negative side, however, is the limited maximal speedup, no matter how many processors are being employed. From Baudet's [1978] data one can estimate a maximum speedup of 5 to 6 when using infinitely many processors. This is because even in the best case (when the successful processor uses a minimal window) at least the critical game tree (with $W^{\lceil \frac{D}{2} \rceil} + W^{\lfloor \frac{D}{2} \rfloor} - 1$ leaves) must be expanded. In practice, parallel window search can be effectively employed on systems with only two or three processors.

5.2.2 Advanced Tree-splitting Methods

Results from fully recursive versions of PVSplit were presented for the experimental *Parabelle* chess program [Marsland and Popowich, 1985] and for *ParaPhoenix* [Marsland *et al.*, 1985]. These results confirmed the earlier simulations and offered some insight into a major problem: In an N-processor system, $N - 1$ processors may be idle for an inordinate amount of time, thus inducing a high synchronization overhead for large systems. Moreover, the synchronization overhead increases as more processors are added, accounting for most of the total losses, because the search overhead (=number of unnecessary node expansions) becomes almost constant for the larger systems, see Figure 16. This led to the development of variations that dynamically assign processors to the search of the principal variation. Notable is the work of Schaeffer [1989a], which uses a loosely coupled network of workstations, and Hyatt *et al.*'s [1989] implementation for a shared-memory computer. These dynamic splitting works have attracted growing attention through 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

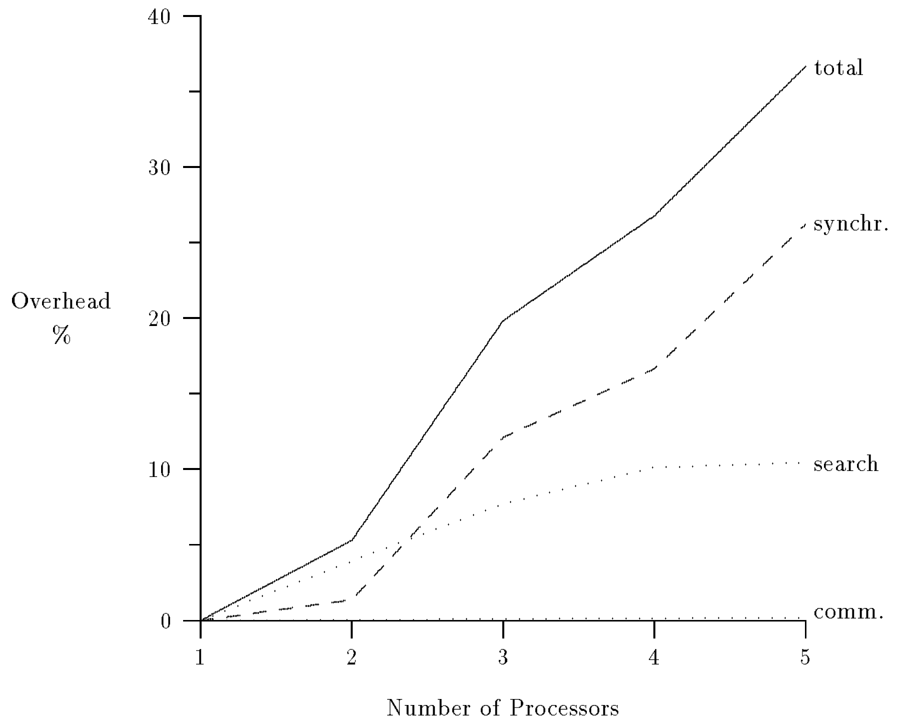


Figure 16: Overheads for 7-Ply Searches in a Chess Program (ParaPhoenix)

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 [1989a]. These later systems dynamically try to identify the ALL nodes of Figure 7 (where every successor must be searched), and search them in parallel, leaving the CUT nodes (where only a few successors might be examined) 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 critical 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] 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 (for example, by omitting the important node-ordering mechanisms) the speedup figure may look good. 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 potential advantages from the use of heuristic methods (such as hashing to replace linked lists) have not been fully exploited.

5.2.3 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 the case of average game trees, or even better the strongly ordered model (where extensive pruning occurs), which result in asymmetric trees and a significant work distribution problem.

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 critical 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, and the "helpful master" scheme to eliminate the idle time of masters waiting for their slaves' results.

