Search

D. Kopec, T.A. Marsland and J.L. Cox

Introduction

Artificial Intelligence (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 **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**, the use of **iterative deepening**, and variations on the **alpha-beta** minimax algorithm, including the recent **MTD(f) algorithm**.

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. Since many of the heuristic methods are computationally intensive, the second half of the Chapter focuses on parallel methods, which can exploit the benefits of parallel processing   The importance of parallel search is presented through an assortment of relatively recent algorithms including the parallel iterative deepening algorithm (PIDA*), **principal variation splitting (PVSplit)**, and the **young brothers wait concept**. In addition, dynamic tree splitting methods have evolved for both shared memory parallel machines and networks of distributed computers. Here the issues include load balancing, processor utilization and communication overhead. For single-agent search problems we consider not only work-driven dynamic parallelism, but also the more recent data-driven parallelism employed in **transposition table driven scheduling (TDS)**. In adversarial games, tree pruning makes work load balancing particularly difficult, and hence we also consider some recent advances in dynamic parallel methods for game-tree search.

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.

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. Forward reasoning is one technique for identifying states that are closer to a goal state. Working backwards from a goal to the current 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-goals. Top down or data-driven reasoning is based on simply being able to reach a state that is defined as closer to a goal. Often application of operators to a problem state may not lead directly to a goal state, so 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 because of 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 a board game) 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.
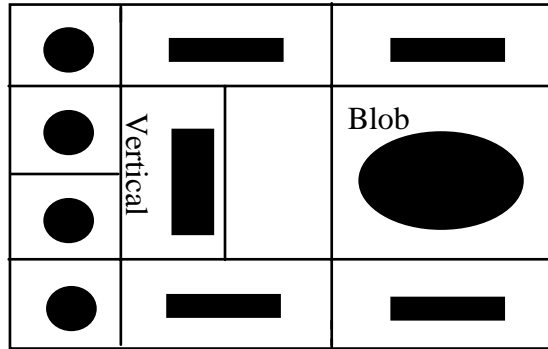
Figure 3: Problem Reduction and The Sliding Block Puzzle

For example, **Figure 3** shows the "Donkey" sliding block puzzle. It 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,
    3. OR 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.
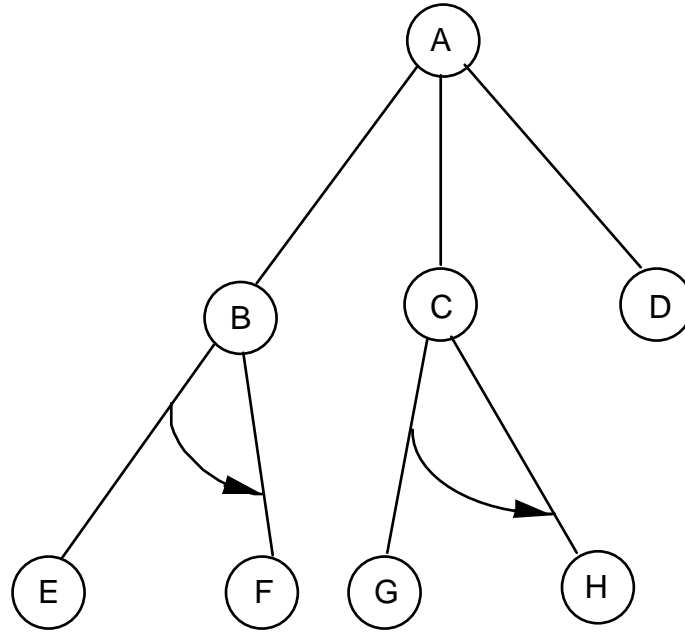
Figure 4: And/Or Tree

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.
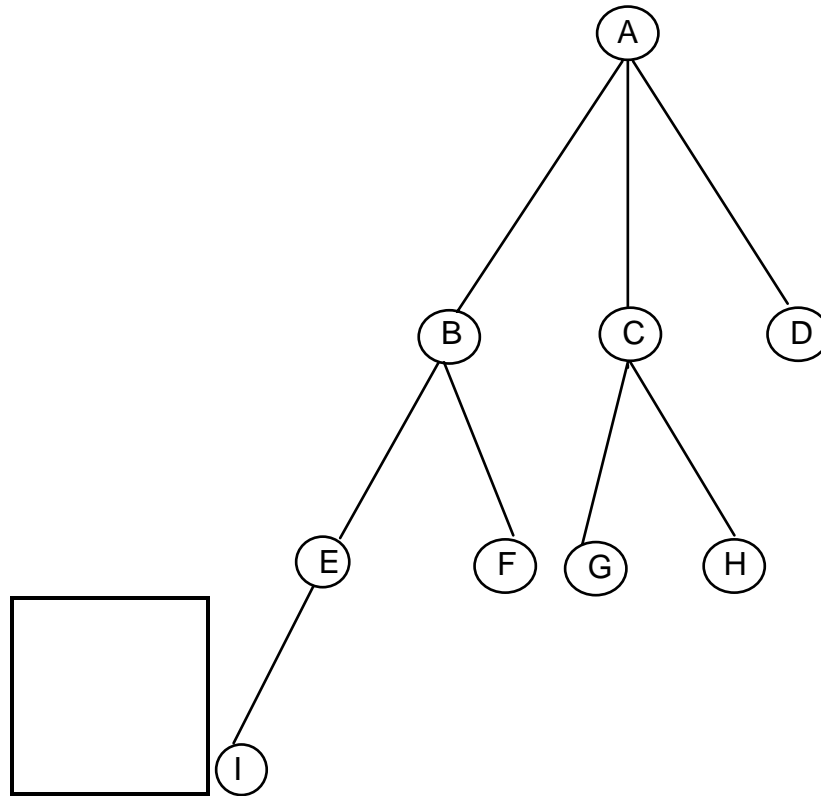
1.2.1 Breadth First Search

