Search

D. Kopec and T.A. Marsland

Introduction

AI efforts to solve problems with computers which humans routinely handle by employing innate cognitive abilities, pattern recognition, perception and experience, invariably must turn to considerations of search. This Chapter explores search methods in AI, including both blind exhaustive methods and informed heuristic and optimal methods, along with some more recent findings. The search methods covered include (for non-optimal, uninformed approaches) State-Space Search, Generate and Test, Means-ends Analysis, Problem Reduction, And/Or Trees, Depth First Search and Breadth First Search. Under the umbrella of heuristic (informed) methods we discuss are Hill Climbing, Best First Search, Bi-directional Search, and the A* Algorithm. Tree Searching algorithms for games have proven to be a rich source of study and provide empirical data about heuristic methods. Included here are the SSS* algorithm, variations on the alpha-beta minimax algorithm, and the use of iterative deepening.

Since many of these methods are computationally intensive, the second half of the Chapter focuses on parallel methods. The importance of parallel search is presented through an assortment of relatively recent parallel algorithms including the Parallel Iterative Deepening Algorithm (PIDA*), Principal Variation Splitting (PVSplit), and the Young Brothers Wait Concept. Coincident with the continuing price-performance improvement of small computers is growing interest in re-implementing 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 an 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. Parallel methods are important not only for single-agent search, but also through 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 we cover some of the dynamic tree-splitting methods that have evolved. One source of extra computing power comes through the use of parallel processing on a multicomputer.

1 Uninformed Search Methods

1.1 Search Strategies

All search methods in computer science share in common three necessities: 1) a world model or database of facts based on a choice of representation providing the current state, as well as other possible states and a goal state; 2) a set of operators that defines possible transformations of states, and 3) a control strategy which determines how transformations amongst states are to take place by applying operators. Reasoning about the current state to identify a state which is closer to a goal is known as forward reasoning. Working backwards to a current state from a goal state is called backward reasoning. As such it is possible to make distinctions between bottom up and top down approaches to problem solving. Bottom up is often "goal oriented" -- that is reasoning backwards from a goal state to solve intermediary sub-goal states. Top down or data-driven reasoning is based on simply being able to get to a state which is defined as closer to a goal state than the current state. Often application of operators to a problem state may not lead directly to a goal state and some **backtracking** may be necessary before a goal state can be found (Barr & Feigenbaum, 1981).

1.2 State Space Search

Exhaustive search of a problem space (or search space) is often not feasible or practical due to the size of the problem space. In some instances it is, however, necessary. More often, we are able to define a set of legal transformations of a state space (moves in the world of games) from which those that are more likely to bring us closer to a goal state are selected while others are never explored further. This technique in problem solving is known as split and prune. In AI the technique that emulates this approach is called **generate and test.** The basic method is:

```
Repeat
      Generate a candidate solution
      Test the candidate solution
Until a satisfactory solution is found, or
      no more candidate solutions can be generated:
If an acceptable solution is found, announce it;
      Otherwise, announce failure.
```

Figure 1: Generate and Test Method

Good generators are complete and will eventually produce all possible solutions, while not proposing redundant ones. They are also informed; that is, they will employ additional information to constrain the solutions they propose.

Means-ends analysis is another state space technique whose purpose is to reduce the difference (distance) between a current state and a goal state. Determining "distance" between any state and a goal state can be facilitated by difference-procedure tables, which can effectively prescribe what the next state might be. To perform means-ends analysis:

```
Repeat
      Describe the current state, the goal state,
      and the difference between the two.
      Use the difference between the current state and goal state,
      to select a promising transformation procedure.
      Apply the promising procedure and update the current state.
Until the GOAL is reached or
      no more procedures are available
If the GOAL is reached, announce success;
Otherwise, announce failure.
```

Figure 2: Means-Ends Analysis

The technique of problem reduction is another important approach in AI. That is, solve a complex or larger problem by identifying smaller manageable problems (or subgoals), which you know can be solved in fewer steps.

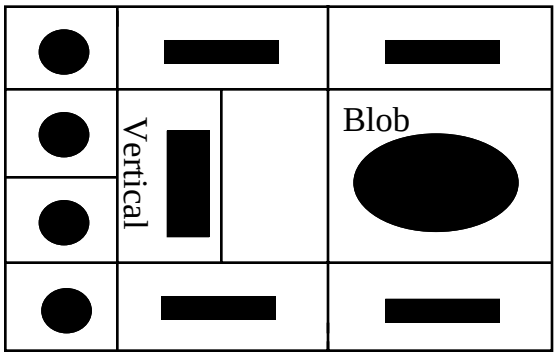

Figure 3: Problem Reduction and The Sliding Block Puzzle

For example, **Figure 3** shows the sliding block puzzle "Donkey" which has been known for over 100 years. Subject to constraints on the movement of "pieces" in the sliding block puzzle, the task is to slide the Blob around the Vertical Bar with the goal of moving it to the other side. The Blob occupies four spaces and needs two adjacent vertical or horizontal spaces in order to be able to move, while the Vertical Bar needs two adjacent empty vertical spaces to move left or right, or one empty space above or below it to move up or down. The Horizontal Bars' movements are complementary to the Vertical Bar. Likewise, the circles can move to any empty space around them in a horizontal or vertical line. A relatively uninformed state space search can result in over 800 moves for this problem to be solved, with plenty of backtracking necessary. By problem reduction, resulting in the subgoal of trying the get the Blob on the two rows above or below the vertical bar, it is possible to solve this puzzle in just 82 moves!

Another example of a technique for problem reduction is called **AND/OR Trees**. Here the goal is to find a solution path to a given tree by applying the following rules:

A node is solvable if --

```
1. it is a terminal node (a primitive problem),
2. it is a nonterminal node whose successors are AND nodes that are all solvable,
      OR
3. it is a nonterminal node whose successors are OR nodes and least one of them
      is solvable.

Similarly, a node is unsolvable if --

1. it is a nonterminal node that has no successors (a nonprimitive problem to
      which no operator applies),
```

2. it is a nonterminal node whose successors are AND nodes and at least one of them is unsolvable, or
3. it is a nonterminal node whose successors are OR nodes and all of them are unsolvable.

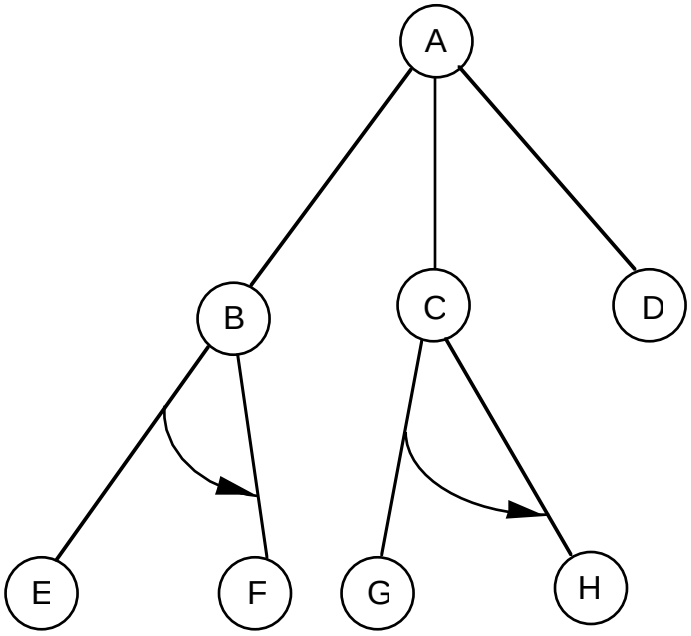

Figure 4: And/Or Tree

In Figure 4 nodes B and C serve as exclusive parents to sub-problems EF and GH, respectively. One way of viewing the tree is with nodes B, C, and D serving as individual, alternative sub-problems representing OR nodes. Node pairs E & F and G & H, respectively, with curved arrowheads connecting them, represent AND nodes. That is, to solve problem B you must solve both sub-problems E and F. Likewise, to solve sub-problem C, you must solve subproblems G and H. Solution paths would therefore be: {A-B-E-F}, {A-C-G-H}, and {A-D}. In the special case where no AND nodes occur, we have the ordinary graph occurring in a state space search. However the presence of AND nodes distinguishes AND/OR Trees (or graphs) from ordinary state structures, which call for their own specialized search techniques. Typical problems tackled by AND/OR trees include games or puzzles, and other well-defined state-space goal oriented problems, such as robot planning, movement through an obstacle course, or setting a robot the task of reorganizing blocks on a flat surface.

