Ann. Rev. Comput. Sci. 1987. 2:451–67

TECHNICAL RERPORT
R-75

SEARCH TECHNIQUES

Judea Pearl and Richard E. Korf

Computer Science Department, University of California, Los Angeles, California 90024

INTRODUCTION

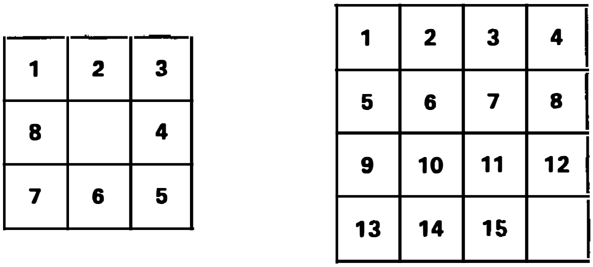
Search is a universal problem-solving mechanism in artificial intelligence (AI). Almost by definition, the sequence of steps required for solution of AI problems is not known a priori but must be determined by a systematic trial-and-error exploration of alternatives. This is known as *search behavior*. Typical search tasks range from solving puzzles such as Rubik's Cube, to playing games such as chess, to solving complex practical problems such as medical diagnosis and configuring computer systems.

Common examples in the AI literature of search problems are the Eight Puzzle and its larger relative the Fifteen Puzzle (see Figure 1). The Eight Puzzle consists of a 3 × 3 square frame containing 8 numbered square tiles and an empty position called the "blank." The legal operators slide any tile horizontally or vertically adjacent to the blank into the blank position. The task is to rearrange the tiles from some random initial configuration into a particular goal configuration.

A *problem space* is the environment in which the search takes place (Newell & Simon 1972). A problem space consists of a set of states of the problem and a set of operators that change the state of the problem. For example, in the Eight Puzzle, the states are the different permutations of the tiles, and the operators are the primitive legal moves. A *problem instance* is a problem space together with an initial state and a goal state. In the case of the Eight Puzzle, the initial state would be the initial configuration of the tiles, and the goal state is a particular configuration, such as that shown in the figure. The problem-solving task is to find a sequence of operators that map the initial state to a goal state. Such a sequence is called a *solution* to the problem.

Problem spaces are usually represented by graphs in which the states of the space are represented by nodes, and the operators by edges between

8756–7016/87/1115–0451$02.00



1	2	3
8		4
7	6	5

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Figure 1 Eight and Fifteen Puzzles.

nodes. Edges may be undirected or directed, depending on whether their corresponding operators are invertible or not. Although most problems are actually represented by graphs with more than one path between a pair of nodes, for simplicity they are often represented as trees with the initial state as the root. The cost of this simplification is that the same state may be represented by more than one node in the tree.

This brief exposition begins by describing general techniques for searching problem spaces. Brute-force search is the simplest and most general method since it requires no knowledge other than that contained in the problem space. Brute-force search algorithms include breadth-first, depth-first, depth-first iterative-deepening, and bidirectional search. Heuristic search requires some additional information in the form of an estimate of the number of operators in the problem space between a pair of states; as a result, it is much more efficient. Heuristic search algorithms include best-first search, the A* algorithm, and iterative-deepening-A*.

After discussing these general techniques, we consider three special cases of problem spaces: AND/OR graphs, game trees, and constraint-satisfaction problems. AND/OR graphs are most suitable for problems where the solution is naturally represented as a tree or graph structure as opposed to a simple path. Most of the brute-force and heuristic search algorithms can be extended to handle AND/OR graphs as well. Game trees are used to model two-player adversary games such as chess. The best-known search algorithms for game trees are minimax with Alpha-Beta pruning, SSS*, and SCOUT. Constraint-satisfaction problems occur in situations where the solution is a particular artifact satisfying a set of constraints rather than a sequence of actions. For example, the task in the Eight-Queens problem is to place eight queens on a chessboard such that no two queens are on the same row, column, or diagonal.

The efficiency of these algorithms, in terms of the costs of the solutions generated, the amount of time required to execute, and the amount of computer memory required, is of central concern. Since search is a universal problem-solving method, the only limitation on its applicability is the efficiency with which it can be performed.

The two parameters of a search tree that determine the efficiency of various search algorithms are its *branching factor* and its *depth*. The branching factor (b) is the average number of children of a given node or the average number of new operators applicable to a given state. The depth (d) is the length of the shortest path from the initial state to a goal state or the length of the shortest sequence of operators solving the problem.

BRUTE-FORCE SEARCH

The most general search algorithms are brute-force searches since they do not require any domain-specific knowledge. All that is required for a brute-force search is a set of states, a set of legal operators, an initial state, and a goal criterion. The most important brute-force techniques are breadth-first, depth-first, depth-first iterative-deepening, and bidirectional searches.