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 single-agent IDA* search. The unexplored part of the trees are marked and are dynamically assigned to any idle processor. In principle, this method (illustrated in Fig. 14) could also be used for deterministic adversary game trees. 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 confident is Hsu's recent thesis [Hsu, 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, based on 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

[Abramson, 1989] B. Abramson. Control strategies for two-player games. *ACM Computing Surveys*, 21(2):137–162, 1989.

[Akl and Newborn, 1977] S.G. Akl and M.M. Newborn. The principal continuation and the killer heuristic. In *1977 ACM Ann. Conf. Procs.*, pages 466–473, Seattle, 1977. (New York: ACM).

[Akl *et al.*, 1982] 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.

[Allis *et al.*, 1991] V.L. Allis, M. van der Meulen, and H.J. van den Herik. Alpha-beta conspiracy number search. In D.F. Beal, editor, *Advances in Computer Chess 6*, pages 73–95. Ellis Horwood, 1991.

[Anantharaman *et al.*, 1988] T. Anantharaman, M. Campbell, and F.h. 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.

[Anantharaman, 1991] T. Anantharaman. Confidently selecting a search heuristic. *Int. Computer Chess Assoc. J.*, 14(1):3–16, 1991.

[Ballard, 1983] B.W. Ballard. The *-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, 21(3):327–350, 1983.

[Baudet, 1978] G.M. Baudet. *The Design and Analysis of Algorithms for Asynchronous Multiprocessors.* PhD thesis, Dept. of Computing Science, Carnegie Mellon Univ., Pittsburgh, 1978.

[Beal, 1989] D. Beal. Experiments with the null move. In D. Beal, editor, *Advances in Computer Chess 5*, pages 65–79. Elsevier, 1989. Revised as "A Generalized Quiescence Search Algorithm" in *Artificial Intelligence*, 43(1):85–98, 1990.

[Bellman and Dreyfus, 1962] R. Bellman and S. Dreyfus. *Applied Dynamic Programming.* Princeton Univ. Press, Princeton, N.J., 1962.

[Berliner, 1979] H.J. Berliner. The B* tree search algorithm: A best first proof procedure. *Artificial Intelligence*, 12(1):23–40, 1979.

[Bhattacharya and Bagchi, 1986] S. Bhattacharya and A. Bagchi. AI search techniques for game trees: Making effective use of available memory and processors. In *Procs. AAAI-86 Conf.*, pages 163–167, Philadelphia, 1986.

[Bhattacharya and Bagchi, 1990] S. Bhattacharya and A. Bagchi. Unified recursive schemes for search in game trees. Technical Report WPS-144(90), Indian Institute of Management, Calcutta, Post Box 16757, Alipore P.O., Calcutta 700 027, India, 1990.

[Birmingham and Kent, 1977] J.A. Birmingham and P. Kent. Tree-searching and tree-pruning techniques. In M. Clarke, editor, *Advances in Computer Chess 1*, pages 89–107. Edinburgh Univ. Press, Edinburgh, 1977.

[Brudno, 1963] 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.

[Campbell and Marsland, 1983] M.S. Campbell and T.A. Marsland. A comparison of minimax tree search algorithms. *Artificial Intelligence*, 20(4):347–367, 1983.

[Chakrabarti *et al.*, 1989] P.P. Chakrabarti, S. Ghose, A. Acharya, and S.C. de Sarkar. Heuristic search in restricted memory. *Artificial Intelligence*, 41(2):197–221, 1989.

[Dechter and Pearl, 1985] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *J. of the ACM*, 32(3):505–536, 1985.

[Doran and Michie, 1966] J.E. Doran and D. Michie. Experiments with the graph traverser program. *Procs. Royal Society A*, 294:235–259, 1966.

[Evett *et al.*, 1990] M. Evett, J. Hendler, A. Mahanti, and D. Nau. PRA*: A memory-limited heuristic search procedure for the Connection Machine. In *3rd Symp. Frontiers Mass. Par. Comp.*, pages 145–149, 1990.

[Feldmann *et al.*, 1990] R. Feldmann, B. Monien, P. Mysliwietz, and O. Vornberger. Distributed game tree search. In V. Kumar, P.S. Gopalakrishnan, and L. Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*, pages 66–101. Springer-Verlag, New York, 1990.

[Felten and Otto, 1988] E.W. Felten and S.W. Otto. A highly parallel chess program. *Procs. Int. Conf. on 5th Generation Computer Systems*, pages 1001–1009, 1988.