1.2.1 Breadth First Search

One way to view search problems is to consider all possible combinations of subgoals, by treating the problem as a tree search. **Breadth First Search** always explores nodes closest to the root node first, thereby visiting all nodes at a given layer first before moving to any longer paths. It pushes uniformly into the search tree. Because of memory requirements, Breadth First Search is only practical on shallow trees, or those with an extremely low branching factor. It is therefore not used much used in practice, except as a basis for such **best-first search** algorithms such as A* and SSS*.

1.2.2 Depth First Search

Depth First Search (DFS) is one of the most basic and fundamental Blind Search Algorithms. It is used for bushy trees (with high branching factor) where a potential solution does not lie too deeply down the tree. That is "DFS is a good idea when you are confident that all partial paths either reach dead ends or become complete paths after a reasonable number of steps". In contrast, "DFS is a bad idea if there are long paths, particularly indefinitely long paths, that neither reach dead ends nor become complete paths" (Winston, 1992). To conduct a DFS:

```
(1) Put the Start Node on the list called OPEN.
(2) If OPEN is empty, exit with failure; otherwise continue.
(3) Remove the first node from OPEN and put it on a list called CLOSED.
      Call this node n.
(4) If the depth of n equals the depth bound, go to (2);
      Otherwise continue.
(5) Expand node n, generating all immediate successors.  Put these at the
      beginning of OPEN (in predetermined order) and provide pointers back to n.
(6) If any of the successors are goal nodes, exit with the solution obtained by
      tracing back through the pointers; Otherwise go to (2).
```

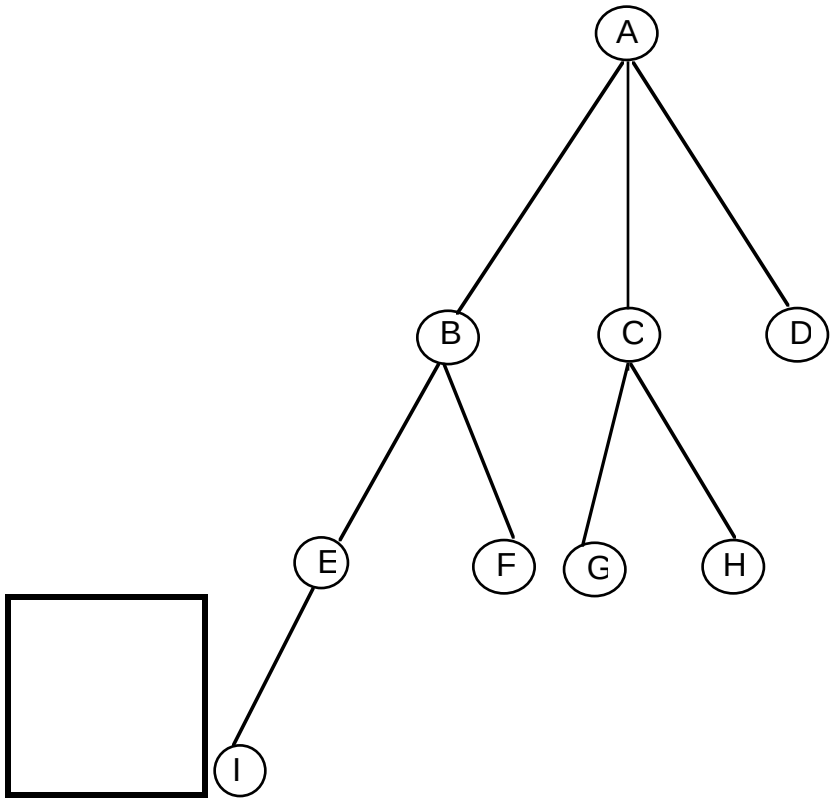

Figure 5: Tree Example for Depth First and Breadth First Search

DFS always explores the deepest node to the left first. That is, the one which is farthest down from the root of the tree. When a dead end (terminal node) is reached, the algorithm backtracks one level and then tries to go forward again. To prevent consideration of unacceptably long paths, a depth bound is often employed to limit the depth of search. At each node immediate successors are generated and a transition made to the left-most node, where the process continues recursively until a dead end or depth limit is reached. DFS would explore the tree in **Figure 5** in the order: I-E-b-F-B-a-G-c-H-C-a-D-A. Here the notation using lowercase letters represents the possible storing of provisional information about the subtree. For example, this could be a lower bound on the value of the tree.

```
//    The A* (DFS) algorithm expands the N.i successors of node N
//    in best first order.  It uses and sets solved, a global indicator.
//    It also uses a heuristic estimate function H(N), and a
//    transition cost C(N,N.i) of moving from N to N.i
//
IDA* (N) → cost
      bound ← H(N)
      while not solved
            bound ← DFS (N, bound)
      return bound                        // optimal cost

DFS (N, bound) → value
      if H(N) ≡ 0                   // leaf node
            solved ← true
            return 0
      new_bound ← ∞
      for each successor N.i of N
            merit ← C(N, N.i) + H(N.i)
            if merit ≤ bound
                  merit ← C(N,N.i) + DFS (N.i, bound - C(N,N.i))
                  if solved
                        return merit
            if merit < new_bound
                  new_bound ← merit
      return new_bound
```

Figure 6: The A* Depth First Search (DFS) algorithm for use with IDA*

Figure 6 enhances depth-first search with a form of iterative deepening that can be used in a single agent search like A*. DFS expands an immediate successor of some node N in a tree. The next successor (N.i) expanded is the one with lowest cost function. Thus the expected value of node N.i is the estimated cost C(N,N.i) plus H(N)--the known value of node N. The basic idea in iterative deepening is that a DFS is started with a depth bound of 1, and this continues with the depth bound increasing by one with each iteration. With each increase in depth the algorithm must re-initiate its depth-first search for the prescribed bound. The idea of iterative deepening, in conjunction with a memory function to retain the best available potential solution paths from iteration to iteration, is credited to Slate and Atkin (1977) who used it in their chess program. Korf (1985) showed how efficient this method is in single agent search, with his iterative deepening A* (IDA*) algorithm.

1.3.3 Bi-directional Search

To this point all search algorithms discussed (with the exception of means-ends analysis and backtracking) have been based on forward reasoning. Searching backwards from goal nodes to predecessors is relatively easy. Pohl (1971) combined forward and backward reasoning into a technique called bi-directional search. The idea is to replace a single search graph, which is likely to grow exponentially, with two smaller graphs -- one starting from the initial state and one starting from the goal. The search terminates when the two graphs intersect. This algorithm is guaranteed to find the shortest solution path through a general state-space graph. Empirical data for randomly generated graphs shows that Pohl's algorithm expands only about 1/4 as many nodes as unidirectional search (Barr and Feigenbaum, 1981). Pohl also implemented heuristic versions of this algorithm. However determining when and how the two searches will intersect is a complex process.