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 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. In **Figure 5**, DFS explores the tree 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 <u>iterative deepening</u> 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 to be

expanded is (N.i), 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 bound increases by one at each new 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   Bidirectional 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 <u>bidirectional search</u>. 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.

Russell and Norvig [2003] analyze the bidirectional search and come to the conclusion that it is $O(b^{d/2})$ in terms of average case time and space complexity. They point out that this is significantly better than $O(b^d)$, which would be the cost of searching exhaustively in one direction. Identification of sub-goal states could do much to reduce the costs. The large space requirements of the algorithm are considered its weakness.

However, Kaindl and Kainz [1997] have demonstrated that the long held belief that the algorithm is afflicted by the frontiers passing each other is wrong. They developed a new generic approach that dynamically improves heuristic values, but is only applicable to bidirectional heuristic. Their empirical results have found that the bidirectional heuristic search can be performed very efficiently, with limited memory demands. Their research has resulted in a better understanding of an algorithm whose practical usefulness has been long neglected, with the conclusion that it is better suited to certain problems than corresponding unidirectional searches. For more details the reader should review their paper [Kaindl and Kainz, 1997]. The next section focuses on heuristic search methods.

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 that 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. deciding which successor or successors to generate when generating a node--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 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 <u>foothills problem</u> is particularly subject to local maxima where global ones are sought, while the <u>plateau problem</u> occurs when the heuristic measure does not hint towards any significant gradient of proximity to a goal. The <u>ridge problem</u> 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. **Simulated annealing** is like trying to combine hill climbing with a random walk in a way that yields both efficiency and completeness [Russell and Norvig, 2003, p.115]. The idea is to "temper" the downhill process of hill climbing in order to avoid some of the pitfalls discussed above by increasing the probability of "hitting" important locations to explore. It is like intelligent guessing.

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 that 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 that 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

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]. Best First Search improves at just the point where hill climbing fails with its

short-sighted and local vision. The following description of the algorithm closely follows that of Luger and

Stubblefield [1993, p.121]:

"   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."
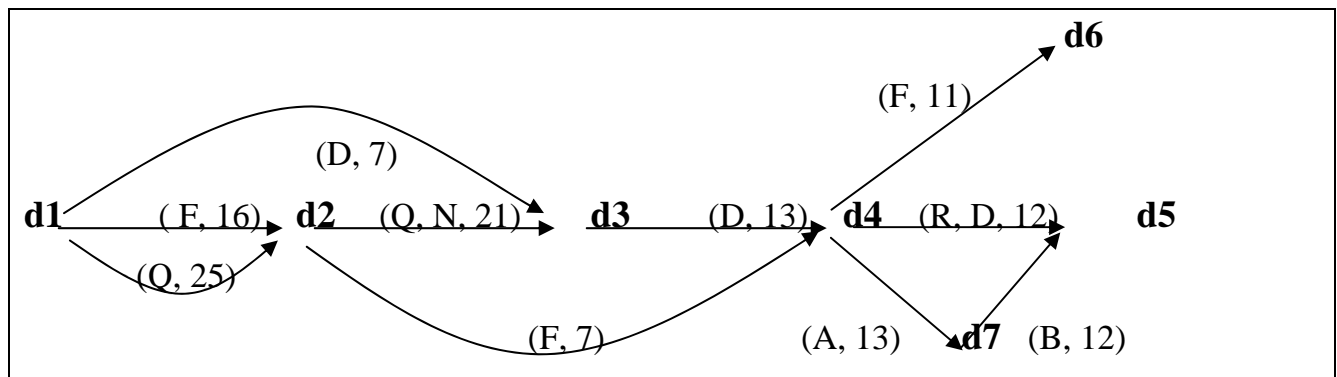
Figure 8.   A state-space graph for a hypothetical subway system