Breadth-First Search

Breadth-first search generates the nodes of the search tree in order of their distance from the root (see Figure 2*a*). Since breadth-first search never generates a deeper-level node in a tree until all the nodes at shallower levels have been generated, the first path found to a goal will be a path of shortest length, or an *optimal* solution by this measure. Since the amount of time used by breadth-first search is proportional to the number of nodes generated, and the number of nodes at level d is b^d, the amount of time used by breadth-first search is $b+b^2+\cdots+b^d$, or $O(b^d)$ in the worst case.

The main drawback of breadth-first search, however, is its memory requirement. Since each level of the tree must be saved in its entirety in order to generate the next level, and the amount of memory is proportional to the number of nodes stored, the space complexity of breadth-first search is also $O(b^d)$. As a result, breadth-first search is severely space-limited in

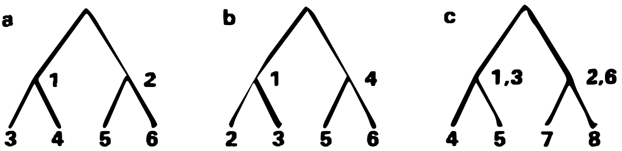


Figure 2 Order of node generation for searches: (*a*) breadth first; (*b*) depth first; (*c*) depth-first iterative deepening.

practice and will exhaust the available memory in a matter of seconds on typical computer configurations.

Depth-First Search

Depth-first search (see Figure 2*b*) expands nodes in a last-in, first-out order. Since it generates the same set of nodes as breadth-first search, the time complexity of depth-first search is also $O(b^d)$. Its advantage, however, lies in its space efficiency. Since depth-first search stores only the current search path, and the maximum length of this path is only d nodes, the space complexity of depth-first search is only $O(d)$. As a practical matter, depth-first search is time-limited rather than space-limited.

However, the disadvantage of depth-first search is that in general, in order to terminate it requires an arbitrary cutoff depth. Without such a cutoff depth, the first path could be explored forever, cycling back in an infinite loop to revisit previous states. Although the ideal cutoff is the solution depth d, this value is almost never known until the problem is solved. If the chosen cutoff depth c is less than the solution depth d, the algorithm will terminate before reaching a solution, whereas if c is greater than d, a large price, $O(b^c)$, is paid in terms of execution time, and the first solution found may not be optimal.

Depth-First Iterative Deepening

An algorithm that suffers neither the drawbacks of breadth-first nor depth-first search is called *depth-first iterative deepening*. Depth-first iterative deepening (DFID) first performs a depth-first search to depth 1, then conducts a complete depth-first search to depth 2, and continues executing depth-first searches, increasing the depth cutoff by 1 at each iteration, until a solution is found (see Figure 2*c*).

Since it never generates a deeper node until all shallower nodes have been generated, the first solution found by DFID is guaranteed to be optimal. Furthermore, since at any given point it is executing a depth-first search, the space complexity of DFID is only $O(d)$. Finally, although it appears that DFID wastes a great deal of time in the iterations prior to the successful one, the asymptotic time complexity of DFID is $O(b^d)$. The intuitive reason for this is that since the number of nodes at a given level of the tree grows exponentially with depth, almost all the time is spent in the deepest level, even though shallower levels are generated an arithmetically increasing number of times.

It can be proven by a simple adversary argument that any brute-force search algorithm must take $O(b^d)$ time. Furthermore, a simple information-theoretic argument shows that any algorithm that takes $O(b^d)$ time must use $O(d)$ space. These facts imply that DFID is asymptotically optimal in terms of time and space among all brute-force search algorithms that find optimal solutions on a tree (Korf 1985).

Bidirectional Search

One final brute-force search algorithm, which requires some additional structure in the problem space, is called bidirectional search (Pohl 1971). Bidirectional search executes two breadth-first searches, one from the initial state and one from the goal state, until a common state is found between the two search frontiers. The path from the initial state is then concatenated with the inverse of the path from the goal state to form the complete solution path.

Bidirectional search requires an explicit goal state rather than a goal criterion; and either the operators of the problem space must be invertible, or backward chaining must be feasible. Assuming that the comparisons for identifying common states can be done in constant time per node by using a technique such as hashing, the time complexity of bidirectional search is $O(b^{d/2})$ since each search need proceed to only half the solution depth. Since at least one of the search frontiers must be stored in order to find a common state, the space complexity of bidirectional search is also $O(b^{d/2})$.

