22

Artificial Intelligence Search Algorithms

22.1 Introduction **22**-1
22.2 Problem Space Model **22**-3
22.3 Brute-Force Search **22**-3
Breadth-First Search • Uniform-Cost Search • Depth-First Search • Depth-First Iterative-Deepening • Bidirectional Search • Frontier Search • Disk-Based Search Algorithms • Combinatorial Explosion
22.4 Heuristic Search **22**-7
Heuristic Evaluation Functions • Pattern Database Heuristics • Pure Heuristic Search • A* Algorithm • Iterative-Deepening-A* • Depth-First Branch-and-Bound • Complexity of Finding Optimal Solutions • Heuristic Path Algorithm • Recursive Best-First Search
22.5 Interleaving Search and Execution **22**-11
Minimin Search • Real-Time-A* • Learning-Real-Time-A*
22.6 Game Playing **22**-13
Minimax Search • Alpha-Beta Pruning • Other Techniques • Significant Milestones • Multiplayer Games, Imperfect and Hidden Information
22.7 Constraint-Satisfaction Problems **22**-16
Chronological Backtracking • Intelligent Backtracking • Constraint Recording • Local Search Algorithms
22.8 Research Issues and Summary **22**-19
Research Issues • Summary
22.9 Further Information **22**-20
Defining Terms **22**-20
Acknowledgment **22**-21
References **22**-21

Richard E. Korf
University of California, Los Angeles

22.1 Introduction

Search is a universal problem-solving mechanism in artificial intelligence (AI). In AI problems, the sequence of steps required for the solution of a problem is not known a priori, but often must be determined by a trial-and-error exploration of alternatives. The problems that have been addressed by AI search algorithms fall into three general classes: single-agent pathfinding problems, game playing, and constraint-satisfaction problems.

Classic examples in the AI literature of single-agent pathfinding problems are the sliding-tile puzzles, including the 3 × 3 eight puzzle (see Figure 22.1) and its larger relatives the 4 × 4 fifteen

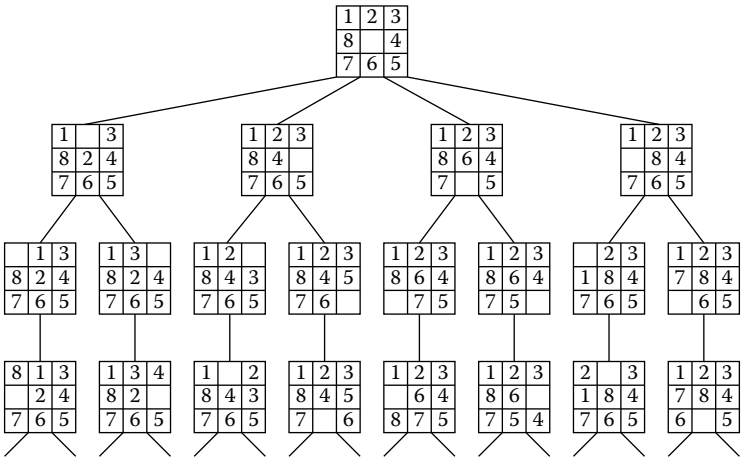

FIGURE 22.1 Eight puzzle search tree fragment.

puzzle and 5×5 twenty-four puzzle. The eight puzzle consists of a 3×3 square frame containing eight numbered square tiles and an empty position called the blank. The legal operators are to slide any tile that is horizontally or vertically adjacent to the blank into the blank position. The problem is to rearrange the tiles from some random initial configuration into a particular desired goal configuration. The sliding-tile puzzles are common testbeds for research in AI search algorithms because they are very simple to represent and manipulate, yet finding optimal solutions to the $N \times N$ generalization of the sliding-tile puzzles is NP-complete [38]. Another well-known example of a single-agent pathfinding problem is Rubik's cube. Real-world examples include theorem proving, the traveling salesman problem, and vehicle navigation. In each case, the task is to find a sequence of operations that maps an initial state to a goal state.

A second class of search problems include games, such as chess, checkers, Othello, backgammon, bridge, poker, and Go. The third category is constraint-satisfaction problems, such as the N-queens problem or Sudoku. The task in the N-queens problem is to place N queens on an $N \times N$ chessboard, such that no two queens are on the same row, column, or diagonal. The Sudoku task is to fill each empty cell in a 9×9 matrix with a digit from zero through nine, such that each row, column, and nine 3×3 submatrices contain all the digits zero through nine. Real-world examples of constraint-satisfaction problems are ubiquitous, including boolean satisfiability, planning, and scheduling applications.

We begin by describing the problem-space model on which search algorithms are based. Brute-force searches are then considered including breadth-first, uniform-cost, depth-first, depth-first iterative-deepening, bidirectional, frontier, and disk-based search algorithms. Next, we introduce heuristic evaluation functions, including pattern databases, and heuristic search algorithms including pure heuristic search, the A* algorithm, iterative-deepening-A*, depth-first branch-and-bound, the heuristic path algorithm, and recursive best-first search. We then consider single-agent algorithms that interleave search and execution, including minimin lookahead search, real-time-A*, and learning-real-time-A*. Next, we consider game playing, including minimax search, alpha-beta pruning, quiescence, iterative deepening, transposition tables, special-purpose hardware, multiplayer games, and imperfect and hidden information. We then examine constraint-satisfaction algorithms, such as chronological backtracking, intelligent backtracking techniques, constraint recording, and local search algorithms. Finally, we consider open research problems in this area. The performance of these algorithms, in terms of the costs of the solutions they generate, the amount of time the algorithms take to execute, and the amount of computer memory they require are of central concern

throughout. Since search is a universal problem-solving method, what limits its applicability is the efficiency with which it can be performed.