Here is a graph of a hypothetical search space. The problem is to find a shortest path from **d1** to **d5** in this

directed and weighted graph (Figure 8), which could represent a sequence of local and express subway train

stops. The "F" train starts at **d1** and visits stops **d2** (cost 16), **d4** (cost 7), and **d6** (cost 11). The "D" train starts

at **d1** and **d3** (cost 7), **d4** (cost 13) and **d5** (cost 12). Other choices involve combinations of "Q", "N", "R",

and "A" trains with the F and/or D train. By applying the "best first search" algorithm we can find the

shortest path from **d1** to **d5**. **Figure 9** shows a state tree representation of this graph.
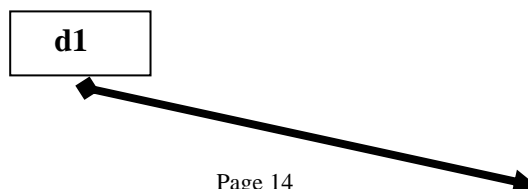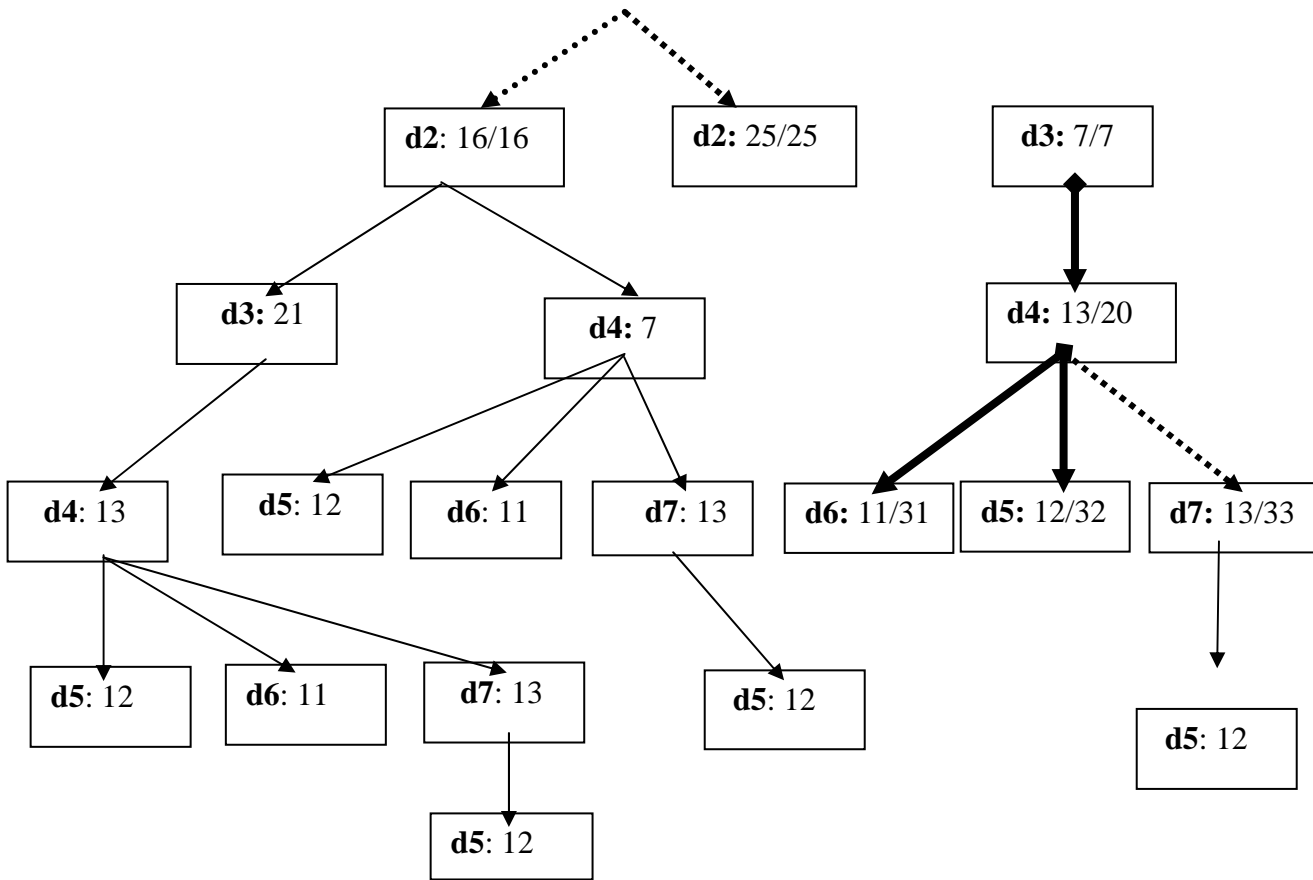
Figure 9.   A Search Tree for the Graph in Figure 8

The thick arrowed path is the shortest path. [d1, d3, d4, d5].   The dashed edges are nodes put on the Open

Node queue, but not further explored. A trace of the execution of procedure Best First Search appears below:

1.   Open = [d1]; Closed = [ ]
2.   Evaluate d1; Open = [d3, d2]; Closed = [d1]
3.   Evaluate d3; Open = [d4, d2]; Closed = [d3, d1]
4.   Evaluate d4; Open = [d6, d5, d7, d2]; Closed = [d4, d3, d1]
5.   Evaluate d6; Open = [d5, d7, d2]; Closed = [d6, d4, d3, d1]
6.   Evaluate d5; a solution is found Closed = [d5, d6, d4, d3, d1]

Note that nodes d6 and d5 are at the same level so we don't take d6 in our search for the shortest path. Hence

the shortest path for this graph is [d1, d3, d4, d5]. After we reach to our goal state "d5", we can also find the

shortest path from "d5" to "d1" by retracing the tree from d5 to d1.