HEURISTIC SEARCH

All brute-force algorithms suffer in efficiency from the fact that they are essentially blind searches; they use no domain knowledge to guide the choice of which nodes to expand next. Heuristic search takes advantage of the fact that most problem spaces provide, at relatively small computational cost, some information that distinguishes among states in terms of their likelihood of leading to a goal state. This information is called a *heuristic evaluation function.*

In the context of a single-agent problem, a heuristic evaluation function is a function from a pair of states to a number that estimates the distance in the problem space between the two states. For example, in navigating from one point to another in a network of roads, the airline distance between a pair of locations gives a rough estimate of the distance in the road network between the states, at a small computational cost. Another example is a common heuristic function for the Eight Puzzle, called Manhattan distance: For each tile not in its goal position, the distance in number of moves along the grid from its goal position is computed, and these values are summed over all tiles. The Manhattan-distance heuristic is an estimate of the number of moves required to get from one state to another, and it can be computed much faster than actually solving the problem and counting the moves. If we fix the goal state, then a heuristic evaluation becomes a function of one state, computing the distance to the given goal state.

A number of different algorithms, including best-first search, A*, and iterative-deepening-A*, make use of heuristic evaluation functions.

Best-First Search

Best-first search, one of the simplest and most general heuristic search algorithms, always expands next the node that appears most promising according to some heuristic evaluation function. The algorithm maintains two data structures, the Open and Closed lists. The Closed list contains those nodes already visited in the search, and disallows revisiting old nodes, thereby preventing infinite loops. The Open list contains the currently active states, in order of their heuristic values, from which the next node to expand is selected. The algorithm begins with just the initial state on the Open list. At each cycle, the best node on the Open list is expanded, generating all of its successors, and is then placed on the Closed list. After eliminating those successors already on the Closed list, the heuristic evaluation function is applied to the remaining successors, and they are placed on the Open list in order of their heuristic values. The algorithm continues until a goal state is reached.

Best-first search guarantees eventually finding a goal state in any finite problem space. Unfortunately, however, best-first search is not guaranteed to find a shortest path to the goal. This is due to the fact that selection of the next node to be expanded is based solely on the estimate of its distance to the goal and does not take into account its distance from the initial state. Thus, given two states, one of which is a long way from the initial state but has a slightly shorter estimate of distance to the goal, and another that is very close to the initial state but has a slightly longer estimate of distance to the goal, best-first search will always choose to expand next the state with the shorter estimate. The A* algorithm remedies this drawback.

A Algorithm*

A* is a best-first search algorithm in which the figure of merit associated with a node is not just the heuristic estimate but rather $f(n) = g(n)+h(n)$, where $g(n)$ is the actual cost of the path from the initial state to node n, and $h(n)$ is the heuristic estimate of the cost of the cheapest path from node n to a goal node. In other words, $f(n)$ is an estimate of the total cost of the cheapest solution path going through node n. At each point the node with the lowest f value is chosen for expansion next, until finally a goal node is chosen for expansion.

An important result is that A* will always find an optimal path to a goal if the heuristic function $h(n)$ never overestimates the actual distance to the goal (Hart & Nilsson 1968). For example, since airline distance never overestimates actual highway distance, A* with the airline-distance heuristic will find optimal solutions to the road-navigation problem. Similarly, the Manhattan-distance heuristic never overestimates the distance to the goal in the Eight Puzzle, and therefore the first solution found by A* using Manhattan distance for $h(n)$ will be optimal.

This result is most clearly evident if the cost function $f(n)$ is *monotonic*—i.e. never decreases along a path away from the initial state. Monotonicity is equivalent to the heuristic function $h(n)$ being *consistent*, or obeying the triangle inequality of metrics. This condition is not really a restriction, since given a nonmonotonic cost function, a monotonic one can easily be constructed by assigning the value of a node to be the maximum value of the function along the path from the initial state to that node (Méro 1984). In either case, since A* expands paths in nondecreasing order of cost, once it expands a goal node, it will have found a lowest-cost path to a goal.

A less-widely known property of A* is that it makes the most efficient use of a given heuristic function in the following sense: Among all algorithms that use a given consistent heuristic function $h(n)$ and that find an optimal solution, A* expands the fewest number of nodes, up to tie-breaking among nodes of equal cost (Dechter & Pearl 1985a). The actual number of nodes expanded by A* depends on the accuracy of the heuristic function. For example, if the heuristic function exhibits constant absolute error, the number of nodes expanded is a linear function of the solution depth (Pohl 1970), whereas if the heuristic is subject to constant relative error (a much more realistic assumption), the number of node expansions is exponential in the solution depth (Gaschnig 1979). In general, the number of nodes expanded by A* is an exponential function of the typical error in the heuristic estimate (Pearl 1984).

The main drawback of A*, and indeed of any best-first search, is its memory requirement. Since the entire Open list must be saved, A* is severely space-limited in practice and is no more practical than breadth-first search on current machines. This limitation is removed by an algorithm called iterative-deepending-A*.