2 Heuristic Search Methods

George Polya, via his wonderful book "How To Solve It" (1945) may be regarded as the "father of heuristics". Polya's efforts focused on problem-solving, thinking and learning. He developed a short "heuristic dictionary" of heuristic primitives. Polya's approach was both practical and experimental. He sought to develop commonalties in the problem solving process through the formalization of observation and experience.

Present-day notions of heuristics are somewhat different from Polya's (Bolc and Cytowski, 1992). Current tendencies seek formal and rigid algorithmic solutions to specific problem domains rather than the development of general approaches which could be appropriately selected and applied to specific problems.

The goal of a heuristic search is to greatly reduce the number of nodes searched in seeking a goal. In other words, problems whose complexity grows combinatorially large may be tackled. Through

knowledge, information, rules, insights, analogies and simplification, in addition to a host of other techniques, heuristic search aims to reduce the number of objects that must be examined. Heuristics do not guarantee the achievement of a solution, although good heuristics should facilitate this. Over the years heuristic search has been defined in many different ways:

- it is a practical strategy increasing the effectiveness of complex problem solving (Feigenbaum & Feldman, 1963)
- it leads to a solution along the most probable path, omitting the least promising ones
- it should enable one to avoid the examination of dead ends, and to use already gathered data.

The points at which heuristic information can be applied in a search include:

1. deciding which node to expand next, instead of doing the expansions in either a strict breadth-first or depth-first order;
2. in the course of expanding a node, deciding which successor or successors to generate -- instead of blindly generating all possible successors at one time, and
3. deciding that certain nodes should be discarded, or pruned, from the search tree.

Bolc and Cytowski (1992) add:

> **"... use of heuristics in the solution construction process increases the uncertainty of arriving at a result ... due to the use of informal knowledge (rules, laws, intuition, etc.) whose usefulness have never been fully proven. Because of this, heuristic methods are employed in cases where algorithms give unsatisfactory results or do not guarantee to give any results. They are particularly important in solving very complex problems (where an accurate algorithm fails), especially in speech and image recognition, robotics and game strategy construction. ...**
>
> **Heuristic methods allow us to exploit uncertain and imprecise data in a natural way. ... The main objective of heuristics is to aid and improve the effectiveness of an algorithm solving a problem. Most important is the elimination from further consideration of some subsets of objects still not examined. ..."**

Most modern heuristic search methods are expected to bridge the gap between the completeness of algorithms and their optimal complexity (Romanycia and Pelletier, 1985). Strategies are being modified in order to arrive at a quasi-optimal -- instead of an optimal -- solution with a significant cost reduction (Pearl, 1984). Games, especially two-person, zero-sum games of perfect information like chess and checkers have proven to be a very promising domain for studying and testing heuristics.

2.1 Hill Climbing

Hill climbing is a depth first search with a heuristic measure that orders choices as nodes are expanded. The heuristic measure is the estimated remaining distance to the goal. The effectiveness of hill climbing is completely dependent upon the accuracy of the heuristic measure.

To conduct a hill climbing search of a tree:

```
Form a one-element queue consisting of a zero-length path that contains only the
      root node.
Repeat
      Remove the first path from the queue;
      Create new paths by extending the first path to all the neighbors of the
      terminal node.
         If New Path(s) result in a loop Then
            Reject New Path(s).
      Sort any New Paths by the estimated distances between their terminal nodes
      and the GOAL.
         If any shorter paths exist Then
            Add them to the front of the queue.
Until the first path in the queue terminates at the GOAL node or
      the queue is empty
If the GOAL node is found, announce SUCCESS, otherwise announce FAILURE.
```

In the above algorithm *neighbors* refer to "children" of nodes which have been explored and *terminal nodes* are equivalent to leaf nodes. Winston (1992) explains the potential problems affecting hill climbing. They are all related to issue of local "vision" versus global vision of the search space. The foothills problem is particularly subject to local maxima where global ones are sought, while the plateau problem occurs when the heuristic measure does not hint towards any significant gradient of proximity to a goal. The ridge problem illustrates just what it's called: you may get the impression that the search is taking you closer to a goal state, when in fact you traveling along a ridge which prevents you from actually attaining your goal.

2.2 Best First Search

Best First Search (Figure 7) is a general algorithm for heuristically searching any state space graph -- a graph representation for a problem which includes initial states, intermediate states, and goal states. In this sense a directed acyclic graph (DAG), for example, is a special case of a state space graph. Best First Search is equally applicable to data and goal driven searchers and supports the use of heuristic evaluation functions. It can be used with a variety of heuristics, ranging from a state's "goodness" to sophisticated measures based on the probability of a state leading to a goal which can be illustrated by examples of Bayesian statistical measures.

```
Procedure Best_First_Search (Start) → pointer
      OPEN ← {Start}                                  // Initialize
      CLOSED ← { }
      While OPEN ≠ { } Do                             // States Remain
            remove the leftmost state from OPEN, call it X;
            if X ≡ goal then
                  return the path from Start to X
            else
                 generate children of X
                 for each child of X do
                 CASE
                 the child is not on open or CLOSED:
                       assign the child a heuristic value
                       add the child to OPEN
                 the child is already on OPEN:
                       if the child was reached by a shorter path
                       then give the state on OPEN the shorter path
                 the child is already on CLOSED:
                       if the child was reached by a shorter path then
                            remove the state from CLOSED
                            add the child to OPEN
                 end_CASE
                 put X on closed;
                 re-order states on OPEN by heuristic merit (best leftmost)
      return NULL                                     // OPEN is empty
```

Figure 7: The Best First Search Algorithm (based on Luger and Stubblefield, 1993 (p121) with permission)

Similar to the depth-first and breadth-first search algorithms, best-first search uses lists to maintain states: OPEN to keep track of the current fringe of the search and CLOSED to record states already visited. In addition the algorithm orders states on OPEN according to some heuristic estimate of their

proximity to a goal. Thus, each iteration of the loop considers the most "promising" state on the OPEN list. According to Luger and Stubblefield (1993), just where hill climbing fails with its short-sighted and local vision is where the Best First Search improves. The following description of the algorithm closely follows that of Luger and Stubblefield (1993, p121):

> **" At each iteration, Best First Search removes the first element from the OPEN list. If it meets the goal conditions, the algorithm returns the solution path that led to the goal. Each state retains ancestor information to allow the algorithm to return the final solution path.**
>
> **If the first element on OPEN is not a goal, the algorithm generates it descendants. If a child state is already on OPEN or CLOSED, the algorithm checks to make sure that the state records the shorter of the two partial solution paths. Duplicate states are not retained. By updating the ancestor history of nodes on OPEN and CLOSED, when they are rediscovered, the algorithm is more likely to find a quicker path to a goal.**
>
> **Best First Search then heuristically evaluates the states on OPEN, and the list is sorted according to the heuristic values. This brings the "best" state to the front of OPEN. It is noteworthy that these estimates are heuristic in nature and therefore the next state to be examined may be from any level of the state space. OPEN, when maintained as a sorted list, is often referred to as a priority queue."**

Our notation in Figure 8 is that the dashed branches represent transitions that were not taken by the search algorithm. The solid lines relate to nodes that were put on the OPEN list, with the thick lines pointing to nodes that have been fully evaluated and moved onto the CLOSED list. Much of the excellent Luger and Stubblefield's (1993) text's fourth Chapter is devoted to the Best First Search. We provide their description below:

> **"Figure 8 shows a hypothetical state space with heuristic evaluations attached to some of its states. The states with attached evaluations are those actually generated in best-first search. The states expanded by the heuristic search algorithm are indicated in BOLD; note that it does not search all of the space. The goal of best-first search is to find the goal state by looking at as few states as possible; the more informed the heuristic, the fewer states are processed in finding the goal.**
>
> **A trace of the execution of Procedure Best_First_Search appears below. P is the goal state in this example, so states along the path to P tend to have low heuristic values. The heuristic is fallible: the state O has a lower value than the goal itself and is examined first. Unlike hill climbing, the algorithm recovers from this error and finds the correct goal."**

1. Open = {A5}; Closed = { }
2. evaluate A5; Open = {B4, C4, D6}; Closed = {A5}
3. evaluate B4; Open = {C4, E5, F5, D6}; Closed = {B4, A5}
4. evaluate C4; Open = {H3, G4, E5, F5, D6}; Closed = {C4, B4, A5}

5. evaluate H3; Open = {O2, P3, G4, E5, F5, D6}; Closed = {H3, C4, B4, A5}
6. evaluate O2; Open = {P3, G4, E5, F5, D6}; Closed = {O2, H3, C4, B4, A5}
7. evaluate P3; a solution is found!