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 is 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 the Open List. 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 node d2 were found to have poorer heuristic evaluations, than sibling d3, and so the search shifted there. However the children of d3 were kept on Open and could be returned to later, if other 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). It uniquely 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 **h1*(n) > h2*(n)**, whenever a node n (other than a goal node) is evaluated. The cost

of computing h* in terms of the overall computational effort involved, and algorithmic utility, determines 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].

Stewart and White [1991] presented the multiple objective A* algorithm (MOA*). Their research is motivated by the observation that most real-world problems have multiple, independent, and possibly conflicting objectives. MOA* explicitly addresses this problem by identifying the set of all non-dominated paths from a specified start node to given set of goal nodes in an OR graph. This work shows that MOA* is complete and is admissible, when used with a suitable set of heuristic functions.

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, then don't determine exactly how good it is for me.

Figure 10 illustrates some of these ideas. Here the thick solid line represents the current solution path. This in turn has replaced a candidate solution, here shown with dotted lines. Everything to the right of the optimal solution path represents 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 and, until the recent invention of the MTD(f) variant, it has served as 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 10**), 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 11**) 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. A transposition table is a cache of previously generated states that are typically hashed into a table to avoid redundant work. These improvements help ensure that the better subtrees are searched sooner, leading to greater pruning efficiency (more cutoffs) in the later subtrees. **Figure 11** 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 everything is examined. In the general case the situation is more complex, as **Figure 10** shows.
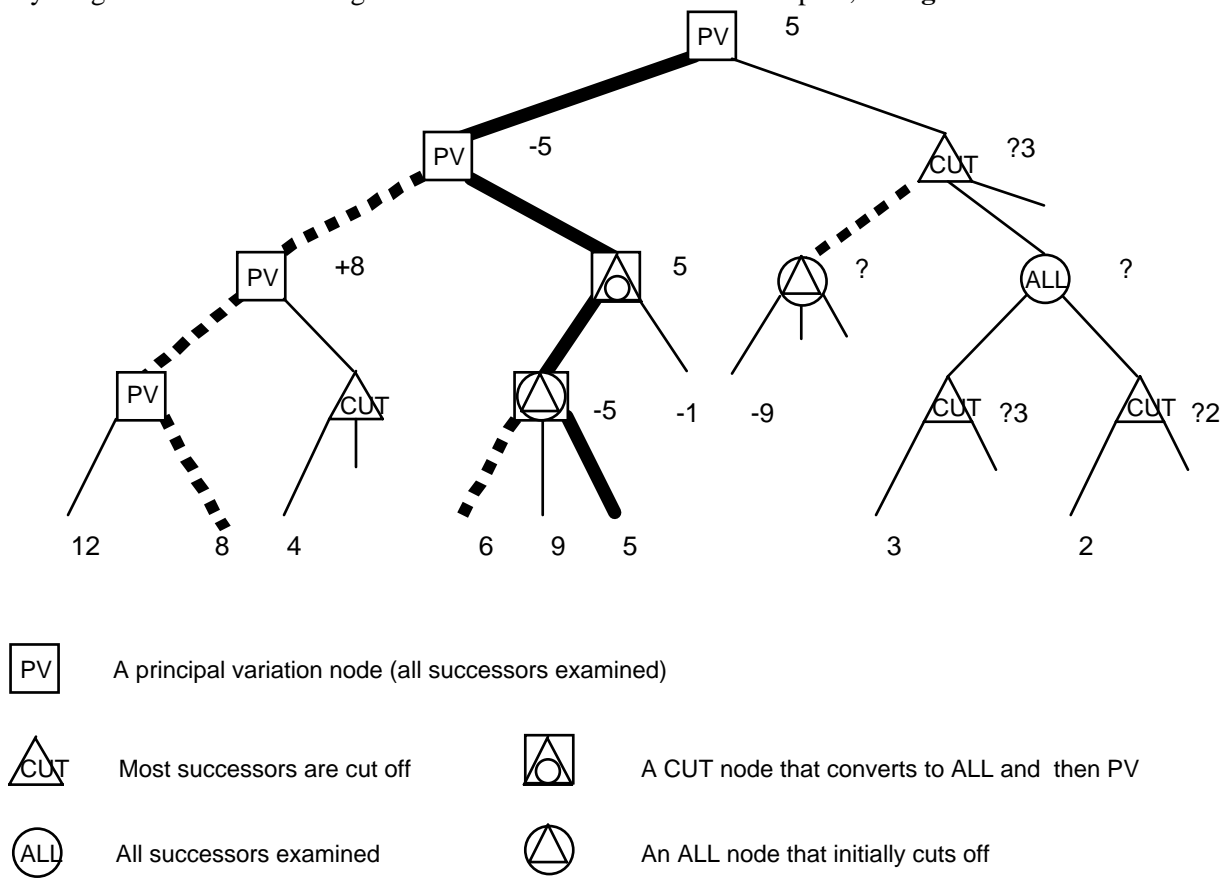
Figure 10: The PV, CUT and ALL nodes of a tree, showing its optimal path (bold) and value (5).

```
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 11: 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*,

state that:

"...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, p.245]."

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 introduction of the MTD(f) algorithm 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*. MTD(f) is an important contribution that has now been widely adopted as the standard two-person game-tree search algorithm. It is described below.

3.3 The MTD(f) Algorithm

MTD(f) is usually run in an iterative deepening fashion, and each iteration proceeds by a sequence of minimal or NULL window alpha-beta calls. The search works by zooming in on the minimax value, as **Figure 12** shows.