*Iterative-Deepening-A**

In the same way that depth-first iterative-deepening defeated the space complexity of breadth-first search, iterative-deepening-A* (IDA*) drastically reduces the memory requirement of A* without sacrificing optimality of the solutions found (Korf 1985). Each iteration of the algorithm is a complete depth-first search that keeps track of the cost, $f(n) = g(n)+h(n)$, of each node generated. As soon as this cost exceeds some threshold, that branch is cut off, and the search backtracks to the most recently generated node. The cost threshold starts with the heuristic estimate of the initial state and in each successive iteration is increased to the minimum value that exceeded the previous threshold.

Since at any point *IDA** is performing a depth-first search, the memory requirement of the algorithm is linear in the solution depth. In addition, it can easily be shown that if the cost function never overestimates, the first solution found by *IDA** will be optimal. Finally, by an argument similar to that presented for depth-first iterative-deepening, it can be shown that *IDA** expands the same number of nodes, asymptotically, as A*, provided that the number of nodes grows exponentially with solution depth and that the number of duplicate nodes in the search tree is relatively small. In practice, both of these assumptions are realistic. These facts, together with the optimality of A*, show that in practice *IDA** is asymptotically optimal in terms of time and space over all heuristic search algorithms that find optimal solutions on a tree.

An additional benefit of *IDA** is that it is easier to implement and runs faster per node than A*. This is because it is a simple depth-first search and does not incur the overhead of managing an Open list. In experiments *IDA** has proven to be the only algorithm capable of finding optimal solutions to the Fifteen Puzzle within practical time and space limits.

AND/OR GRAPHS

Another type of problem space is an AND/OR graph. In an AND/OR graph, the nodes typically represent complete subproblems. The root node of the tree represents the original problem to be solved, and the leaf or terminal nodes of the graph represent solved problems. The edges represent problem-reduction operators, which decompose a given problem into a set of subproblems. If only one of the subproblems needs to be solved to solve the main problem, the node is called an OR node. If all the subproblems must be solved to solve the main problem, the node is called an AND node. A problem space containing both AND and OR nodes is called an AND/OR graph. A solution to an AND/OR graph is a subgraph that contains the root node, one branch from every OR node, all the branches from every AND node, and only goal states at the terminal nodes.

Generally, AND/OR graphs are suited to problems for which the final solution is most conveniently represented as a tree or a graph, rather than an ordered sequence of actions. Strategy-seeking tasks are typical examples of this class of problems, where the AND links represent changes in the problem situation caused by external, uncontrolled conditions and the OR links represent alternative reactions to such changes. In planning, the uncontrolled conditions could be possible outcomes of an uncertain event or results of a given test. In games, those conditions are created by the moves of the adversary. In program-synthesis, they may be the results of computations on unspecified data.

Another important class of problems suitable for AND/OR graph representations includes cases in which the solution required is a partially ordered sequence of actions. In symbolic integration, for example, certain legal transformations (e.g. integration by parts, long division, etc) split the integrand into sums of expressions to be integrated separately in any order. The set of applicable transformations will be represented as OR links emanating from the node representing the integrand; the AND links represent individual summands within the integrand, all of which must eventually be integrated.

The tasks of logical reasoning and theorem proving also give rise to AND/OR structures. We begin with a set of axioms and a set of inference rules that allow us, at each step, to deduce a new statement from a subset of axioms and previously deduced statements. The new statement is added to the database, and the process continues until the desired conclusion (e.g. the theorem) is derived. The solution object pursued by the search is a plan specifying, at each step, which of the inference rules is to be applied to which subset of statements in the database and what the deduced statement is. This plan is best structured as an unordered tree because when a certain conclusion is derived from a given subset of statements the internal order in which these statements were themselves derived is of no consequence as long as they reside in the database at the appropriate time. Thus the solution structure is a tree and the appropriate search space would be an AND/OR graph.

AND/OR graphs are also suitable for representing problems for which the solution sought is an ordered sequence of actions, as long as the search for some subsequences that make up the solution can be conducted in any order. A classical example is the Towers-of-Hanoi puzzle (Pearl 1984) where the three main subgoals (i.e. clearing the largest disk, moving that disk to a given peg, and placing the other disks on top of the largest one) must be executed in a certain order but the search for their solutions can be conducted in any order.