[Ferguson and Korf, 1988] C. Ferguson and R.E. Korf. Distributed tree search and its application to alpha-beta pruning. In *Proc. 7th Nat. Conf. on Art. Intell. (Vol 1)*, pages 128–132, Saint Paul, 1988. (Los Altos: Kaufmann).

[Finkel and Fishburn, 1982] R.A. Finkel and J.P. Fishburn. Parallelism in alpha-beta search. *Artificial Intelligence*, 19:89–106, 1982.

[Fishburn, 1981] J.P. Fishburn. Analysis of speedup in distributed algorithms. Technical Report TR 431, Univ. of Wisconsin, Madison, Wis., 1981. Also 1984 book, UMI Press, Ann Arbor, MI, 118 pages.

[Frye and Myczkowski, 1990] R. Frye and J. Myczkowski. Exhaustive search of unstructured trees on the Connection Machine. Technical Report TMC-196, Thinking Machines Corporation, 1990. Also in J. Par. and Distr. Comp., ?(?):?-?, 199?

[Gillogly, 1972] J.J. Gillogly. The technology chess program. *Artificial Intelligence*, 3(1–4):145–163, 1972. Also in D. Levy, editor, *Computer Chess Compendium*, pages 67–79, Springer-Verlag, 1988.

[Ginsberg and Harvey, 1992] M.L. Ginsberg and W.D. Harvey. Iterative broadening. *Artificial Intelligence*, 55:367–383, 1992.

[Guiliano *et al.*, 1990] 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, editors, *Parallel Algorithms for Machine Intelligence and Vision*, pages 182–231. Springer-Verlag, New York, 1990.

[Hart *et al.*, 1968] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the determination of minimum cost paths. *IEEE Trans. on Sys. Sc. and Cybern.*, 4(2):100–107, 1968.

[Hillis, 1986] W.D. Hillis. *The Connection Machine*. MIT Press, 1986.

[Hsu, 1990] F.h. Hsu. Large scale parallelization of alpha-beta search: An algorithmic and architectural study with computer chess. Technical Report CMU–CS–90–108, Carnegie-Mellon Univ., Pittsburgh, 1990.

[Huang and Davis, 1989] S. Huang and L.R. Davis. Parallel iterative A* search: An admissible distributed search algorithm. In *Procs. 11th Int. Joint Conf. on AI (vol 1)*, pages 23–29, Detroit, 1989. (Los Altos: Kaufmann).

[Hyatt *et al.*, 1989] R.M. Hyatt, B.W. Suter, and H.L. Nelson. A parallel alpha/beta tree searching algorithm. *Parallel Computing*, 10(3):299–308, 1989.

[Irani and Shih, 1986] K.B. Irani and Y. Shih. Parallel A* and AO* algorithms: An optimality criterion and performance evaluation. In *Procs. Int. Conf. on Par. Comp.*, pages 274–277, 1986.

[Jones *et al.*, 1977] A.K. Jones, R. Chansler, I. Durham, P. Feiler, and K. Schwans. Software management of Cm* – a distributed multiprocessor. In *AFIPS Conf. Procs.*, volume 46, pages 557–663. AFIPS Press, 1977.

[Kaindl, 1983] H. Kaindl. Searching to variable depth in computer chess. In *Procs. 8th Int. Joint Conf. on Art. Intell.*, pages 760–762, Karlsruhe, Germany, 1983. (Los Altos: Kaufmann).

[Kaindl, 1989] H. Kaindl. *Problemlösen durch heuristische Suche in der Artificial Intelligence.* Springer-Verlag, Wien, 1989.

[Kaindl, 1990] H. Kaindl. Tree searching algorithms. In T.A. Marsland and J. Schaeffer, editors, *Computers, Chess, and Cognition*, pages 133–158. Springer-Verlag, New York, 1990.

[Kalé and Saletore, 1990] L.V. Kalé and V.A. Saletore. Parallel state-space search for a first solution with consistent linear speedups. *Intl. J. Par. Progr.*, 19(4):251–293, 1990.

[Knuth and Moore, 1975] D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[Korf, 1985] R.E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[Korf, 1989] R.E. Korf. Generalized game trees. In *Procs. 11th Int. Joint Conf. on AI (vol 1)*, pages 328–333, Detroit, 1989. (Los Altos: Kaufmann).

[Korf, 1990] R.E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42:189–211, 1990.

[Korf, 1991] R.E. Korf. Multi-player alpha-beta pruning. *Artificial Intelligence*, 48(1):99–111, 1991.