22.2 Problem Space Model

A problem space is the environment in which a search takes place [32]. A problem space consists of a set of states of the problem, and a set of operators that change the state. For example, in the eight puzzle, the states are the different possible permutations of the tiles, and the operators slide a tile into the blank position. 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 whatever initial permutation the puzzle starts out in, and the goal state is a particular desired permutation. The problem-solving task is to find a sequence of operators that map the initial state to a goal state. In the eight puzzle the goal state is given explicitly. In other problems, such as the N-Queens Problem, the goal state is not given explicitly, but rather implicitly specified by certain properties that must be satisfied by any goal state.

A problem-space graph is often used to represent a problem space. The states of the space are represented by nodes of the graph, and the operators by edges between nodes. Edges may be undirected or directed, depending on whether their corresponding operators are reversible or not. The task in a single-agent path-finding problem is to find a path in the graph from the initial node to a goal node. Figure 22.1 shows a small part of the eight puzzle problem-space graph.

Although most problem spaces correspond to graphs with more than one path between a pair of nodes, for simplicity they are often represented as trees, where the initial state is the root of the tree. The cost of this simplification is that any state that can be reached by two different paths will be represented by duplicate nodes in the tree, increasing the size of the tree. The benefit of a tree is that the absence of multiple paths to the same state greatly simplifies many search algorithms.

One feature that distinguishes AI search algorithms from other graph-searching algorithms is the size of the graphs involved. For example, the entire chess graph is estimated to contain over 10^{40} nodes. Even a simple problem like the Twenty-four puzzle contains almost 10^{25} nodes. As a result, the problem-space graphs of AI problems are never represented explicitly by listing each state, but rather are implicitly represented by specifying an initial state and a set of operators to generate new states from existing states. Furthermore, the size of an AI problem is rarely expressed as the number of nodes in its problem-space graph. Rather, the two parameters of a search tree that determine the efficiency of various search algorithms are its branching factor and its solution depth. The branching factor is the average number of children of a given node. For example, in the eight puzzle the average branching factor is $\sqrt{3}$, or about 1.732. The solution depth of a problem instance is the length of a shortest path from the initial state to a goal state, or the length of a shortest sequence of operators that solves the problem. For example, if the goal were in the bottom row of Figure 22.1, the depth of the problem instance represented by the initial state at the root would be three moves.

22.3 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 state description, a set of legal operators, an initial state, and a description of the goal state. The most important brute-force techniques are breadth-first, uniform-cost, depth-first, depth-first iterative-deepening, bidirectional, and frontier search. In the descriptions of the algorithms in the following text, to generate a node means to create the data structure corresponding to that node, whereas to expand a node means to generate all the children of that node.

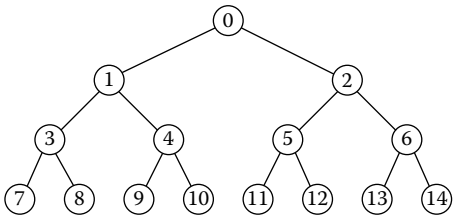

FIGURE 22.2 Order of node generation for breadth-first search.

22.3.1 Breadth-First Search

Breadth-first search expands nodes in order of their depth from the root, generating one level of the tree at a time until a solution is found (see Figure 22.2). It is most easily implemented by maintaining a first-in first-out queue of nodes, initially containing just the root, and always removing the node at the head of the queue, expanding it, and adding its children to the tail of the queue.

Since it never generates a node in the tree until all the nodes at shallower levels have been generated, breadth-first search always finds a shortest path to a goal. Since each node can be generated in constant time, the amount of time used by breadth-first search is proportional to the number of nodes generated, which is a function of the branching factor b and the solution depth d. Since the number of nodes in a uniform tree at level d is b^d, the total number of nodes generated in the worst case is $b + b^2 + b^3 + \cdots + b^d$, which is $O(b^d)$, the asymptotic time complexity of breadth-first search.

The main drawback of breadth-first search is its memory requirement. Since each level of the tree must be stored 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-bound in practice, and will exhaust the memory available on typical computers in a matter of minutes.

22.3.2 Uniform-Cost Search

If all edges do not have the same cost, then breadth-first search generalizes to uniform-cost search. Instead of expanding nodes in order of their depth from the root, uniform-cost search expands nodes in order of their cost from the root. At each step, the next node n to be expanded is one whose cost $g(n)$ is lowest, where $g(n)$ is the sum of the edge costs from the root to node n. The nodes are stored in a priority queue. This algorithm is similar to Dijkstra's single-source shortest-path algorithm [7]. The main difference is that uniform-cost search runs until a goal node is chosen for expansion, while Dijkstra's algorithm runs until every node in a finite graph is chosen for expansion.

Whenever a node is chosen for expansion by uniform-cost search, a lowest-cost path to that node has been found. The worst-case time complexity of uniform-cost search is $O(b^{c/e})$, where c is the cost of an optimal solution, and e is the minimum edge cost. Unfortunately, it also suffers the same memory limitation as breadth-first search.

22.3.3 Depth-First Search

Depth-first search removes the space limitation of breadth-first and uniform-cost search by always generating next a child of the deepest unexpanded node (see Figure 22.3). Both algorithms can be implemented using a list of unexpanded nodes, with the difference that breadth-first search manages the list as a first-in first-out queue, whereas depth-first search treats the list as a last-in first-out stack. More commonly, depth-first search is implemented recursively, with the recursion stack taking the place of an explicit node stack.