Like state-space graphs, AND/OR graphs lend themselves to systematic search methods such as breadth-first, depth-first, and various forms of heuristic best-first algorithms. The basic rationale behind heuristic search methods is that the examination of various solution candidates (OR links) should start with the candidate most likely to succeed, while the examination of subgoals within each candidate (AND links) should begin with the one most likely to fail. Numerical estimates of these likelihoods are often used to guide the search so that a solution graph can be found after exploring only a small portion of the AND/OR graph that underlies a given problem. The algorithm AO*, for example, estimates the costs of the solution graphs rooted at the various candidate nodes and is guaranteed to find a cheapest solution if all cost estimates are optimistic (Bagchi & Mahanti 1983).

GAME TREES

An important special case of an AND/OR tree is a two-player game tree, the classic example of which is chess. In a game tree, the nodes represent particular game situations, and the edges represent legal moves. The root node represents the initial game situation, and the leaves represent situations that are won, lost, or drawn positions. The edges descending from the root represent the legal moves of the first player to move, and thereafter alternate levels of the tree represent moves for one player or the other. Since the problem solver has control over the moves of only one player, the game is represented as an AND/OR tree. From the perspective of one of the players his moves represent OR nodes because the choice of only one winning move guarantees a win. His opponent's moves represent AND nodes since he must consider all his opponent's responses and secure a win in each one. The task in a two-player game is to force the path that is traversed through the state space to end in a winning state—in other words, to find a solution subtree that has all wins at the leaves.

Game Searching Algorithms

Solving a game tree means labeling the root node as win, loss or, possibly, draw. The labeling can be done recursively, starting from the leaves (where the labels are determined externally) and backing up toward the root. This is usually done by a depth-first algorithm that traverses the tree from left to right but skips all nodes that cannot provide useful information. For example, as soon as one successor of a node is found to be a win, that node can be labeled a win without solving the rest of its children.

In practical games such as chess or checkers, the tree is too deep for finding winning moves and, instead, moves are determined by heuristically evaluating the strength of board positions a few moves ahead. The prevailing strategy, called minimax, is to treat the estimated evaluation of nodes on the search frontier as if these were true terminal payoffs given to the player upon reaching these nodes. This assumption allows the recursive assignment of values to any node in the tree, based on those of its successors. The player's positions are valued at the maximum value of their successors and the opponents' positions are valued at the minimum of their successors.

By far the most popular algorithm for searching game trees is Alpha-Beta pruning. Other search algorithms that have been implemented and analyzed are SSS* and SCOUT.

Alpha-Beta is a depth-first algorithm that, in order to determine the minimax value of the root of a game tree, traverses the tree in a predetermined order (e.g. left to right) and uses the information obtained to cut off branches of the tree that could no longer influence the minimax value of the root (Knuth & Moore 1975). Such cut-off branches consist of options available at game positions that the player, having better choices, will surely avoid. For example, in the tree shown in Figure 3, where squares represent moves for MAX and circles represent moves for MIN, the rightmost terminal node need not be evaluated since it cannot affect the value of the root.

SSS* is a best-first search procedure that maintains upper bounds on the values of partially developed candidate strategies, selecting the most promising one for further development. A strategy specifies one response of the player to each of the opponent's moves. The development process continues until one game strategy is fully developed, at which point it represents the optimal strategy (Stockman 1979). Like its A* counterpart for single-player games, SSS* is optimal in terms of the average number of nodes examined; but its superior pruning power is more than offset by the substantial storage space and bookkeeping required.

SCOUT evaluates a position J by first computing the minimax value of its leftmost successor J_1 and assigning it to a temporary variable v. It then scouts the remaining successors from left to right, testing whether any attains a value higher than v, assuming MAX is to move at J. (Testing such suppositions can be performed faster than actually evaluating J_{i}.) If a successor J_i passes the test, its value is then computed and assigned to v for use in subsequent scouting tests, otherwise; J_i is exempted from evalu-

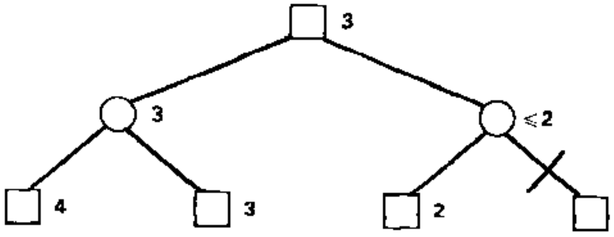


Figure 3 Alpha-Beta pruning.

ation. (In contrast, Alpha-Beta pursues the evaluation of J_i initially and stops the evaluation only upon receiving evidence that J_i will not evaluate to more than v.) When all successors have been either tested and evaluated or tested and found unworthy of evaluation, the last value of v obtained is returned as the value of J.