```
int MTDF ( node_type root, int f, int d)
{
     g = f;
     upperbound = +INFINITY;
     lowerbound = -INFINITY;
     repeat
           if (g == lowerbound)
```

```
            beta = g + 1
        else
            beta = g;
        g = AlphaBetaWithMemory(root, beta - 1, beta, d);
        if (g < beta)
            then upperbound = g
            else lowerbound = g;
    until (lowerbound >= upperbound);
return g;
}
```

Figure 12. The MTD(f) Algorithm Pseudo-code

The bounds stored in *upperbound* and *lowerbound* form an interval around the true minimax value for a

particular search depth *d*. The interval is initially set to [- ∞, + ∞]. Starting with the value *f,* returned from a

previous call to MTD(f), each call to alpha-beta returns a new minimax value *g,* which is used to adjust the

bounding interval and to serve as the pruning value for the next alpha-beta call. For example, if the initial

minimax value is 50, alpha-beta will be called with the pruning values 49 and 50. If the new minimax value

returned, *g,* is less than 50, upperbound is set to *g*. If the minimax value returned, *g*, is greater than or equal

to 50, lowerbound is set to *g*. The next call to alpha-beta will use *g*-1 and *g* for the pruning values (or *g* and *g*

+1, if *g* is equal to the lowerbound). This process continues until *upperbound* and *lowerbound* converge to a

single value, which is returned. MTD(f) will be called again with this newly returned minimax estimate and

an increased depth bound until the tree has been searched to a sufficient depth.

As a result of the iterative nature of MTD(f), the use of transposition tables is essential to its efficient

implementation. In tests with a number of tournament game-playing programs, MTD(f) outperformed ABS

(Scout/PVS, **Figure 11**). It generally produces trees that are 5-15% smaller than ABS [Plaat, Schaeffer, Pijls,

& de Bruin, 1996]. MTD(f) is now recognized as the most efficient variant of ABS and has been rapidly

adopted as the new standard in minimax search.

3.4   **Recent Developments**

Sven Koenig has developed Minimax Learning Real-Time A* (Min-Max LRTA*), a real-time heuristic

search method that generalizes Korf's [1990] earlier LRTA* to non-deterministic domains. Hence it can be

applied to "robot navigation tasks in mazes, where robots know the maze but do not know their initial position

and orientation (pose). These planning tasks can be modeled as planning tasks in non-deterministic domains whose states are sets of poses." Such problems can be solved quickly and efficiently with Min-Max LRTA* requiring only a small amount of memory [Koenig, 2001].

Martin Mueller [2001] introduces the use of Partial Order Bounding (POB) rather than scalar values for construction of an evaluation function for computer game-playing. Propagation of partially ordered values through a search tree has been known to lead to many problems in practice. Instead POB compares values in the leaves of a game tree and backs up boolean values through the tree. The effectiveness of this method was demonstrated in examples of capture races in the game of GO [Mueller 2001].

Schaeffer, Plaat, and Junghanns [2001] demonstrate that the distinctions for evaluating heuristic search should not be based on whether the application is for single-agent or two-agent search. Instead, they argue that the search enhancements applied to both single-agent and two-agent problems for creating high performance applications are the essentials. Focus should be on generality for creating opportunities for reuse. Examples of some of the generic enhancements (as opposed to problem specific ones) include the alpha-beta algorithm, transposition tables and IDA*, Efforts should be made to enable more generic application of algorithms.

Hong, Huang, and Lin [2001] present a **genetic algorithm** approach that can find a good next move by reserving the board evaluation of new offspring in partial game-tree search. Experiments have proven promising in terms of speed and accuracy when applied to the game of GO.

The FF (Fast Forward) Planning System of Hoffman and Nebel [2001] uses a heuristic that estimates goal distances by ignoring delete lists. Facts are not assumed to be independent. It uses a new search strategy that combines hill-climbing with systematic search.   Powerful heuristic information is extended and used to prune the search space.

4   Parallel Search

The easy availability of low-cost computers has stimulated interest in the use of multiple 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:

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

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:

> time taken to solve a problem of size s by a single-processor

scale-up =   --

> 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 described subsequently) 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].
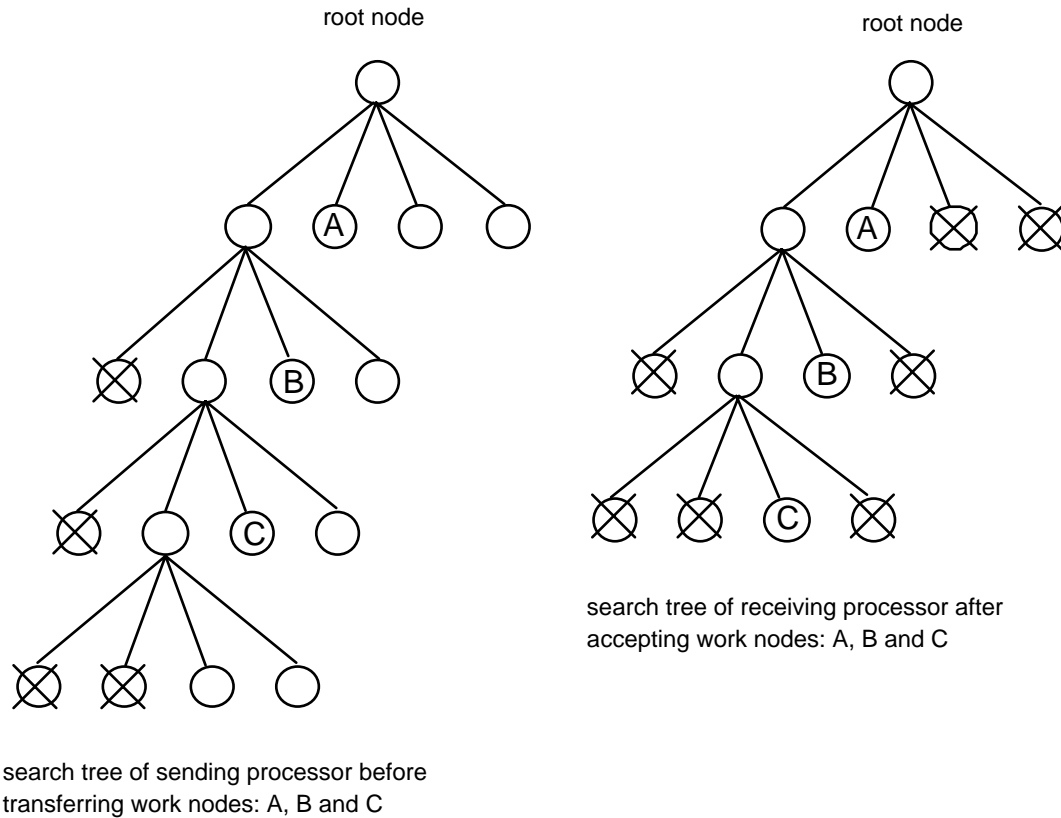
Distributed strategies using local openlists reduce the memory contention problem. But again some communication must be provided to allow processors to share the most promising state descriptors, so that no computing resources are wasted in expanding inferior states. For this purpose a global "Blackboard" 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.

root node

root node