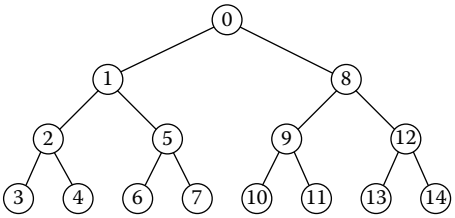

FIGURE 22.3 Order of node generation for depth-first search.

The advantage of depth-first search is that its space requirement is only linear in the maximum search depth, as opposed to exponential for breadth-first search. The reason is that the algorithm only needs to store a stack of nodes on the path from the root to the current node. The time complexity of a depth-first search to depth d is $O(b^d)$, since it generates the same set of nodes as breadth-first search, but simply in a different order. Thus, as a practical matter, depth-first search is time-limited rather than space-limited.

The primary disadvantage of depth-first search is that it may not terminate on an infinite tree, but simply go down the left-most path forever. For example, even though there are a finite number of states of the eight puzzle, the tree fragment shown in Figure 22.1 can be infinitely extended down any path, generating an infinite number of duplicate nodes representing the same states. The usual solution to this problem is to impose a cutoff depth on the search. Although the ideal cutoff is the solution depth d, this value is rarely known in advance of actually solving the problem. If the chosen cutoff depth is less than d, the algorithm will fail to find a solution, whereas if the cutoff depth is greater than d, a large price is paid in execution time, and the first solution found may not be an optimal one.

22.3.4 Depth-First Iterative-Deepening

Depth-first iterative-deepening (DFID) combines the best features of breadth-first and depth-first search [18,46]. DFID first performs a depth-first search to depth one, then starts over, executing a complete depth-first search to depth two, and continues to run depth-first searches to successively greater depths, until a solution is found (see Figure 22.4).

Since it never generates a node until all shallower nodes have been generated, the first solution found by DFID is guaranteed to be via a shortest path. Furthermore, since at any given point it is executing a depth-first search, saving only a stack of nodes, and the algorithm terminates when it finds a solution at depth d, the space complexity of DFID is only $O(d)$.

Although DFID spends extra time in the iterations before the one that finds a solution, this extra work is usually insignificant. To see this, note that the number of nodes at depth d is b^d, and each of these nodes are generated once, during the final iteration. The number of nodes at depth $d-1$ is b^{d-1},

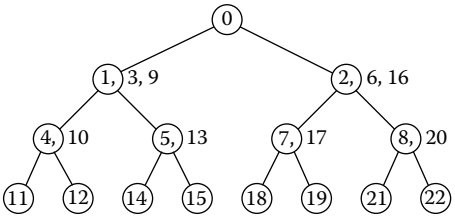

FIGURE 22.4 Order of node generation for depth-first iterative-deepening search.

and each of these are generated twice, once during the final iteration, and once during the penultimate iteration. In general, the number of nodes generated by DFID is $b^d + 2b^{d-1} + 3b^{d-2} + \cdots + db$. This is asymptotically $O(b^d)$ if b is greater than one, since for large values of d the lower order terms are insignificant. In other words, most of the work goes into the final iteration, and the cost of the previous iterations is relatively small. The ratio of the number of nodes generated by DFID to those generated by breadth-first search on a tree is approximately $b/(b-1)$. In fact, DFID is asymptotically optimal in terms of time and space among all brute-force shortest-path algorithms on a tree [18].

If the edge costs differ from one another, then one can run an iterative deepening version of uniform-cost search, where the depth cutoff is replaced by a cutoff on the $g(n)$ cost of a node. At the end of each iteration, the threshold for the next iteration is set to the minimum cost of all nodes generated but not expanded on the previous iteration.

On a graph with multiple paths to the same node, however, breadth-first search may be much more efficient than depth-first or depth-first iterative-deepening search. The reason is that a breadth-first search can easily detect all duplicate nodes, whereas a depth-first search can only check for duplicates along the current search path. Thus, the complexity of breadth-first search grows only as the number of states at a given depth, while the complexity of depth-first search depends on the number of paths of a given length. For example, in a square grid, the number of nodes within a radius r of the origin is $O(r^2)$, whereas the number of paths of length r is $O(3^r)$, since there are three children of every node, not counting its parent. Thus, in a graph with a large number of very short cycles, breadth-first search is preferable to depth-first search, if sufficient memory is available. For two approaches to the problem of pruning duplicate nodes in depth-first search, see [8,47].

22.3.5 Bidirectional Search

Bidirectional search is a brute-force algorithm that requires an explicit goal state instead of simply a test for a goal condition [36]. The main idea is to simultaneously search forward from the initial state, and backward from the goal state, until the two search frontiers meet. 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 still guarantees optimal solutions. Assuming that the comparisons for identifying a common state between the two frontiers can be done in constant time per node, by hashing for example, the time complexity of bidirectional search is $O(b^{d/2})$ since each search need only proceed to half the solution depth. Since at least one of the searches must be breadth-first in order to find a common state, the space complexity of bidirectional search is also $O(b^{d/2})$. As a result, bidirectional search is space bound in practice.

22.3.6 Frontier Search

Best-first searches, such as breadth-first, uniform-cost, and bidirectional search, store both a closed list of expanded nodes, and an open list of nodes that have been generated but not yet expanded. Another approach to the memory limitation of these algorithms, called frontier search, is to store only the open list, and delete nodes once they are expanded [25]. To keep from regenerating the parent of a node, frontier search stores with each node a bit for each operator that would generate an expanded node. This reduces the space complexity of a breadth-first search, for example, from the size of the problem space to the width of the problem space, or the maximum number of nodes at any given depth. For example, the size of a two dimensional grip graph is quadratic in the radius, while the width of the graph is only linear. Alternatively, the immediate parents of those nodes on the open list can be stored [49].