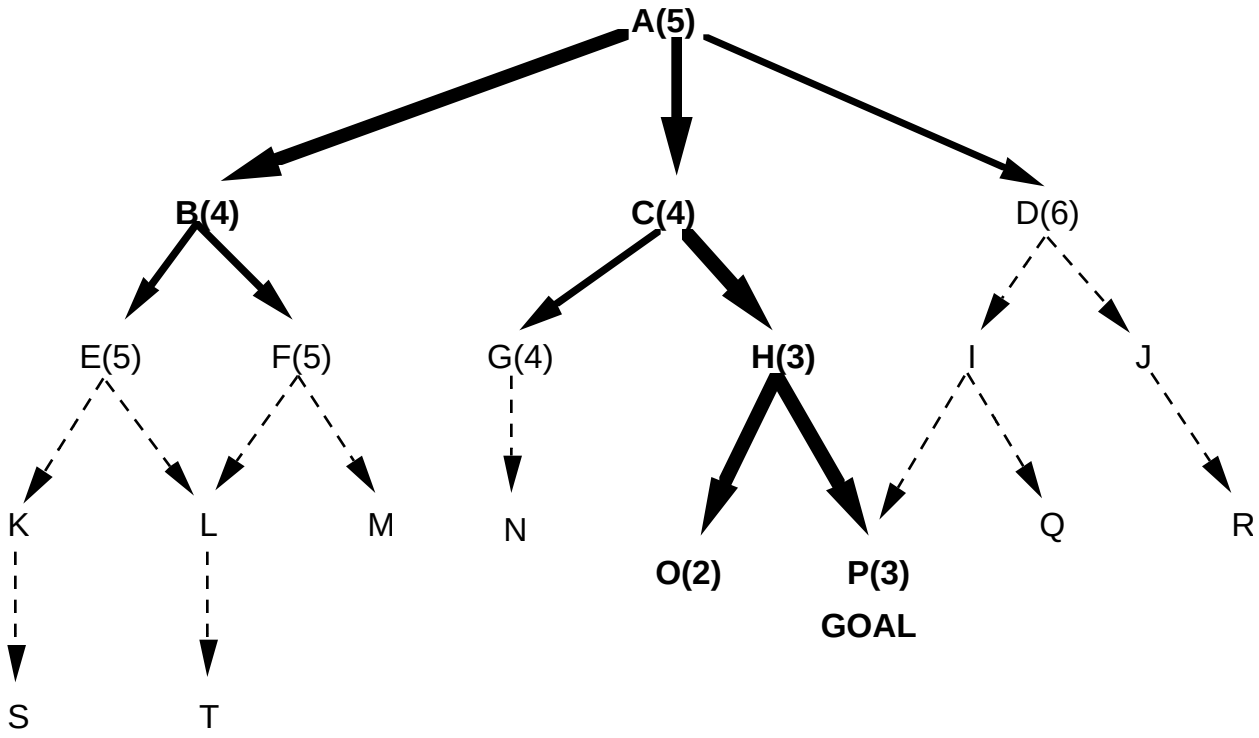

Figure 8: A hypothetical state space with heuristic evaluations for Best First Search,
Source: Based on Luger and Stubblefield, 1993. Artificial Intelligence 2nd ed., p122.
Benjamin/Cummings Publishing Company, Inc. Redwood City, CA. (with Permission)

When the Best First Search algorithm is used, the states are sent to the Open list in such a way that the most promising one will be expanded next. Because the search heuristic being used for measurement of distance from the goal state may prove erroneous, the alternatives to the preferred state are kept on Open. If the algorithm follows an incorrect path, it will retrieve the "next best" state and shift its focus to another part of the space. In the example above, children of State B were found to have poorer heuristic evaluations, than B's sibling, C, and so the search shifted there. However the children of B were kept on Open and could be returned to later if other optimal solutions are sought.

2.3 The A* Algorithm

The A* Algorithm, first described by Hart, Nilsson and Raphael (1968), attempts to find the minimal cost path joining the start node and the goal in a state-space graph. The algorithm employs an ordered state-space search and an estimated heuristic cost to a goal state, f* (known as an evaluation function), as does the Best-First Search (Section 2.2), but is unique in how it defines f*, so that it can guarantee an optimal solution path. The A* Algorithm falls into the branch and bound class of algorithms, typically employed in operations research to find the shortest path to a solution node in a graph.

The evaluation function, f*(n), estimates the quality of a solution path through node n, based on values returned from two components, g*(n) and h*(n). Here g*(n) is the minimal cost of a path from a start node to n, and h*(n) is a lower bound on the minimal cost of a solution path from node n to a goal node. As in branch and bound algorithms, for trees g* will determine the single unique shortest path to node n. For graphs, on the other hand, g* can err only in the direction of overestimating the minimal cost; if a shorter path is found, its value re-adjusted downward. The function h* is the carrier of heuristic information, and the ability to ensure that the value of h*(n) is less than h(n) (that is h*(n) is an underestimate of the actual cost, h(n), of an optimal path from n to a goal node) is essential to the optimality of the A* algorithm. This property, whereby h*(n) is always less than h(n), is known as the **admissibility condition.** If h* is zero, then A* reduces to the blind uniform-cost algorithm. If two otherwise similar algorithms, A1 and A2 can be compared to each other with respect to their h* function, i.e. h1* and h2*, then algorithm A1 is said to be more informed than A2 if, whenever a node n (other than a goal node) is evaluated, **h1*(n) > h2*(n).** Considerations for the cost of computing h* in terms of the overall computational effort involved, and algorithmic utility, determine the heuristic power of an algorithm. That is, an algorithm which employs an h* which is usually accurate, but sometimes inadmissible, may be preferred over an algorithm where h* is always minimal but hard to effect (Barr & Feigenbaum, 1981).

Thus we can summarize that the A* Algorithm is a branch and bound algorithm augmented by the dynamic programming principle: the best way through a particular, intermediate node, is the best way to that intermediate node from the starting place, followed by the best way from that intermediate node to

the goal node. There is no need to consider any other paths to or from the intermediate node (Winston, 1992).

3 Game-Tree Search

3.1 The Alpha-Beta Algorithms

To the human player of 2-person games the notion behind the Alpha-Beta algorithm is understood intuitively as:

> If I have determined that a move or a sequence of moves is bad for me (because of a refutation move or variation by my opponent), then I don't need to determine just how bad that move is. Instead I can spend my time exploring other alternatives earlier in the tree.
>
> Conversely, if I have determined that a variation or sequence of moves is bad for my opponent, I don't need to determine how bad it is.