search tree of receiving processor after
accepting work nodes: A, B and C

search tree of sending processor before
transferring work nodes: A, B and C

Figure 13: A Work distribution scheme

PIDA* employs a task attraction scheme like that shown in **Figure 13** 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 13**) 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 (described subsequently). 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 10** 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. There are four primary issues in dynamic search: **1) Search overhead** measures the size of the tree searched by the parallel method with respect to the best sequential algorithm. As mentioned above, in some cases super-linear speedup can occur when the parallel algorithm actually visits fewer nodes. **2) Synchronization overhead** problems occur when processors are idle waiting for results from other processors, thus reducing the effective use of the parallel computing power (processor utilization). **3) Load balancing** reflects how evenly the work has been divided among available processors and similarly effects processor

utilization. **4) Communication overhead** in a distributed memory system occurs when results must be communicated between processors via message passing.

Each of these issues must be considered in designing a dynamic parallel algorithm. The distribution of work to processors can either be accomplished in a work-driven fashion, whereby idle processors must acquire new work either from a blackboard or by requesting work from another processor.

The young brothers wait concept [Feldmann, 1993] is a work-driven 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 the PVSplit algorithm, 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, for example, in the case of one thousand such processors, possibly 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.

On the other hand, transposition-driven work scheduling (TDS) for parallel single-agent and game-tree searches, proposed by Romein, Plaat, Bal, & Schaeffer [1999], is a data-driven technique that in many cases offers considerable improvements over work-driven scheduling on distributed memory architectures. TDS reduces the communication and memory overhead associated with the remote lookups of transposition tables partitioned among distributed memory resources. This permits lookup communication and search computation to be integrated. The use of the transposition tables in TDS, as in IDA*, prevents the repeated searching of previously expanded states.

TDS employs a distributed transposition table that works by assigning to each state a "home processor," where the transposition entry for that state is stored. A signature associated with the state indicates the number of its home processor. When a given processor expands a new state, it evaluates its signature and sends it to its home processor, without having to wait for a response, thus permitting the communication to be carried out asynchronously. In other words, the work is assigned to where the data on a particular state is stored, rather than having to lookup a remote processor's table and wait for the results to be transmitted back. Alternatively, when a processor receives a node, it performs a lookup of its local transposition table to determine whether the node has been searched before. If not, the node is stored in the transposition table and added to the local work queue. Furthermore, since each transposition table entry includes a search bound, this prevents redundant processing of the same subtree by more than one processor. The resulting reduction in both communication and search overhead yield significant performance benefits. Speedups that surpass IDA* by a factor of more than 100 on 128 processors [Romein, Platt, Bal, & Schaeffer, 1999] have been reported in selected games.

Cook and Varnell [1998] report that TDS may be led into doing unnecessary work at the goal depth, however, and therefore they favor a hybrid combination of techniques that they term *adaptive*

parallel iterative deepening search. They have implemented their ideas in the system called EUREKA.

Their system employs machine learning to select the best technique for a given problem domain.

4.2.4   Recent Developments

Despite advances in parallel single agent search, significant improvement in methods for game-tree search has remained elusive. Theoretical studies have often focused 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 may 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 can occur), resulting in asymmetric trees with a significant work distribution problem and significant search overhead. The search overhead occurs when a processor examines nodes that would be pruned by the sequential algorithm, but has not yet received the relevant results from another processor.

The intrinsic difficulty of searching game trees under pruning conditions has been widely recognized. Hence considerable research has been focused on the goal of dynamically identifying when unnecessary search is being performed, thereby freeing processing resources for redeployment. For example, Feldmann et al. [1990] used the concept of making "young brothers wait" to reduce search overhead, and developed the "helpful master" scheme to eliminate the idle time of masters waiting for their slaves' results. On the other hand, young brothers wait can still lead to significant synchronization overhead.