PERFORMANCE OF GAME SEARCHING ALGORITHMS A lower bound on the number of nodes that must be examined by any game searching algorithm can be established by the following argument. Evaluating a game with value V amounts to verifying that, no matter how the opponent reacts, the player can guarantee a payoff of at least V and, simultaneously, that no matter how our player acts, the opponent can prevent him from getting more than V. These two verification tasks require the display of two adversary strategies with equal values, one for the player and one for the opponent. Since each strategy branches out once in every two moves of the game, the number of nodes contained in a typical strategy is roughly the square root of the number of nodes in the game tree. Therefore, every search strategy that evaluates a game tree must examine at least twice the square root of the number of nodes in the entire game tree.

In practice, this lower bound of twice-the-square-root is rarely achieved, because we do not know in advance which of the partially exposed strategies are in fact compatible. Many incompatible strategies are partially searched, only to be abandoned when more of their leaves are exposed. To guide the search toward finding two compatible strategies, one needs to know, at each game configuration, the best next move for each player. If no information is available regarding the relative merits of the pending moves, roughly the 3/4-root of the number of nodes in the game tree will, on the average, be explored. As the move-rating information becomes more accurate, the number of nodes examined gradually approaches the absolute square-root bound (Pearl 1984).

The average pruning power of various game-playing strategies is usually measured by a parameter called the "effective branching factor." Formally, if d stands for the maximal depth reached by an algorithm A, and $I_A(d)$ stands for the average number of nodes generated during the search, then the effective branching factor, B_A, is defined by

$$B_A = \lim_{d \to \infty} [I_A(d)]^{1/d}.$$

This definition extracts the *basis* of the dominant exponential term in the expression of $I_A(d)$. Thus, the effective branching factor measures the relative increase in average complexity due to extending the search depth by one extra level or, equivalently, it measures the average number of branches explored by an algorithm from a typical node of the search space.

Theoretical analyses of game-searching strategies usually assume uniform, b-ary game-trees, searched to depth d, with random distinct values assigned to nodes at the search frontier (Knuth & Moore 1975). Based on this model, it can be shown (Pearl 1982) that the effective branching factor of the Alpha-Beta pruning algorithm (as well as that of SCOUT and SSS*) is given by

$$B = \frac{\xi_b}{1-\xi_b} \approx b^{3/4},$$

where ξ_b is the unique positive root of the equation

$$x^b + x - 1 = 0.$$

Moreover, this branching factor is the best achievable by any game-searching algorithm.

Roughly speaking, only a fraction $(b^{3/4})$ of the b legal moves available from each game position will be explored by Alpha-Beta. Alternatively, for a given search-time allotment, the Alpha-Beta pruning allows the search depth to be increased by a factor $\log b/\log B \approx 4/3$ over that of an exhaustivc minimax search.

Under perfect ordering of successors, Alpha-Beta examines a total of $2b^{d/2}-1$ game positions. Thus, we have:

1. $B = b$ for exhaustive search;
2. $B \approx b^{3/4}$ for Alpha-Beta with random ordering; and
3. $B = b^{1/2}$ for Alpha-Beta with perfect ordering.

It is important to mention that the branching factor only captures the asymptotic growth rate of a search strategy as the search depth increases indefinitely; it does not reflect the size of nonexponential factors in $I(d)$, regardless of how large they are. However, an exact evaluation of the average performances of three game-searching strategies shows (Pearl 1984) that the ratio $I(d)/B^d$ is in fact fairly small; it remains below 5 over wide ranges of b and d ($b \leq 20$, $d \leq 20$).

CONSTRAINT-SATISFACTION PROBLEMS

In the Eight Puzzle, the goal state is given explicitly. In other problems, however, the goal state is not given explicitly but rather is specified by a set of constraints that must be satisfied by any goal state. Such a problem is called a constraint-satisfaction problem (CSP). For example, the Eight-Queens problem is to place eight queens on a chessboard so that no two queens are attacking each other along a row, column, or diagonal. The task here is not to find a sequence of actions but rather to exhibit a particular state that satisfies all the constraints simultaneously. Since this problem is too difficult to be solved in a single step, it must be broken down into a series of simpler steps. In the Eight-Queens problem, this typically involves defining states that are partial assignments of queens to board positions, and operators that consist of placing an additional queen on the board. Thus, even though the sequence of operators required to solve the problem is not of interest, the problem is still formulated as a search through a problem space.

Formally, a constraint-satisfaction problem involves a set of n variables $X_1, \ldots, X_n$, each represented by its domain values, $R_1, \ldots, R_n$ and a set of constraints. A constraint $C_i(X_{i_1}, \ldots, X_{i_j})$ is a subset of the Cartesian product $R_{i_1} \times \cdots \times R_{i_j}$ that specifies which values of the variables are compatible with each other. A solution is an assignment of values to all the variables that satisfy all the constraints, and the task is to find one or all solutions. A constraint is usually represented by the set of all tuples permitted by it. A *binary CSP* is one in which all the constraints are binary—i.e. involve only pairs of variables. A binary CSP can be associated with a *constraint graph* in which nodes represent variables and arcs connect pairs of variables that are constrained explicitly.