The main drawback of frontier search is that once the search completes, the solution path is not available, since the expanded nodes have been deleted. One way to reconstruct the solution path is to perform a bidirectional frontier search, from both the initial state and the goal state. Once the two search frontiers meet, then we have a middle node approximately half-way along an optimal solution path. Then, we can use divide and conquer frontier search to recursively generate a path from the initial state to the middle state, and from the middle state to the goal state. Frontier search can also be used to reduce the memory of heuristic best-first searches described in the following text.

22.3.7 Disk-Based Search Algorithms

Another approach to the memory limitation of graph-search algorithms is to store search nodes on magnetic disk instead of main memory. Currently, two terabyte disks can be bought for about $300, which is almost three orders of magnitude cheaper than semiconductor memory. The drawback of disk storage, however, is that the latency to access a single byte can be as much as ten milliseconds, which is about five orders of magnitude slower than semiconductor memory. The main reason to store generated nodes is to detect duplicate nodes, which is usually accomplished by a hash table in memory. A hash table cannot be directly implemented on magnetic disk, however, because of the long latencies for random access.

An alternative technique is called delayed duplicate detection. Rather than checking for duplicate nodes every time a node is generated, duplicate detection is delayed, for example, until the end of a breadth-first search iteration. At that point the queue of nodes on disk can be sorted by their state representation, using only sequential access algorithms, bringing duplicate nodes to adjacent locations. Then, a single sequential pass over the sorted queue can merge duplicate nodes in preparation for the next iteration of the search. This technique can also be combined with frontier search to further reduce the storage needed. See [27] for more details.

Another disk-based search algorithm is called structured duplicate detection [53].

22.3.8 Combinatorial Explosion

The problem with all brute-force search algorithms is that their time complexities often grow exponentially with problem size. This is called combinatorial explosion, and as a result, the size of problems that can be solved with these techniques is quite limited. For example, while the eight puzzle, with about 10^5 states, is easily solved by brute-force search, and the fifteen puzzle, with over 10^{13} states, can be searched exhaustively using brute-force frontier search, storing nodes on disk [26], the 5×5 twenty-four puzzle, with 10^{25} states, is completely beyond the reach of brute-force search.

22.4 Heuristic Search

In order to solve larger problems, domain-specific knowledge must be added to improve search efficiency. In AI, heuristic search has a general meaning, and a more specialized technical meaning. In a general sense, the term heuristic is used for any advice that is often effective, but is not guaranteed to work in every case. Within the heuristic search literature, however, the term heuristic usually refers to the special case of a heuristic evaluation function.

22.4.1 Heuristic Evaluation Functions

In a single-agent path-finding problem, a heuristic evaluation function estimates the cost of an optimal path between a pair of states. For a fixed goal state, a heuristic evaluation $h(n)$ is a function

of a node *n*, which estimates the distance from node *n* to the goal state. For example, the Euclidean or airline distance is an estimate of the highway distance between a pair of locations. A common heuristic function for the sliding-tile puzzles is called the Manhattan distance. It is computed by counting the number of moves along the grid that each tile is displaced from its goal position, and summing these values over all tiles.

The key properties of a heuristic evaluation function are that it estimate actual cost, and that it is efficiently computable. For example, the Euclidean distance between a pair of points can be computed in constant time. The Manhattan distance between a pair of states can be computed in time proportional to the number of tiles.

Most heuristic functions are derived by generating a simplified version of the problem to be solved, then using the cost of an optimal solution to the simplified problem as a heuristic evaluation function for the original problem. For example, if we remove the constraint that we must drive on the roads, the cost of an optimal solution to the resulting helicopter navigation problem is the Euclidean distance. In the sliding-tile puzzles, if we remove the constraint that a tile can only be slid into the blank position, then any tile can be moved to any adjacent position at any time. The optimal number of moves required to solve this simplified version of the problem is the Manhattan distance.

Since they are derived from relaxations of the original problem, such heuristics are also lower bounds on the costs of optimal solutions to the original problem, a property referred to as admissibility. For example, the Euclidean distance is a lower bound on road distance between two points, since the shortest path between a pair of points is a straight line. Similarly, the Manhattan distance is a lower bound on the actual number of moves necessary to solve an instance of a sliding-tile puzzle, since every tile must move at least as many times as its Manhattan distance to its goal position, and each move moves only one tile.

22.4.2 Pattern Database Heuristics

Pattern databases are heuristics that are precomputed and stored in lookup tables for use during search [1]. For example, consider a table with a separate entry for each possible configuration of a subset of the tiles in a sliding tile puzzle, called the pattern tiles. Each entry stores the minimum number of moves required to move the pattern tiles from the corresponding configuration to their goal configuration. Such a value is a lower bound on the number of moves needed to reach the goal from any state where the pattern tiles are in the given configuration, and can be used as an admissible heuristic during search.

To construct such a table, we perform a breadth-first search backward from the goal state, where a state is defined by the configuration of the pattern tiles. As each such configuration is reached for the first time, we store in the pattern database the number of moves needed to reach that configuration. This table is computed once before the search, and can be used for multiple searches that share the same goal state. The number of tiles in a pattern is limited by the amount of memory available to store an entry for each possible configuration of pattern tiles.

If when computing a pattern database we only count moves of the pattern tiles, then we can add the values from different pattern databases based on disjoint sets of tiles, without overestimating actual cost [24]. Thus, we can partition all the tiles in a problem into disjoint sets of pattern tiles, and sum the values from the different databases to get a more accurate but still admissible heuristic function.

A number of algorithms make use of heuristic evaluations, including pure heuristic search, the A* algorithm, iterative-deepening-A*, depth-first branch-and-bound, the heuristic path algorithm, recursive best-first search, and real-time-A*. In addition, heuristic evaluations can be employed in bidirectional search, and are used in two-player games as well.