[Kumar and Rao, 1990] V. Kumar and V.N. Rao. Scalable parallel formulations of depth-first search. In V. Kumar, P.S. Gopalakrishnan, and L. Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*, pages 1–41. Springer-Verlag, New York, 1990.

[Kumar *et al.*, 1988] V. Kumar, K. Ramesh, and V. Nageshwara-Rao. Parallel best-first search of state-space graphs: A summary of results. In *Procs. 7th Nat. Conf. on Art. Int., AAAI-88*, pages 122–127, Saint Paul, 1988. (Los Altos: Kaufmann).

[Lai and Sahni, 1984] T.H. Lai and S. Sahni. Anomalies in parallel branch-and-bound algorithms. *Comm. ACM*, 27:594–602, 1984.

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

[Little *et al.*, 1963] J.D.M. Little, K.G. Murty, D.W. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11:972–989, 1963.

[Luckhardt and Irani, 1986] C.A. Luckhardt and K.B. Irani. An algorithmic solution of n-person games. In *Procs. AAAI-86 Conf.*, pages 158–162, Philadelphia, 1986.

[Mahanti and Daniels, 1992] A. Mahanti and C.D. Daniels. IDPS: A massively parallel heuristic search algorithm. In *6th IEEE Intl. Par. Proc. Symp.*, pages 220–223, Beverly Hills, CA, 1992.

[Mahanti *et al.*, 1992a] A. Mahanti, S. Ghosh, D.S. Nau, A.K. Pal, and L. Kanal. Performance of IDA* on trees and graphs. In *Procs. 10th Nat. Conf. on Artificial Intelligence (AAAI-92)*, pages 539–544, San Jose, CA, 1992.

[Mahanti *et al.*, 1992b] A. Mahanti, D.S. Nau, S. Ghosh, and L. Kanal. An efficient iterative threshold heuristic search algorithm. Technical Report CS-TR-2853, Comp. Sci. Dept., University of Maryland, College Park, 1992.

[Marsland and Campbell, 1981] T.A. Marsland and M. Campbell. A survey of enhancements to the alpha-beta algorithm. *ACM Annual Conf. Procs.*, pages 109–114, 1981.

[Marsland and Campbell, 1982] T.A. Marsland and M. Campbell. Parallel search of strongly ordered game trees. *ACM Computing Surveys*, 14(4):533–551, 1982.

[Marsland and Popowich, 1985] T.A. Marsland and F. Popowich. Parallel game-tree search. *IEEE Trans. on Pattern Anal. and Mach. Intell.*, 7(4):442–452, 1985.

[Marsland and Srimani, 1986] T.A. Marsland and N. Srimani. Phased state space search. In *Procs. ACM/IEEE Fall Joint Computer Conf.*, pages 514–518, Dallas, 1986.

[Marsland *et al.*, 1985] T.A. Marsland, M. Olafsson, and J. Schaeffer. Multiprocessor tree-search experiments. In D. Beal, editor, *Advances in Computer Chess 4*, pages 37–51. Pergamon Press, Oxford, 1985.

[Marsland *et al.*, 1987] T.A. Marsland, A. Reinefeld, and J. Schaeffer. Low overhead alternatives to SSS*. *Artificial Intelligence*, 31(2):185–199, 1987.

[Marsland, 1983] T.A. Marsland. Relative efficiency of alpha-beta implementations. In *Procs. 8th Int. Joint Conf. on Art. Intell.*, pages 763–766, Karlsruhe, Germany, 1983. (Los Altos: Kaufmann).

[Marsland, 1987] T.A. Marsland. Computer chess methods. In E. Shapiro, editor, *Encyclopedia of Art. Intell.*, pages 159–171. Wiley, 1987. See also, 2nd Edition, "Computer chess and search," pp. 224–241, 1992.

[McAllester, 1988] D. McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence*, 35(3):287–310, 1988.

[Mohan, 1982] J. Mohan. A study in parallel computation – the traveling salesman problem. Technical Report CMU-CS-82-136, Comp. Sci. Dept., Carnegie Mellon Univ., Pittsburgh, 1982.

[Musczycka and Shinghal, 1985] 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.

[Nelson, 1985] H.L. Nelson. Hash tables in Cray Blitz. *Int. Computer Chess Assoc. J.*, 8(1):3–13, 1985.

[Newborn, 1985] M.M. Newborn. A parallel search chess program. *Procs. ACM Ann. Conf.*, pages 272–277, 1985. See also (March 1982) TR SOCS 82.3, Computer Science, McGill Univ., Montreal, Canada, 20pp.