Some of these ideas are illustrated in **Figure 9**. Here the thick solid line represents the current solution path. This in turn has replaced a candidate solution, here shown with dotted lines, Everthing to the right of the optimal solution path represent alternatives that are simply proved inferior. The path of the current solution is called the Principal Variation (PV) and nodes on that path are marked as PV nodes. Similarly the alternatives to PV nodes are CUT nodes, where only a few successors are examined before a proof of inferiority is found. In time the successor to a cut-node will be an ALL node where everything must be examined to prove the cut-off at the CUT node. The number or bound value by each node represents the return to the root of the cost of the solution path.

In the forty years since its inception, the alpha-beta minimax algorithm has undergone many revisions and refinements to improve the efficiency of its pruning, so that today it is the primary search engine for two-person games. There have been many landmarks on the way, including Knuth and Moore's (1975) formulation in a negamax framework, Pearl's (1980) introduction of Scout and the special formulation for chess with the Principal Variation Search (Marsland and Campbell, 1982) and NegaScout (Reinefeld, 1983). The essence of the method is that the search seeks a path whose value falls between two bounds called alpha and beta, which form a window. With this approach one can also incorporate an artificial narrowing of the alpha-beta window, thus encompassing the notion of "aspiration search", with a mandatory research on failure to find a value within the corrected bounds. This leads naturally to the incorporation of Null Window Search (NWS) to improve upon Pearl's Test procedure. Here the Null Window Search (NWS) procedure covers the search at a CUT node (**Figure 9**), where the

cutting bound (Beta) is negated and increased by 1 in the recursive call. This refinement has some advantage in the parallel search case, but otherwise NWS, **Figure 10**, is entirely equivalent to the minimal window call in NegaScout. Additional improvements include the use of iterative deepening with "transposition tables" and other move-ordering mechanisms to retain a memory of the search from iteration to iteration. These improvements help ensure that the better subtrees are searched sooner, leading to greater pruning efficiency (more cutoffs) in the later subtrees. **Figure 10** encapsulates the essence of the algorithm and shows how the first variation from a set of PV nodes, as well as any superior path that emerges later, is given special treatment. Alternates to PV nodes will always be CUT nodes, where a few successors will be examined. In a minimal game tree only one successor to a CUT node will be examined, and it will be an ALL node where every thing is examined. In the general case the situation is more complex, as **Figure 9** shows.

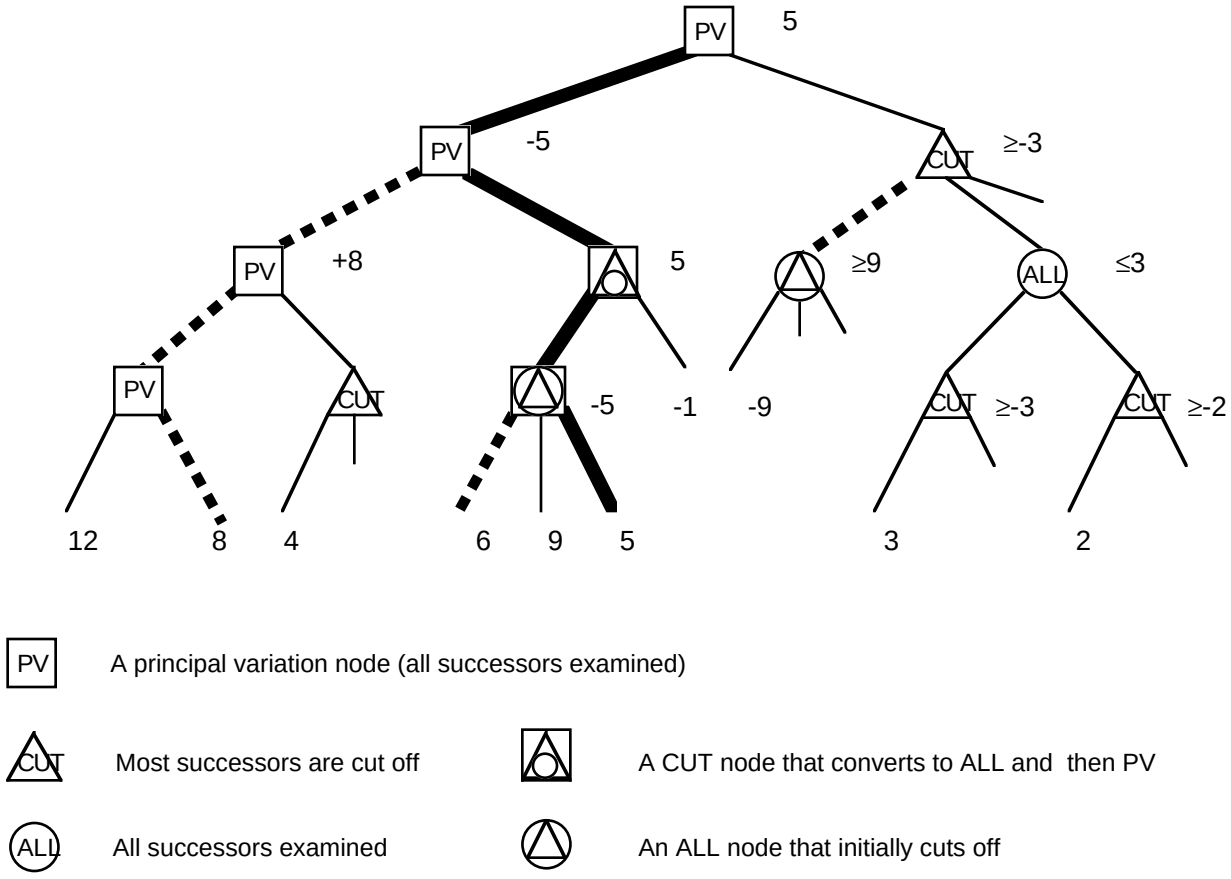

Figure 9: The PV, CUT and ALL nodes of a tree, showing its optimal path (bold) and value (5).

```
ABS (node, alpha, beta, height) → tree_value
      if height ≡ 0
            return Evaluate(node)                    // a terminal node
      next ← FirstSuccessor (node)             // a PV node
      best ← - ABS (next, -beta, -alpha, height -1)
      next ← SelectSibling (next)
      while next ≠ NULL do
            if best ≥ beta then
                  return best                  // a CUT node
            alpha ← max (alpha, best)
            merit ← - NWS (next, -alpha, height-1)
            if merit > best then
                  if (merit ≤ alpha) or (merit ≥ beta) then
                        best ← merit
                  else best ← -ABS (next, -beta, -merit, height-1)
            next ← SelectSibling (next)
      end
      return best                                  // a PV node
end

NWS (node, beta, height) → bound_value
      if height ≡ 0 then
            return Evaluate(node)                    // a terminal node
      next ← FirstSuccessor (node)
      estimate ← - ∞
      while next ≠ NULL do
            merit ← - NWS (next, -beta+1, height-1)
            if merit > estimate then
                  estimate ← merit
            if merit ≥ beta then
                  return estimate                    // a CUT node
            next ← SelectSibling (next)
      end
      return estimate                                  // an ALL node
end
```

Figure 10: Scout/PVS version of Alpha-Beta Search (ABS) in the Negamax framework

3.2 SSS* Algorithm

The SSS* Algorithm was introduced by Stockman (1979) as a game-searching algorithm that traverses subtrees of the game tree in a best-first fashion similar to the A* Algorithm. SSS* was shown to be superior to the original Alpha-Beta algorithm in the sense that it never looks at more nodes, while occasionally examining fewer (Pearl, 1984). Roizen and Pearl (1983) the source of the following description of SSS* continue**:**

> **"...the aim of SSS* is the discovery of an optimal solution tree ...In accordance with the best-first split-and-prune paradigm, SSS* considers "clusters" of solution trees and splits (or refines) that cluster having the highest upper bound on the merit of its constituents. Every node**

in the game tree represents a cluster of solution trees defined by the set of all solution trees that share that node. ...the merit of a partially developed solution tree in a game is determined solely by the properties of the frontier nodes it contains, not by the cost of the paths leading to these nodes. The value of a frontier node is an upper bound on each solution tree in the cluster it represents, ... SSS* establishes upper bounds on the values of partially developed solution trees by seeking the value of terminal nodes, left to right, taking the minimum value of those examined so far. These monotonically nonincreasing bounds are used to order the solution trees so that the tree of highest merit is chosen for development. The development process continues until one solution tree is fully developed, at which point that tree represents the optimal strategy and its value coincides with the minimax value of the root.