22.4.3 Pure Heuristic Search

The simplest of these algorithms, pure heuristic search, expands nodes in order of their heuristic values $h(n)$ [9]. As a best-first search, it maintains a closed list of those nodes that have been expanded, and an open list of those nodes that have been generated but not yet expanded. The algorithm begins with just the initial node on the open list. At each step, a node on the open list with the minimum $h(n)$ value is expanded, generating all of its children, and is placed on the closed list. The heuristic function is applied to each of the children, and they are placed on the open list, sorted by their heuristic values. The algorithm continues until a goal node is chosen for expansion.

In a graph with cycles, multiple paths will be found to the same node, and the first path found may not be the shortest. When a shorter path is found to an open node, the shorter path is saved and the longer one discarded. When a shorter path to a closed node is found, the node is moved to open, and the shorter path is associated with it. The main drawback of pure heuristic search is that since it ignores the cost of the path so far to node n, it does not find optimal solutions.

22.4.4 A* Algorithm

The A* algorithm [14] combines features of uniform-cost search and pure heuristic search to efficiently compute optimal solutions. A* is a best-first search in which the cost associated with a node is $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the path from the initial state to node n, and $h(n)$ is the heuristic estimate of the cost of a path from node n to a goal. At each point a node with lowest f value is chosen for expansion. Ties among nodes of equal f value are broken in favor of nodes with lower h values. The algorithm terminates when a goal node is chosen for expansion.

A* finds an optimal path to a goal if the heuristic function $h(n)$ is admissible, meaning it never overestimates actual cost. For example, since airline distance never overestimates actual highway distance, and Manhattan distance never overestimates actual moves in the sliding-tile puzzles, A* using these evaluation functions will find optimal solutions to these problems. In addition, A* makes the most efficient use of a given heuristic function in the following sense: among all shortest-path algorithms using a given heuristic function $h(n)$, A* expands the fewest number of nodes [5].

The main drawback of A*, and indeed of any best-first search, is its memory requirement. Since the open and closed lists are stored in memory, A* is severely space-limited in practice, and will exhaust the available memory on current machines in minutes. For example, while it can be run successfully on the eight puzzle, it cannot solve most random instances of the fifteen puzzle before exhausting memory. Frontier-A* only stores the open list, which ameliorates the memory limitation, but does not eliminate it. An additional drawback of A* is that managing the open and closed lists is complex, and takes time.

22.4.5 Iterative-Deepening-A*

Just as depth-first iterative-deepening solved the space problem of breadth-first search, iterative-deepening-A* (IDA*) eliminates the memory constraint of A*, without sacrificing solution optimality [18]. Each iteration of the algorithm is a depth-first search that keeps track of the cost, $f(n) = g(n) + h(n)$, of each node generated. As soon as a node is generated whose cost exceeds a threshold for that iteration, it is deleted, and the search backtracks before continuing along a different path. The cost threshold is initialized to the heuristic estimate of the initial state, and is increased in each successive iteration to the lowest cost of all the nodes that were generated but not expanded during the previous iteration. The algorithm terminates when a goal state is reached whose cost does not exceed the current threshold.

Since IDA* performs a series of depth-first searches, its memory requirement is linear in the maximum search depth. In addition, if the heuristic function is admissible, IDA* finds an optimal solution. Finally, by an argument similar to that presented for depth-first iterative-deepening, IDA* expands the same number of nodes, asymptotically, as A* on a tree, provided that the number of nodes grows exponentially with solution cost. These facts, together with the optimality of A*, imply that IDA* is asymptotically optimal in time and space over all heuristic search algorithms that find optimal solutions on a tree. The additional benefits of IDA* are that it is much easier to implement, and often runs faster than A*, since it does not incur the overhead of managing the open and closed lists. Using appropriate admissible heuristic functions, IDA* can optimally solve random instances of the fifteen puzzle [18], the twenty-four puzzle [24], and Rubik's cube [22].

22.4.6 Depth-First Branch-and-Bound

For many problems, the maximum search depth is known in advance, or the search tree is finite. For example, the traveling salesman problem (TSP) is to visit each of a given set of cities and return to the starting city in a tour of shortest total distance. The most natural problem space for this problem consists of a tree where the root node represents the starting city, the nodes at level one represent all the cities that could be visited first, the nodes at level two represent all the cites that could be visited second, and so on. In this tree, the maximum depth is the number of cities, and all candidate solutions occur at this depth. In such a space, a simple depth-first search guarantees finding an optimal solution using space that is only linear in the depth.

The idea of depth-first branch-and-bound (DFBnB) is to make this search more efficient by keeping track of the best solution found so far. Since the cost of a partial tour is the sum of the costs of the edges traveled so far, whenever a partial tour is found whose cost equals or exceeds the cost of the best complete tour found so far, the branch representing the partial tour can be pruned, since all its descendents must have equal or greater cost. Whenever a lower-cost complete tour is found, the cost of the best tour is updated to this lower cost. In addition, an admissible heuristic function, such as the cost of the minimum spanning tree of the remaining unvisited cities, can be added to the cost so far of a partial tour to increase the amount of pruning. Finally, by ordering the children of a given node from smallest to largest estimated total cost, a low-cost solution can be found more quickly, further improving the pruning efficiency.