For example, the constraint graph associated with the Eight-Queens problem is a complete graph, because every queen placed on the board influences the permitted positions available for any other queen. Many CSPs, however, are characterized by sparse constraint graphs, the topology of which can be instrumental in pruning the search for a solution. For example, the task of choosing numerical values for the variables X, Y, and Z, such that Z divides both X and Y, defines a chain-structured constraint graph X–Z–Y. A nonbinary CSP can be represented by a hypergraph, where the hyperarcs connect subsets of variables tied by an explicit constraint.

The topology of the constraint graph can sometimes be used to identify easy solution methods. The best known and most useful result in this direction is that binary CSPs whose constraint graph is a tree can be optimally solved in $O(nk^2)$ time where n is the number of variables and k is the number of values for each variable (Dechter & Pearl 1985b; Freuder 1982). Starting from the leaves toward the root, we simply delete from every node those values that do not have at least one match for each of its successors. If any of the nodes ends up empty, the problem has no solution. Otherwise, we trace any of the remaining values, from the root down, and issue a consistent solution.

The search space associated with a CSP has states being consistent assignments of values to subsets of variables. A state $(X_1 = x_1, \ldots, X_i = x_i)$

can be extended by any consistent assignment to any of the remaining variables. States at depth n represent solutions to the problem, namely n-tuples satisfying all the constraints. If the order by which variables are instantiated is fixed, then the search space is limited to contain only states in that specific order.

The most common algorithm for solving CSPs is a depth-first search called *backtracking*. In its most primitive version, backtracking traverses the variables in a predetermined order, provisionally assigning consistent values to a subsequence $(X_1, \ldots, X_i)$ of variables and attempting to append to it a new instantiation such that the whole set is consistent. (An assignment of values to a subset of the variables is *consistent* if it satisfies all the constraints applicable to this subset.) If no consistent assignment can be found for the next variable, X_{i+1}, a dead-end situation occurs, and the algorithm backtracks to one of the earlier variables and changes its assignment.

Since backtracking is the main control strategy in many AI applications (e.g. theorem proving in PROLOG, Truth Maintenance Systems), numerous schemes have been devised for improving its performance. These schemes, often termed "intelligent backtracking," "selective backtracking," and "dependency-directed backtracking," can be classified as follows:

1. *Look-ahead schemes:* affecting the decision of what variable to instantiate or what value to assign to the next variable among all the consistent choices available.
   a. *Variable ordering:* An attempt is made to instantiate that variable which would render the rest of the problem easy to solve. Usually, the variable participating in the highest number of constraints is selected (Freuder 1982; Purdom 1983).
   b. *Value ordering:* An attempt is made to assign a value that maximizes the number of options available for future assignments (Dechter & Pearl 1987; Haralick & Elliott 1980).
2. *Look-back schemes:* affecting the decision of where and how to go in case of a dead-end situation. Look-back schemes are centered around two fundamental ideas:
   a. *Go-back to source of failure:* An attempt is made to detect and change previous decisions that caused the dead end without changing decisions irrelevant to the dead end.
   b. *Constraint recording:* The reasons for the dead end are recorded so that the same conflicts will not arise again in the continuation of the search.

The best-known "go-back" scheme is Gaschnig's (Gaschnig 1979) "backjumping." Backjumping marks each value of the dead-end variable with the age of the oldest ancestor forbidding that value and then jumps back to the youngest among these ancestors. A simpler version of backjumping, still yielding remarkable performance improvement, jumps to the youngest ancestor constraining (via a constraint arc) the dead-end variable.

Constraint recording can be implemented either by preprocessing the problem prior to search or by recording constraints dynamically as they are discovered during the search. The most common preprocessing techniques are arc consistency and path consistency (Freuder 1982; Mackworth 1977; Montanari 1974). Arc consistency involves deleting from the domain of some variables those values that find no match in other, directly connected variables. Path consistency consists of recording sets of forbidden value pairs, if these pairs find no match at a third variable. Preprocessing a CSP for full path consistency may be quite expensive, requiring at least $O(n^3k^3)$ operations, while many of the forbidden pairs discovered may not be encountered in the actual search. For that reason, *learning* techniques have been devised (Dechter 1986) that are more efficient in both time and storage. These perform partial arc and path consistencies by recording only those constraints that emanate from paths discovered during the search.