...The disadvantage of SSS* lies in the need to keep in storage a record of all contending candidate clusters, which may require large storage space, growing exponentially with search depth (Pearl, 1984, p245)."

Heavy space and time overheads have kept SSS* from being much more than an example of a best-first search, but current research seems destined to now relegate SSS* to a historical footnote. Recently Plaat et al. (1995) formulated the node-efficient SSS* algorithm into the alpha-beta framework using successive NWS search invocations (supported by perfect transposition tables) to achieve a memory-enhanced test procedure that provides a best-first search. With their MTD(f) function Plaat et al. (1995) claim that SSS* can be viewed as a special case of the time-efficient alpha-beta algorithm, instead of the earlier view that alpha-beta is a k-partition variant of SSS*. This is an important contribution that should find wider application for best-first search because of its improved efficiency.

4 Parallel Search

The easy availability of low-cost computers has stimulated interest in the use of a multitude of processors for parallel traversals of decision trees. The few theoretical models of parallelism do not accommodate communication and synchronization delays that inevitably impact the performance of working systems. There are several other factors to consider too, including:

1. How best to employ the additional memory and i/o resources that become available with the extra processors.
2. How best to distribute the work across the available processors.
3. How to avoid excessive duplication of computation.

Some important combinatorial problems have no difficulty with the third point because every eventuality must be considered, but these tend to be less interesting in an artificial intelligence context.

One problem of particular interest is game-tree search, where it is necessary to compute the value of the tree, while communicating an improved estimate to the other parallel searchers as it becomes available. This can lead to an "acceleration anomaly" when the tree value is found earlier than is possible with a sequential algorithm. Even so, uniprocessor algorithms can have special advantages in that they can be optimized for best pruning efficiency, while a competing parallel system may not have the right information in time to achieve the same degree of pruning, and so do more work (suffer from 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.

Although the standard basis for comparing the efficiency of parallel methods is simply:

$$\text{speedup} == \frac{\text{time taken by a sequential single-processor algorithm}}{\text{time taken by a P-processor system}}$$

This basis is often misused, since it depends on the efficiency of the uniprocessor implementation.

The exponential growth of the tree size (solution space) with depth of search makes parallel search algorithms especially susceptible to anomalous speedup behavior. Clearly, acceleration anomalies are among the welcome properties, but more commonly anomalously bad performance is seen, unless the algorithm has been designed with care.

In game playing programs of interest to artificial intelligence, parallelism is not primarily intended to find the answer more quickly, 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 a result sooner, scalability measures the ability to expand the size of both the problem and the system at the same time:

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

Thus scale-up close to unity reflects successful parallelism.

4.1 Parallel Single Agent Search

Single agent game tree search is important because it is useful for several robot planning activities, such as finding the shortest path through a maze of obstacles. It 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 seen--especially when optimal solutions are sought. This traversal can safely be done in parallel, since there are no cutoffs to be missed. Although move ordering can reduce node expansions, it does not play the same crucial role as in dual-agent game-tree search, where significant parts of the search space are often pruned away. For this reason, parallel single agent search techniques usually achieve better speedups than their counterparts in adversary games.

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, although parallel IDA* needs only linear storage space

and runs faster. In addition, cost-effective methods exist (e.g. parallel window search in **Section 4.2.1**) that determine non-optimal solutions with even less computing time.

4.1.1 Parallel A*

Given P processors, the simplest way to parallelize A* is to let each machine work on one of the currently best states on a global openlist (a place holder for nodes that have not yet been examined). This approach minimizes the search overhead, as confirmed in practice by (Kumar et al., 1988). Their relevant experiments were run on a shared memory BBN-Butterfly machine with 100 processors, where a search overhead of less than 5% was observed for the traveling sales-person (TSP) problem.

But elapsed time is more important than the node expansion count, because the global openlist is accessed both before and after each node expansion, so that memory contention becomes a serious bottleneck. It turns out, that a centralized strategy for managing the openlist is only useful in domains where the node expansion time is large compared to the openlist access time. In the TSP problem, near linear time speedups were achieved with up to about 50 processors, when a sophisticated heap data structure was used to significantly reduce the openlist access time (Kumar et al., 1988).

Distributed strategies using local openlists reduce 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 a global "Blackboard" table can be used to hold state descriptors of the currently best nodes. After selecting a state from its local openlist, each processor compares its f-value (lower bound on the solution cost) 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, then node descriptors are sent (or received), so that all active processors are 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 openlist, it still causes memory contention with increasing parallelism. To alleviate this problem, (Huang and Davis, 1989) proposed a distributed heuristic search algorithm called Parallel Iterative A* (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 to better ones. 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.

4.1.2 Parallel IDA*

IDA* (**Figure 6**) has proved to be effective, when excessive memory requirements undermine best-first schemes. 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. The 15-puzzle is a popular game made up of 15 tiles that slide within a 4x4 matrix. The object is to slide the tiles through the one empty spot until all tiles are aligned in some goal state. An optimal solution to a hard problem might take 66 moves. 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 more than a P-fold speedup is possible when a processor finds a goal node early in the final iteration. In fact, Rao et al. (1987) report an average speedup of 9.24 with 9 processors on the 15-puzzle! Perhaps more relevant is the all-solution-case where no superlinear speedup is possible. Here, an average speedup of 0.93P with up to thirty (P) processors on a bus-based multiprocessor architecture (Sequent Balance 21000) was achieved. This suggests that only low multiprocessing overheads (locking, work transfer, termination detection and synchronization) were experienced.