Interestingly, IDA* and DFBnB exhibit complementary behavior. Both are guaranteed to return an optimal solution using only linear space, assuming that their cost functions are admissible. In IDA*, the cost threshold is always a lower bound on the optimal solution cost, and increases in each iteration until it reaches the optimal cost. In DFBnB, the cost of the best solution found so far is always an upper bound on the optimal solution cost, and decreases until it reaches the optimal cost. While IDA* never expands any nodes whose cost exceeds the optimal cost, its overhead consists of expanding some nodes more than once. While DFBnB never expands any node more than once, its overhead consists of expanding some nodes whose cost exceed the optimal cost. For problems whose search trees are of bounded depth, or for which it is easy to construct a good solution, such as the TSP, DFBnB is usually the algorithm of choice for finding an optimal solution. For problems with infinite search trees, or for which it is difficult to construct a low-cost solution, such as the sliding-tile puzzles or Rubik's cube, IDA* is usually the best choice. Both IDA* and DFBnB suffer the same limitation of all linear-space search algorithms, however, which is that they can generate exponentially more nodes on a graph with many short cycles, relative to a best-first search.

22.4.7 Complexity of Finding Optimal Solutions

The time complexity of a heuristic search algorithm depends on the accuracy of the heuristic function. For example, if the heuristic evaluation function is an exact estimator, then A* runs in linear time,

expanding only those nodes on an optimal solution path. Conversely, with a heuristic that returns zero everywhere, A* becomes brute-force uniform-cost search, with exponential complexity. In general, the effect of a good heuristic function is to reduce the effective depth of search required [23].

22.4.8 Heuristic Path Algorithm

Since the complexity of finding optimal solutions to these problems is still exponential, even with a good heuristic function, in order to solve significantly larger problems, the optimality requirement must be relaxed. An early approach to this problem was the heuristic path algorithm (HPA) [35]. HPA is a best-first search, where the cost of a node n is computed as $f(n) = g(n) + w * h(n)$. Varying w produces a range of algorithms from uniform-cost search ($w = 0$), through A* ($w = 1$), to pure heuristic search ($w = \infty$). Increasing w beyond 1 generally decreases the amount of computation, while increasing the cost of the solution generated. This trade-off is often quite favorable, with small increases in solution cost yielding huge savings in computation [21]. Furthermore, it can be shown that the solutions found by this algorithm are guaranteed to be no more than a factor of w greater than optimal [2], but often are significantly better.

Breadth-first search, uniform-cost search, pure heuristic search, A*, and the heuristic path algorithm are all special cases of best-first search. In each step of a best-first search, the node that is best according to some cost function is chosen for expansion. These best-first algorithms differ only in their cost functions: the depth of node n for breadth-first search, $g(n)$ for uniform-cost search, $h(n)$ for pure heuristic search, $g(n) + h(n)$ for A*, and $g(n) + w * h(n)$ for the heuristic path algorithm.

22.4.9 Recursive Best-First Search

The memory limitation of the heuristic path algorithm can be overcome simply by replacing the best-first search with IDA* using the same weighted evaluation function. However, with $w \geq 1$, IDA* is no longer a best-first search, since the total cost of a child can be less than that of its parent, and thus nodes are not necessarily expanded in best-first order. An alternative algorithm is recursive best-first search (RBFS) [21]. RBFS simulates a best-first search in space that is linear in the maximum search depth, regardless of the cost function used. Even with an admissible cost function, RBFS generates fewer nodes than IDA*, and is generally superior to IDA*, except for a small increase in the cost per node generation.

It works by maintaining on the recursion stack the complete path to the current node being expanded, as well as all siblings of nodes on that path, along with the cost of the best node in the subtree explored below each sibling. Whenever the cost of the current node exceeds that of some other node in the previously expanded portion of the tree, the algorithm backs up to their closest common ancestor, and continues the search down the new path. In effect, the algorithm maintains a separate threshold for each subtree diverging from the current search path. See [21] for full details on RBFS.

22.5 Interleaving Search and Execution

In the discussion above, it is assumed that a complete solution can be computed, even before the first step of the solution is executed. This is in contrast to the situation in two-player games, discussed below, where because of computational limits and uncertainty due to the opponent's moves, search and execution are interleaved, with each search determining only the next move to be made. This paradigm is also applicable to single-agent problems. In the case of autonomous vehicle navigation, for example, information is limited by the horizon of the vehicle's sensors, and it must physically move to acquire more information. Thus, only a few moves can be computed at a time, and those

moves must be executed before computing the next. Below we consider algorithms designed for this scenario.

22.5.1 Minimin Search

Minimin search determines individual single-agent moves in constant time per move [19]. The algorithm searches forward from the current state to a fixed depth determined by the informational or computational resources available. At the search horizon, the A* evaluation function $f(n) = g(n) + h(n)$ is applied to the frontier nodes. Since all decisions are made by a single agent, the value of an interior node is the minimum of the frontier values in the subtree below that node. A single move is then made to a neighbor of the current state with the minimum value.

Most heuristic functions obey the triangle inequality characteristic of distance measures. As a result, $f(n) = g(n) + h(n)$ is guaranteed to be monotonically nondecreasing along a path. Furthermore, since minimin search has a fixed depth limit, we can apply depth-first branch-and-bound to prune the search tree. The performance improvement due to branch-and-bound is quite dramatic, in some cases extending the achievable search horizon by a factor of 5 relative to brute-force minimin search on sliding-tile puzzles [19].

Minimin search with branch-and-bound is an algorithm for evaluating the immediate neighbors of the current node. As such, it is run until a best child is identified, at which point the chosen move is executed in the real world. We can view the static evaluation function combined with lookahead search as simply a more accurate, but computationally more expensive, heuristic function. In fact, it provides an entire spectrum of heuristic functions trading off accuracy for cost, depending on the search horizon.

22.5.2 Real-Time-A*

Simply repeating minimin search for each move ignores information from previous searches and results in infinite loops. In addition, since actions are committed based on limited information, often the best move may be to undo the previous move. The principle of rationality is that backtracking should occur when the estimated cost of continuing the current path exceeds the cost of going back to a previous state, plus the estimated cost of reaching the goal from that state. Real-time-A* (RTA*) implements this policy in constant time per move on a tree [19].