Generalized depth-first searches are fundamental to many AI problems. In this vein, 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 parts of the trees are marked and are dynamically assigned to any idle processor. In principle, this work distribution method (illustrated in **Figure 13**) could also be used for deterministic adversary game trees. Shoham and Toledo [2001] developed a parallel, randomized best-first minimax search (RBFM). RBFM expands terminal nodes randomly chosen, with higher probabilities

assigned to nodes with better values. This method seems amenable to parallelism, but seems to suffer from speculative search overhead.

The advent of MTD(f) as the primary sequential search algorithm has greatly improved circumstances and seems likely to lead to significant advances in parallel game tree search. The use of a minimal window, and hence a single bound for pruning the tree can greatly reduce the search overhead. Local pruning is no longer sensitive to results obtained by different processors searching other parts of the game tree. Romein [2000] developed a parallel version of MTD(f) that achieved speedup over other parallel game tree algorithms. Since MTD(f) is iterative in nature, its use of transposition tables is especially critical to performance. Kishimoto and Schaeffer [2002] have developed an algorithm called TDSAB that combines MTD(f) with the transposition table driven scheduling of TDS to achieve significant performance gains in a number of selected games.

Many researchers feel that hardware advances will ultimately increase parallel processing power to the point where we will simply overwhelm the problem of game tree search with brute force. They feel that this will make the need for continued improvement of parallel game tree algorithms less critical. Perhaps in time we will know if massive parallelism solves all our game-tree search problems. However, the results of complexity theory should assure us that unless P=NP, improvements in search techniques and heuristics will remain both useful and desirable.

5   New Vistas in AI Search

The advent of the world wide web (WWW) in 1993 naturally led to interest in search techniques for the internet, especially when intelligence can be applied. Search Engines are computer programs that can automatically contact other network resources on the Internet, searching for specific information or key words, and report the results of their search. Intelligent Agents are computer programs that help the users to conduct routine tasks, to search and retrieve information, to support decision-making and to act as domain experts. They do more than just "search and match." There are Intelligent Agents for consumers that search and filter information. Intelligent Agents have been developed for product and vendor finding, for negotiation and for learning. They can help determine what to buy to satisfy a specific need by looking for specific products'

information and then critically evaluating them. Some examples are: Firefly, which uses a *collaborative filtering* process that can be described as "word of mouth" to build the profile. It asks a consumer to rate a number of products, then matches his/her ratings with the ratings of other consumers with similar tastes, recommends products that have not yet been rated by the consumer, etc. Intelligent Agents for Product and Vendor Finding can find bargains.

One example is Jango from NetBot/Excite. It originates requests from the user's site (rather than Jango's). Vendors have no way of determining whether the request is from a real customer or from the agent. Jango provides product reviews. Kasbah, from MIT Lab, is for users who want to sell or buy a product. It assigns the task to an agent who is then sent out to proactively seek buyers or sellers. Intelligent Agents for consumers can act as negotiation agents, helping to determine the price and other terms of transactions. Kasbah has multiple agents. Users create agents for the purpose of selling and buying goods. It can exemplify 3 strategies: anxious, cool-headed, and frugal. Tete - @ - tete is an intelligent agent for considering a number of different parameters: price, warranty, delivery time, service contracts, return policy, loan options, and other value-added services. Upon request it can even be argumentative. Finally, there are learning agents capable of learning individuals' preferences and making suitable suggestions based upon these preferences. Memory Agent from IBM and Learn Sesame from Open Sesame use learning theory for monitoring customers' interactions. It learns customers' interests, preferences and behavior and delivers customized service to them accordingly [Deitel, Deitel, Steinbuhler, 2001]. Expect many interesting developments in this arena combining some of the theoretical findings we have presented with practical results.

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, hence alternate solutions to subproblems through the use of AND/OR node labelling schemes can be found.

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 that finds the most promising node to explore next by maintaining and exploring an ordered Open node list.