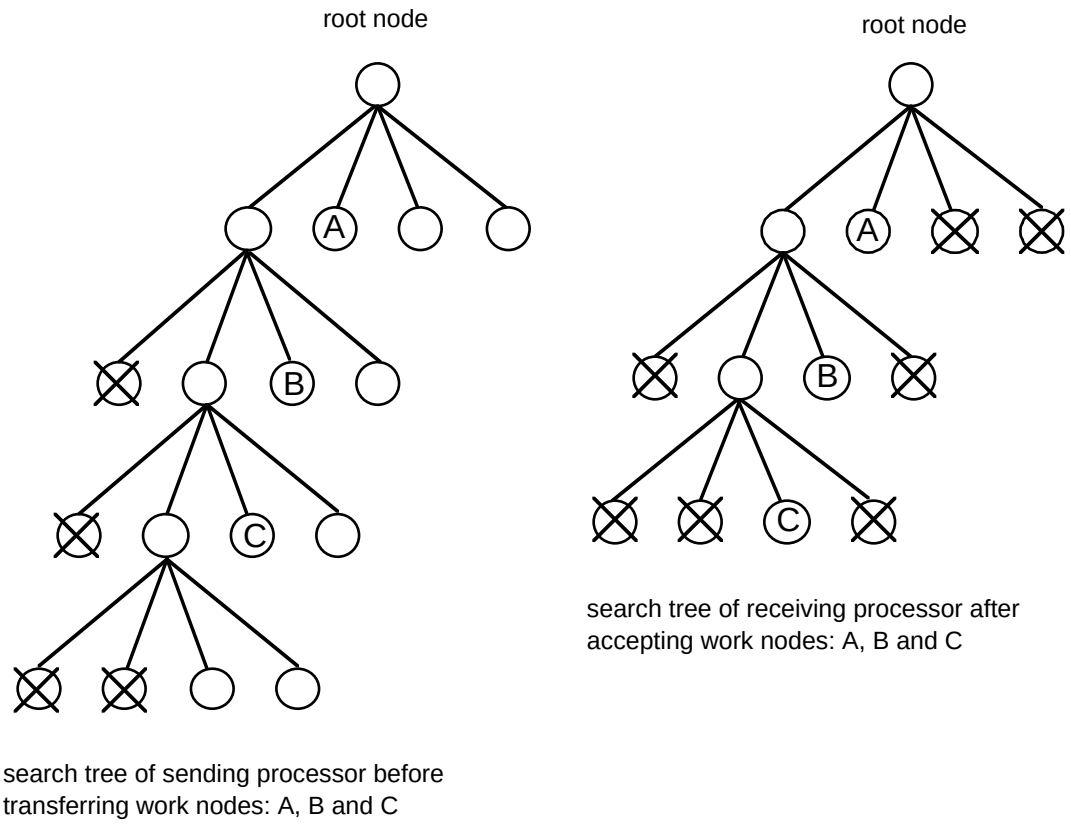

Figure 11: A Work distribution scheme

PIDA* employs a task attraction scheme like that shown in **Figure 11** for distributing the work among the processors. When a processor becomes idle, it asks a neighbor for a 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. An optimal splitting strategy would depend on the regularity (uniformity of width and height) of the search tree, though short subtrees should never be given away. When the tree is 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 **Figure 11**) should be transferred.

4.1.3 A Comparison with Parallel Window Search

Another parallel IDA* approach borrows from Baudet's (1978) parallel window method for searching adversary games (**Section 4.2.1**). Powley and Korf (1991) adapted this method to single agent search,

under the title Parallel Window Search (PWS). Their basic idea is to simultaneously start as many iterations as there are processors. This works for a small number of 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.

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 may contain cheaper solutions). But according to Powley and Korf (1991), 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." Compared to PIDA*, the degree of parallelism is limited, and it remains unclear, how to apply PWS in domains where the cost-bound increases are variable.

In summary, PWS and PIDA* complement each other, so it seems natural to combine them to form a single search scheme that runs PIDA* on groups of processors administered by a global PWS algorithm. The amount of communication needed depends on the work distribution scheme. A fine-grained distribution requires more communication, while a coarse grained work distribution generates fewer messages (but may induce unbalanced work load). Note that the choice of the work distribution scheme also affects the frequency of good acceleration anomalies. Along these lines perhaps the best results have been reported by Reinefeld (1995). Using AIDA* (Asynchronous Parallel IDA*) near linear speedup was obtained on a 1024 transputer-based system solving thirteen instances of the 19-puzzle. Reinefeld's paper includes a discussion of the communication overheads in both ring and toroid systems, as well as a description of the work distribution scheme.

4.2 Adversary Games

In the area of two-person games, early simulation studies with a Mandatory Work First (MWF) scheme (Akl et al., 1982), and the PVSplit algorithm (Marsland and Campbell, 1982), showed that a high degree

of parallelism was possible, despite the work imbalance introduced by pruning. Those papers saw that in key applications, (e.g. chess) the game-trees are well ordered, because of the wealth of move ordering heuristics that have been developed (Slate and Atkin, 1977), and so the bulk of the computation occurs during the search of the first subtree. The MWF approach uses the shape of the critical tree that must be searched. Since that tree is well-defined and has regular properties, it is easy to generate. In their simulation 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. Marsland & Campbell (1982), on the other hand, recognized that 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 tested and analyzed by Marsland and Popowich (1985). Even so, the static processor allocation schemes like MWF and PVSplit cannot achieve high levels of parallelism, although PVSplit does very well with up to half a dozen 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, 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. Depth limits are also an important part of changing search modes and in managing transposition tables.

4.2.1 Parallel Aspiration-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 of the range 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, Baudet estimates a maximum speedup of between 5 and 6 even when using infinitely many processors. In practice, parallel window search can only be effectively employed on systems with two or three processors. This is because even in the best case (when the successful processor uses a minimal window) at least the critical game tree must be expanded. The critical tree has about the square root of the leaf nodes of a uniform tree of the same depth, and it represents the smallest tree that must be searched under any circumstances.

4.2.2 Advanced Tree-splitting Methods

Results from fully recursive versions of PVSplit using the Parabelle chess program (Marsland and Popowich, 1985), confirmed the earlier simulations and offered some insight into a major problem: In a P-processor system, P - 1 processors are often 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. This led to the development of variations that dynamically assign processors to the search of the principal variation. Notable is the work of Schaeffer (1989), which uses a loosely coupled network of workstations, and Hyatt et al.'s (1989) independent 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 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. from PVSplit), to the dynamic processor re-allocation schemes of Hyatt et al. (1989), and also Schaeffer (1989). These later systems try to dynamically identify the ALL nodes of **Figure 9** and search them in parallel, leaving the CUT nodes (where only a few successors might be examined) for serial expansion. 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 realizes a speedup of 12 with 32 processors for alpha-beta trees generated by Othello programs. This speedup result might be attributed to the smaller average branching factor of about 10 for Othello trees, compared to an average branching factor of about 35 for chess. If that uniprocessor solution is inefficient--for example, by omitting an important node-ordering mechanism like the use of transposition tables (Reinefeld and Marsland, 1994)--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 until recently, when the potential advantages of using heuristic methods like hash tables to replace the openlist were exploited (Plaat et al., 1995).

4.2.3 Dynamic Distribution of Work

The key to successful large-scale parallelism lies in the dynamic distribution of work. The Young Brothers Wait Concept (Feldmann, 1993) is one such scheme in which the parallelism is best described through the help of a definition: The search for a successor N.j of a node N in a game tree must not be started until after the left-most sibling N.1 of N.j is completely evaluated. Thus N.j can be given to

another processor if and only if it has not yet been started and the search of N.1 is complete. Since this is also the requirement for PVSplit, how then do the two methods differ and what are the tradeoffs? There are two significant differences. The first is at startup and the second is in the potential for parallelism. PVSplit starts much more quickly since all the processors traverse the first variation (first path from the root to the search horizon of the tree), and then split the work at the nodes on the path as the processors back up the tree to the root. Thus all the processors are busy from the beginning but, on the other hand, this method suffers from increasingly large synchronization delays as the processors work their way back to the root of the game tree (Marsland and Popowich, 1985). Thus good performance is possible only with relatively few processors, because the splitting is purely static. In the work of Feldmann et al. (1990) the startup time for this system is lengthy, because initially only one processor (or a small group of processors) is used to traverse the first path. When that is complete, the right siblings of the nodes on the path can be distributed for parallel search to the waiting processors. If there are a thousand such processors, then perhaps less than one percent would initially be busy. Gradually, the idle processors are brought in to help the busy ones, but this takes time. However, and here comes the big advantage, the system is now much more dynamic in the way it distributes work, so it is less prone to serious synchronization loss. Further, although many of the nodes in the tree will be CUT nodes (which are a poor choice for parallelism because they generate high search overhead), others will be ALL nodes, where every successor must be examined and they can simply be done in parallel. Usually CUT nodes generate a cut-off quite quickly, so by being cautious about how much work is initially given away once N.1 has been evaluated, one can keep excellent control over the search overhead, while getting full benefit from the dynamic work distribution that Feldmann's method provides.