Another method devised for improving the performance of backtracking, called the *cycle-cutset* method (Dechter & Pearl 1987), is based on identifying a set of nodes that, once removed, would render the constraint-graph cycle-free—i.e. tree-structured. The improvement relies on the following observation: If, in the course of a backtrack search, we remove from the constraint graph the nodes corresponding to instantiated variables and find that the remaining subgraph is a tree, then the rest of the search can be completed in linear time. Thus, rather than continue the search blindly, we invoke a tree-searching algorithm tailored to the topology of the remaining subproblem.

This observation also offers a theoretical upper bound on the complexity of CSPs. If c stands for the size of some cycle cutset found in the constraint graph, and if we choose the cutset variables to be instantiated first, then the complexity of the search is at most $O(nk^c)$. Thus, compared with the exponential complexity $O(k^n)$ usually associated with backtrack search, this method may yield substantial savings when the constraint graph is sparse.

CONCLUSIONS

Search techniques are extremely general problem-solving methods. All that is required to formulate a search problem is a set of states, a set of operators, an initial state, and a goal criterion. The cost of this generality, however, is exponential complexity. In order to perform searches more effectively, additional knowledge about the problem, such as heuristic evaluation functions, must be brought to bear. Much of the research in search techniques is focussed on devising algorithms to make use of this additional knowledge, and analyzing the effect of such knowledge on the complexity of those algorithms.

ACKNOWLEDGMENTS

This work was supported in part by NSF Grant DCR 85-01234 to the first author, and by NSF Grant IST 85-15302, an NSF Presidential Young Investigator Award, an IBM Faculty Development Award, and a grant from the Delco Electronics Corporation to the second author. The authors would like to thank Lillian Larijani for her comments on an earlier draft of this manuscript, and Greg Korf for proofreading.

Literature Cited

Bagchi, A., Mahanti, A. 1983. Admissible heuristic search in AND/OR graphs. *Theor. Comput. Sci.* 2(24): 207–19

Dechter, R. 1986. Learning while searching in constraint-satisfaction problems. In *Proc. AAAI-86, Philadelphia*, pp. 178–83

Dechter, R., Pearl, J. 1985a. Generalized best-first search strategies and the optimality of A*. *J. Assoc. Comput. Mach.* 32(3): 505–36

Dechter, R., Pearl, J. 1985b. The anatomy of easy problems: a constraint-satisfaction formulation. In *Proc. Int. Joint Conf. Artif. Intell., Los Angeles, Calif.*, pp. 1066–72

Dechter, R., Pearl, J. 1987. Network-based heuristics for constraint-satisfaction problems. To appear in *Artificial Intelligence*

Dechter, R., Pearl, J. 1987. The cycle-cutset method for improving search performance in AI applications. In *Proc. 3rd IEEE Conf. AI Applic., Orlando, Fla.*, pp. 224–30

Freuder, E. C. 1982. A sufficient condition for backtrack-free search. *J. Assoc. Comput. Mach.* 29(1): 24–32

Gaschnig, J. 1979. *Performance measurement and analysis of certain search algorithms.* PhD thesis. Dept. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, Pa.

Haralick, R. M., Elliott, G. L. 1980. Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.* 14: 263–313

Hart, P. E., Nilsson, N. J., Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybernet.* 4(2): 100–7

Knuth, D. E., Moore, R. E. 1975. An analysis of Alpha-Beta pruning. *Artif. Intell.* 6(4): 293–326

Korf, R. E. 1985. Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell.* 27(1): 97–109

Mackworth, A. K. 1977. Consistency in networks of relations. *Artif. Intell.* 8(1): 99–118

Mero, L. 1984. A heuristic search algorithm with modifiable estimate. *Artif. Intell.* 23(1): 13–27

Montanari, U. 1974. Networks of constraints: fundamental properties and applications to picture processing. *Inform. Sci.* 7: 95–132

Newell, A., Simon, H. A. 1972. *Human Problem Solving.* Englewood Cliffs, NJ: Prentice-Hall

Pearl, J. 1982. The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. *C.A.C.M.* 25(8): 559–64

Pearl, J. 1984. *Heuristics.* Reading, Mass: Addison-Wesley. 382 pp.

Pohl, I. 1971. Bi-directional search. In *Machine Intelligence 6*, ed. B. Meltzer, D. Michie, pp. 219–36. NY: American Elsevier

Pohl, I. 1970. First results on the effect of error in heuristic search. In *Machine Intelligence 5*, ed. B. Meltzer, D. Michie, pp. 219–36. NY: American Elsevier

Purdom, P. W. 1983. Search rearrangement backtracking and polynomial average time. *Artif. Intell.* 21(1,2): 117–33

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

Annual Review of Computer Science
Volume 2, 1987

CONTENTS

ARTIFICIAL INTELLIGENCE

HARDWARE

SOFTWARE

THEORY

APPLICATIONS

INDEXES