Bidirectional 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 that proceeds level by level visiting all the nodes at each level (closest to the root node) before proceeding to the next level.

Data-Driven Parallelism: A load-balancing scheme in which work is assigned to processors based on the characteristics of the data.

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 that proposes possible solutions and then tests them for their feasibility.

Genetic Algorithm: A stochastic hill-climbing search in which a large population of states is maintained. New states are generated by mutation and crossover, which combines pairs of earlier states from the population.

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 that progressively searches a longer and longer tree until an acceptable solution path is found.

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

MTD(f) Algorithm: A minimal window minimax search recognized as the most efficient Alpha-Beta variant.

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.

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

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.

Simulated Annealing: a stochastic algorithm that returns optimal solutions when given an appropriate "cooling schedule".

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

Transposition Table Driven Scheduling (TDS): A data-driven load-balancing scheme for parallel search that assigns a state to a processor based on the characteristics or signature of the given state.

Work-Driven Parallelism: A load-balancing scheme in which idle processors explicitly request work from other processors

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.

Cook, D.J. and Varnell, R.C. 1998. Adaptive Parallel Iterative Deepening Search. *J. of AI Research* 8, 139-166.

Deitel, H.M., Deitel, P.J., Steinbuhler, K. 2000. *E-Business and e-Commerce for Managers.* Prentice Hall Publishers.

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,* pp. 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,* pp. 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), pp. 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.

Hoffman, J. and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *J. of AI Research* 14:253-302.

Hong, T.P, Huang, K.Y., and Lin, W.Y. 2001. Adversarial Search by Evolutionary Computation. *Evolutionary Computation* 9(3):371-385.

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)*, pp. 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.

Kaindl, H. and Kainz, G. 1997. Bidirectional Heuristic Search Reconsidered. *J. of AI Research* 7:283-317.

Kishimoto, A. and Schaeffer, J. Distributed Game-Tree Search Using Transposition Table Driven Work Scheduling, *International Conference on Parallel Processing (ICPP)*, pp. 323-330, 2002.

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

Koenig, S. 2001. Minimax real-time heuristic search. *Artificial Intelligence*. 129:165-195.

Korf, R.E. 1990. Real-time heuristic search, *Artificial Intelligence*. 42(2-3):189-211.

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,* pp. 122-127, Saint Paul. (Los Altos: Kaufmann).

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

Luger, J. 2002. *Artificial Intelligence*: *structures and strategies for complex problem solving,* 4th 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.

Mueller, M. 2001. Partial order bounding: A new approach to evaluation in game tree search. *Artificial Intelligence,* 129:279-311.

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

Plaat, A., Schaeffer, J., Pijls, W. and de Bruin, A. 1996. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87(1-2):1-38.

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.* pp. 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.*, pp. 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*. pp. 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. and Pelletier, F. 1985. What is heuristic? *Computational Intelligence* 1:24-36

Romein, J., Plaat, A., Bal, H. and Schaeffer, J. 1999. Transposition Table Driven Work Scheduling in Distributed Search. In *Procs. AAAI National Conf.*, pp. 725-731, Orlando.

Romein, J.W. 2000. Multigame -- an environment for distributed game-tree search. *Ph.D. Thesis*. ASCI dissertation series No. 53. Vrije Universiteit Amsterdam.

Russell, S. and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach (2nd ed)*, Prentice Hall, Englewood Cliffs, NJ.

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

Schaeffer, J., Plaat, A. and Junghanns, J. 2001. Unifying single-agent and two-player search. *Information Sciences* 134(3-4):151-175.

Shoham, Y. and Toledo, S. 2002. *Parallel Randomized Best-First Minimax Search.* Artificial Intelligence 137:165-196.

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,* pp. 82-118. Springer-Verlag: New York.

Stewart, B.S. and White, C.C. 1991. Multiobjective A*. *J of the ACM* 38(4):775-814

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

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, ICGA Journal, and the International Journal of Computer Human 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]. AI Texts which have considerable focus on Search techniques: are George Luger's, *Artificial Intelligence* [4th ed, 2002] and particularly current is Russell and Norvig's *Artificial Intelligence: a modern approach* [(2nd ed.) 2003].