[Newborn, 1988] M.M. Newborn. Unsynchronized iteratively deepening parallel alpha-beta search. *IEEE Trans. on Pattern Anal. and Mach. Intell.*, 10(5):687–694, 1988.

[Newell *et al.*, 1958] 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, editors, *Computers and Thought*, pages 39–70, 1963.

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

[Palay, 1985] A.J. Palay. *Searching with Probabilities.* Pitman, 1985. See earlier Ph.D. Thesis (1983), Comp. Sci. Dept., Carnegie-Mellon Univ., Pittsburgh, 152pp.

[Patrick *et al.*, 1992] B.P. Patrick, M. Almulla, and M.M. Newborn. An upper bound on the time complexity of iterative-deepening A*. *Annals of Mathematics and Artificial Intelligence*, 5, 1992.

[Pearl, 1980] J. Pearl. Asymptotic properties of minimax trees and game searching procedures. *Artificial Intelligence*, 14(2):113–138, 1980.

[Pearl, 1984] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, Reading, MA, 1984.

[Pohl, 1970] I. Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1:193–204, 1970.

[Pohl, 1971] I. Pohl. Bi-directional search. In B. Meltzer and D. Michie, editors, *Machine Intelligence 6*, pages 127–140. American Elsevier, New York, 1971.

[Powley and Korf, 1991] C. Powley and R.E. Korf. Single-agent parallel window search. *IEEE Trans. on Pattern Anal. and Mach. Intell.*, 13(5):466–477, 1991.

[Powley *et al.*, 1990] C. Powley, C. Ferguson, and R.E. Korf. Parallel heuristic search: Two approaches. In V. Kumar, P.S. Gopalakrishnan, and L. Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*, pages 42–65. Springer-Verlag, New York, 1990.

[Powley *et al.*, 1992] C. Powley, C. Ferguson, and R.E. Korf. Parallel tree search on a SIMD machine. In *3rd IEEE Symp. Par. and Distr. Proc.*, pages 249–256, Dallas, 1992.

[Powley *et al.*, 199x] C. Powley, C. Ferguson, and R.E. Korf. Depth-first heuristic search on a SIMD machine. *Artificial Intelligence*, 199x. to appear.

[Rao and Kumar, 1987] V.N. Rao and V. Kumar. Parallel depth first search. Part I. Implementation. *Intl. J. Par. Progr.*, 16(6):479–499, 1987.

[Rao *et al.*, 1987] V.N. Rao, V. Kumar, and K. Ramesh. A parallel implementation of Iterative-Deepening A*. In *Procs. 6th Nat. Conf. on Art. Intell.*, pages 178–182, Seattle, 1987.

[Rao *et al.*, 1991] V.N. Rao, V. Kumar, and R.E. Korf. Depth-first vs. best-first search. In *Procs. 9th Nat. Conf. on Artificial Intelligence (AAAI-91)*, pages 434–440, Anaheim, CA, 1991.

[Ratner and Warmuth, 1986] D. Ratner and M. Warmuth. Finding a shortest solution to the $N \times N$ extension of the 15-puzzle is intractable. In *Procs. 5th Nat. Conf. on Art. Int., AAAI-86*, pages 168–172, Philadelphia, Pa., 1986. (Los Altos: Kaufmann).

[Reinefeld and Marsland, 1987] A. Reinefeld and T.A. Marsland. A quantitative analysis of minimal window search. In *Procs. 10th Int. Joint Conf. on Art. Intell.*, pages 951–954, Milan, Italy, 1987. (Los Altos: Kaufmann).

[Reinefeld and Marsland, 1991] A. Reinefeld and T.A. Marsland. Memory functions in iterative-deepening search. Technical Report FBI-HH-M-198/91, Univ. Hamburg, Fachbereich Informatik, Germany, 1991.

[Reinefeld *et al.*, 1985] A. Reinefeld, J. Schaeffer, and T.A. Marsland. Information acquisition in minimal window search. *Procs. 9th Int. Joint Conf. on Art. Intell.*, pages 1040–1043, 1985.

[Reinefeld, 1983] A. Reinefeld. An improvement of the Scout tree-search algorithm. *Int. Computer Chess Assoc. J.*, 6(4):4–14, 1983.

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

[Rivest, 1988] R.L. Rivest. Game tree searching by min/max approximation. *Artificial Intelligence*, 34(1):77–96, 1988.