For each move, the $f(n) = g(n) + h(n)$ value of each neighbor of the current state is computed, where $g(n)$ is now the cost of the edge from the current state to the neighbor, instead of from the initial state. The problem solver moves to a neighbor with the minimum $f(n)$ value, and stores in the previous state the best $f(n)$ value among the remaining neighbors. This represents the $h(n)$ value of the previous state from the perspective of the new current state. This is repeated until a goal is reached. To determine the $h(n)$ value of a previously visited state, the stored value is used, while for a new state the heuristic evaluator is called. Note that the heuristic evaluator may employ minimin lookahead search with branch-and-bound as well.

In a finite problem space in which there exists a path to a goal from every state, RTA* is guaranteed to find a solution, regardless of the heuristic evaluation function [19]. Furthermore, on a tree, RTA* makes locally optimal decisions based on the information available at the time of the decision.

22.5.3 Learning-Real-Time-A*

If a problem is to be solved repeatedly with the same goal state but different initial states, we would like our algorithm to improve its performance over time. Learning-real-time-A* (LRTA*) is such an algorithm. It behaves almost identically to RTA*, except that instead of storing the second-best f

value of a node as its new heuristic value, it stores the best value instead. Once one problem instance is solved, the stored heuristic values are saved and become the initial values for the next problem instance. While LRTA* is less efficient than RTA* for solving a single problem instance, if it starts with admissible initial heuristic values, over repeated trials its heuristic values eventually converge to their exact values, at which point the algorithm returns optimal solutions.

22.6 Game Playing

The second major application of heuristic search algorithms in AI is game playing. One of the original challenges of AI, which in fact predates the term artificial intelligence, was to build a program that could play chess at the level of the best human players [48], a goal finally achieved in 1997 [18]. The earliest work in this area focussed on two-player perfect information games, such as chess and checkers, but recently the field has broadened considerably.

22.6.1 Minimax Search

The standard algorithm for two-player perfect-information games is minimax search with heuristic static evaluation [44]. The simplest version of the algorithm searches forward to a fixed depth in the game tree, limited by the amount of time available per move. At this search horizon, a heuristic evaluation function is applied to the frontier nodes. In this case, the heuristic is a function that takes a board position and returns a number that indicates how favorable that position is for one player relative to the other. For example, a very simple heuristic evaluator for chess would count the total number of pieces on the board for one player, weighted by their relative strength, and subtract the weighted sum of the opponent's pieces. Thus, large positive values would correspond to strong positions for one player, called MAX, whereas negative values of large magnitude would represent advantageous situations for the opponent, called MIN.

Given the heuristic evaluations of the frontier nodes, values for the interior nodes in the tree are recursively computed according to the minimax rule. The value of a node where it is MAX's turn to move is the maximum of the values of its children, while the value of a node where MIN is to move is the minimum of the values of its children. Thus, at alternate levels of the tree, the minimum or the maximum values of the children are backed up. This continues until the values of the immediate children of the current position are computed, at which point one move is made to the child with the maximum or minimum value, depending on whose turn it is to move.

22.6.2 Alpha-Beta Pruning

One of the most elegant of all AI search algorithms is alpha-beta pruning. It was invented in the late 1950s, and a thorough treatment of the algorithm can be found in [28]. The idea, similar to branch-and-bound, is that the minimax value of the root of a game tree can be determined without examining all the nodes at the search frontier.

Figure 22.5 shows some examples of alpha-beta pruning. Only the pictured nodes are generated by the algorithm, with the heavy black lines indicating pruning. At the square nodes MAX is to move, while at the circular nodes it is MIN's turn. The search proceeds depth-first to minimize the memory required, and only evaluates a frontier node when necessary. First, nodes *e* and *f* are statically evaluated at 4 and 5, respectively, and their minimum value, 4, is backed up to their parent node *d*. Node *h* is then evaluated at 3, and hence the value of its parent node *g* must be less than or equal to 3, since it is the minimum of 3 and the unknown value of its right child. The value of node *c* must be 4 then, because it is the maximum of 4 and a value that is less than or equal to 3. Since

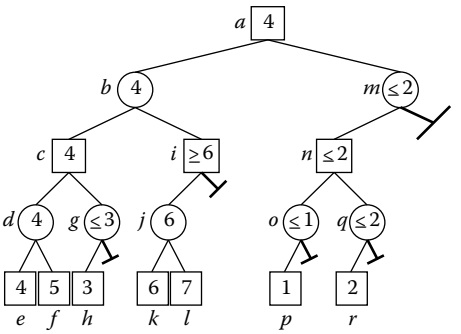

FIGURE 22.5 Alpha-beta pruning example.

we have determined the minimax value of node *c*, we do not need to evaluate or even generate any more children of node *g*.

Similarly, after statically evaluating nodes *k* and *l* at 6 and 7, respectively, the backed up value of their parent node *j* is 6, the minimum of these values. This tells us that the minimax value of node *i* must be greater than or equal to 6, since it is the maximum of 6 and the unknown value of its right child. Since the value of node *b* is the minimum of 4 and a value that is greater than or equal to 6, it must be 4, and hence we prune the remaining children of node i.