4.2.4 Recent Developments

Although there have been several successful implementations involving parallel computing systems, significantly better methods for NP-hard problems, like game-tree search, remain elusive. Theoretical studies often focus 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. For example, 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 are fundamental to many AI problems, and Kumar and Rao (1990) have fully examined 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 work distribution method (illustrated in **Figure 11**) 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 design of the Deep Blue chess program is to manufacture a new VLSI processor in large quantity. Though the original aim was to make a system that was scalable to 1000 processors, initially the machine may use 32 nodes of an SP-2 computer each with half a dozen processors that are chess-specific. The original design was the major contribution of his thesis, 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 solves our game-tree search problems.

Defining Terms

Admissibility Condition: The necessity that the heuristic measure never over estimates the cost of the remaining search path, thus ensuring that an optimal solution will be found.

A* Algorithm: A best-first procedure that uses an admissible heuristic estimating function to guide the search process to an optimal solution.

Alpha - Beta: The conventional name for the bounds on a depth-first minimax procedure that are used to prune away redundant subtrees in two-person games.

And/Or Tree: A tree which enables the expression of the decomposition of a problem into subproblems enabling alternate solutions to subproblems through the use of AND/OR node labelling schemes.

Backtracking: A component process of many search techniques whereby recovery from unfruitful paths is sought by backing up to a juncture where new paths can be explored.

Best First Search: A heuristic search technique which finds the most promising node to explore next by maintaining and exploring an ordered Open node list.

Bi-directional Search: A search algorithm which replaces a single search graph, which is likely to grow exponentially, with two smaller graphs -- one starting from the initial state and one starting from the goal state.

Blind Search : A characterization of all search techniques which are heuristically uninformed. Included amongst these would normally be state space search, means ends analysis, generate and test, depth first search, and breadth first search.

Branch and Bound Algorithm: A potentially optimal search technique which keeps track of all partial paths contending for further consideration, always extending the shortest path one level.

Breadth First Search: An uninformed search technique which proceeds level by level visiting all the nodes at each level (closest to the root node) before proceeding to the next level.

Depth First Search: A search technique which first visits each node as deeply and to the left as possible

Generate and Test: A search technique which proposes possible solutions and then tests them for their feasibility

Heuristic Search: an informed method of searching a state space with the purpose of reducing its size and finding one or more suitable goal states.

Iterative Deepening: A successive refinement technique which progressively searches a longer and longer tree until an acceptable solution path is found.

Means-Ends Analysis: An AI technique which tries to reduce the "difference" between a current state and a goal state.

SSS*: A best-first search procedure for two-person games.

Parallel Window Aspiration Search: A method in which a multitude of processors search the same tree, but each with different (non-overlapping) alpha-beta bounds.

Mandatory Work First: A static two-pass process which first traverses the minimal game tree and uses the provisional value found to improve the pruning during the second pass over the remaining tree.

PVSplit (Principal Variation Splitting): A static parallel search method that takes all the processors down the first variation to some limiting depth, and then splits the subtrees among the processors as they back up to the root of the tree.

Young Brothers Wait Concept: A dynamic variation of PVSplit in which idle processors wait until the first path of left-most subtree has been searched before giving work to an idle processor.

References

Akl, S.G., Barnard, D.T. and R.J. Doran. 1982. Design, analysis and implementation of a parallel tree search machine. *IEEE Trans. on Pattern Anal. and Mach. Intell.,* 4(2):192-203.

Barr, A, and Feigenbaum, E.A. 1981. *The Handbook of Artificial Intelligence V1*. William Kaufmann, Inc. Stanford, California.

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

Bolc, L. and Cytowski, J. 1992. *Search Methods for Artificial Intelligence.* Academic Press, Inc., San Diego, CA.

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

Feldmann, R. 1993. Game Tree Search on Massively Parallel Systems. PhD. thesis, University of Paderborn, Germany.

Felten, E.W. and S.W. Otto. 1989. A highly parallel chess program. *Procs. Int. Conf. on 5th Generation Computer Systems,* p. 1001-1009.

Feigenbaum, E , Feldman, J. 1963. *Computers and Thought* New York: McGraw-Hill.

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

Hart, P.E., Nilsson, N.J., and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on SSC.* SSC-4:100-107.

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

Hsu, F-h. 1990. 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.

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

Knuth D. and Moore, R. 1975. An analysis of Alpha -Beta pruning. *Artificial Intelligence.* 6(4):293-326.

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

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

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

Luger, J and Stubblefield, W. 1993. *Artificial Intelligence*: *structures and strategies for complex problem solving,* 2nd ed. Benjamin/Cummings Publishing Company, Inc., Redwood City, CA.

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

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

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

Nilsson, N. 1971. *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill Publishing Co. New York.

Pearl, J. 1980. Asymptotic properties of minimax trees and game-searching procedures. *Artificial Intelligence* 14(2):113-38.

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

Plaat, A., Schaeffer, J., Pijls, W. and A. de Bruin. 1995. Best-first fixed-depth game-tree search in practice. In Proceedings of IJCAI-95, pp 273-279. Montreal, Canada, August. Kaufmann

Polya, G. 1945. *How to Solve It.* Princeton University Press. Princeton, N.J.

Pohl, I. 1971. Bi-directional search. In B. Meltzer and D. Michie (Eds.), *Machine Intelligence 6.* 127-140 American Elsevier, New York.

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

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

Reinefeld, A. 1983. An improvement to the Scout tree-search algorithm. *Intl. Computer Chess Assoc. Journal*. 6(4):4-14.

Reinefeld, A. and T.A. Marsland. 1994. Enhanced Iterative-Deepening Search. *IEEE Trans. on Pattern Anal. and Mach. Intell.,* 16(7):701-710.

Reinefeld, A. 1995. Scalability of Massively Parallel Depth-First Search. In P.M. Pardalos, M.G.C. Resende, K.G. Ramakrishnan (eds.), *Parallel Processing of Discrete Optimization Problems*. p.305-322. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 22, American Mathematical Society, Providence, RI.

Roizen, I and Pearl, J. 1983 A Minimax Algorithm Better Than Alpha-Beta?: Yes and No" *Artificial Intelligence* 21(1-2), 199-220, March 1983.

Romanycia, M, Pelletier, F. 1985. What is heuristic? *Computer Intelligence* 1:(24-36)

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

Slate, D.J. and L.R. Atkin. 1977. Chess 4.5 - the Northwestern University Chess Program. In P. Frey, editor, *Chess Skill in Man and Machine,* p. 82-118. Springer-Verlag: New York.

Stockman, G. 1979. A minimax algorithm better than alpha-beta? *Artificial Intelligence* 12(2): 179-96.

Winston, P.H. 1992. *Artificial Intelligence* 3rd ed. Addison-Wesley, Reading, MA.

Acknowledgement

The authors thank David Kopec for assistance with artwork.

For Further Information: The most regularly and consistently cited source of information for this article is the Journal of Artificial Intelligence. There are numerous other Journals including, for example, AAAI Magazine, CACM, IEEE Expert, ICCA Journal, and the International Journal of Man-Machine Studies which frequently publish articles related to this subject area. Also prominent has been the *Machine Intelligence Series* of Volumes edited by Donald Michie with various others. An excellent reference source is the 3-volume Handbook of Artificial Intelligence by Barr and Feigenbaum (1981),

In addition there are numerous national and international and national conferences on AI with Published Proceedings, headed by the International Joint Conference on AI (IJCAI). Classic books on AI methodology include Feigenbaum and Feldman's (1963) *Computers and Thought* and Nils Nilsson's (1971) *Problem-Solving Methods in Artificial Intelligence*. There are a number of popular and thorough textbooks on AI. Two relevant books on the subject of search in are *Heuristics* (Pearl, 1984) and the more recent *Search Methods for Artificial Intelligence* (Bolc & Cytowski, 1992).