[Roizen and Pearl, 1983] I. Roizen and J. Pearl. A minimax algorithm better than Alpha-Beta? yes and no. *Artificial Intelligence*, 21(1):199–220, 1983.

[Russell, 1992] S. Russell. Efficient memory-bounded search methods. In *Procs. of the European Conf. on Art. Intell.*, pages 1–5, Vienna, Austria, 1992.

[Sarkar *et al.*, 1991] U.K. Sarkar, P.P. Chakrabarti, S. Ghose, and S.C. de Sarkar. Reducing re-expansions in iterative deepening search by controlled cutoff. *Artificial Intelligence*, 50(2):207–221, 1991.

[Schaeffer, 1983] J. Schaeffer. The history heuristic. *Int. Computer Chess Assoc. J.*, 6(3):16–19, 1983.

[Schaeffer, 1986] J. Schaeffer. *Experiments in Search and Knowledge.* PhD thesis, Univ. of Waterloo, Waterloo, Canada, 1986. Also Comp. Sci. Tech. Rept. TR 86-12, Univ. of Alberta, July 1986.

[Schaeffer, 1989a] J. Schaeffer. Distributed game-tree search. *J. of Parallel and Distributed Computing*, 6(2):90–114, 1989.

[Schaeffer, 1989b] J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Trans. on Pattern Anal. and Mach. Intell.*, 11(11):1203–1212, 1989.

[Schaeffer, 1990] J. Schaeffer. Conspiracy numbers. *Artificial Intelligence*, 43(1):67–84, 1990.

[Scott, 1969] J.J. Scott. A chess-playing program. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 255–265. Edinburgh Univ. Press, 1969.

[Sen and Bagchi, 1989] A.K. Sen and A. Bagchi. Fast recursive formulations for best-first search that allow controlled use of memory. In *Procs. 11th Int. Joint Conf. on AI*, pages 297–302. (Los Altos: Kaufmann), 1989.

[Shams *et al.*, 1991] R. Shams, H. Kaindl, and H. Horacek. Using aspiration windows for minimax algorithms. In *Procs. 8th Int. Joint Conf. on Art. Intell.*, pages 192–197, Sydney, Australia, 1991. (Los Altos: Kaufmann).

[Slagle and Dixon, 1969] J.R. Slagle and J.K. Dixon. Experiments with some programs that search game trees. *J. ACM*, 16(2):189–207, 1969.

[Slate and Atkin, 1977] D.J. Slate and L.R. Atkin. Chess 4.5 - the Northwestern University Chess Program. In P. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer-Verlag, 1977.

[Steinberg and Solomon, 1990] I. Steinberg and M. Solomon. Searching game trees in parallel. In *Procs. Int. Conf. on Parallel Processing (vol 3)*, pages 9–17, University Park, PA, 1990.

[Stickel and Tyson, 1985] M.E. Stickel and W.M. Tyson. An analysis of consecutively bounded depth-first search with applications in automated deduction. In *Procs. 9th Int. Joint Conf. on Art. Intell.*, pages 1073–1075, Los Angeles, 1985. (Los Altos: Kaufmann).

[Stockman, 1979] G.C. Stockman. A minimax algorithm better than alpha-beta? *Artificial Intelligence*, 12(20):179–196, 1979.

[von Neumann and Morgenstern, 1944] John von Neumann and O. Morgenstern. *The Theory of Games and Economic Behavior.* Princeton Univ. Press, N.J., 1944.

[Vornberger and Monien, 1987] O. Vornberger and B. Monien. Parallel alpha-beta versus parallel SSS*. In *Procs. IFIP Conf. on Distributed Processing*, pages 613–625. (Amsterdam: North Holland), 1987.

[Wah, 1991] B.W. Wah. MIDA*: An IDA* search with dynamic control. Technical Report UILU-ENG-91-2216, CRHC-91-9, Coordinated Science Laboratory, Univ. of Illinois at Urbana, Champaign, Urbana, Ill., 1991.

[Warnock and Wendroff, 1988] T. Warnock and B. Wendroff. Search tables in computer chess. *Int. Computer Chess Assoc. J.*, 11(1):10–13, 1988.

[Zobrist, 1970] A.L. Zobrist. A new hashing method with applications for game playing. Technical Report TR 88, Comp. Sci. Dept., Univ. of Wisconsin, Madison, 1970. Also in *Int. Computer Chess Assoc. J.*, 13(2):169–173, 1990.