The right half of the tree shows an example of deep pruning. After evaluating the left half of the tree, we know that the value of the root node *a* is greater than or equal to 4, the minimax value of node *b*. Once node *p* is evaluated at 1, the value of its parent node *o* must be less than or equal to 1. Since the value of the root is greater than or equal to 4, the value of node *o* cannot propagate to the root, and hence we need not generate any more children of node *o*. A similar situation exists after the evaluation of node *r* as 2. At that point, the value of node *o* is less than or equal to 1, and the value of node *q* is less than or equal to 2; hence, the value of node *n*, which is the maximum of the values of nodes *o* and *q*, must be less than or equal to 2. Furthermore, since the value of node *m* is the minimum of the value of node *n* and that of its brothers, and node *n* has a value less than or equal to 2, the value of node *m* must also be less than or equal to 2. This causes the remaining children of node *m* to be pruned, since the value of the root node *a* is greater than or equal to 4. Thus, we compute the minimax value of the root of the tree to be 4 by generating only seven leaf nodes in this case.

Since alpha-beta pruning performs a minimax search while pruning much of the tree, its effect is to allow a deeper search in the same amount of time. This raises the question of how much does alpha-beta improve performance? The efficiency of alpha-beta pruning depends upon the order in which nodes are encountered at the search frontier. For any set of frontier node values, there exists some ordering of the values such that alpha-beta will not perform any cutoffs at all. In that case, all frontier nodes must be evaluated, and the time complexity is $O(b^d)$.

On the other hand, there is an optimal or perfect ordering in which every possible cutoff is realized. In that case, the asymptotic time complexity is reduced from $O(b^d)$ to $O(b^{d/2})$. Another way of viewing the perfect ordering case is that for the same amount of computation, one can search twice as deep with alpha-beta pruning as without. Since the search tree grows exponentially with depth, doubling the search horizon is a dramatic improvement.

In between worst-possible ordering and perfect ordering is random ordering, which is the average case. Under random ordering of the frontier nodes, alpha-beta pruning reduces the asymptotic time complexity to approximately $O(b^{3d/4})$ [33]. This means that one can search 4/3 times as deep with alpha-beta than with simple minimax, yielding a 33% improvement in search depth.

In practice, however, the time complexity of alpha-beta is closer to the best case of $O(b^{d/2})$ due to node ordering. The idea of node ordering is that instead of generating the children of a node in an

arbitrary order, we can order the tree based on static evaluations of the interior nodes. In particular, the children of MAX nodes are expanded in decreasing order of their static values, while the children of MIN nodes are expanded in increasing order of their static values.

22.6.3 Other Techniques

A wide variety of additional techniques are employed in modern game-playing programs, including quiescent search, iterative deepening, transposition tables, opening books, end-game databases, and special-purpose hardware. We consider each of these in turn.

22.6.3.1 Quiescence

The idea of quiescence is that the static evaluator should not be applied to positions whose values are unstable, such as those occurring in the middle of a piece trade. In those positions, a small secondary search is conducted until the static evaluation becomes more stable. In games such as chess or checkers, this can be achieved by not statically evaluating any position that allows capture moves, but exploring capture moves one level deeper.

22.6.3.2 Iterative Deepening

Iterative deepening is used to solve the problem of where to set the search horizon [45], and in fact predated its use as a memory-saving device in single-agent search. In a tournament game, there is a limited amount of time allowed for moves. Unfortunately, it is very difficult to accurately predict how long it will take to perform an alpha-beta search to a given depth. The solution is to perform a series of searches to successively greater depths. When time runs out, the move recommended by the last completed search is made.

22.6.3.3 Transposition Tables

The search graphs of most games contain multiple paths to the same node, often reached by making the same moves in a different order, referred to as a transposition of the moves. Since alpha-beta is a depth-first search, it does not detect such duplicate nodes. A transposition table is a table of previously encountered game states, together with their backed-up minimax values. Whenever a new state is generated, if it is stored in the transposition table, its stored value is used instead of researching the tree below the node. Transposition tables can also be used in single-agent depth-first searches to detect some duplicate nodes.

22.6.3.4 Opening Books

Just like people, programs often learn and store the best first few moves of a game, since many games start from a common initial state. Such an opening book can be hand-coded by a human game expert, or automatically learned from deep off-line searches.

22.6.3.5 End-Game Databases

In many games, such as checkers, the number of positions near the end of the game is small relative to the mid-game. This allows the precomputation and storage of end-game databases, which contain the true value, such as win, lose, or draw, of a large number of positions near the end of the game. For example, the Chinook checkers program stores on disk a database of all positions with 10 or fewer pieces on the board, along with their exact values [42]. When a position in the database is encountered during search, the stored value is used, rather than searching the position any further.

22.6.3.6 Special-Purpose Hardware

While the basic algorithms are described above, much of the performance advances in computer chess in previous decades came from faster hardware. The faster the machine, the deeper it can search in the time available, and the better it plays. Despite the rapidly advancing speed of general-purpose computers, the best machines in the 1980s and 1990s included special-purpose hardware designed and built only to play chess. For example, IBM's DeepBlue could evaluate about 200 million chess positions per second [17].

22.6.4 Significant Milestones

The most significant milestones in this area include defeating the human world champion at chess, and solving the game of checkers.

In May 1997, IBM's DeepBlue defeated Gary Kasparov, the world champion, in a six-game exhibition match, achieving a long-anticipated goal in artificial intelligence [18]. Currently, the best chess machines are general-purpose computers that rely entirely on software for their performance, and play comparably to the best humans.

In April 2007, Jonathan Schaeffer's Chinook group at the University of Alberta announced that they had solved the game of checkers [43], proving that when both sides play perfectly, checkers is a draw. This is by far the most complex game solved to date, using two dozen computers running for about 18 years. The combination of its opening book, its mid-game search, and 10-piece end-game databases allows Chinook to play perfect checkers.

22.6.5 Multiplayer Games, Imperfect and Hidden Information

Minimax search with static evaluation and alpha-beta pruning is most appropriate for two-player games with perfect information and alternating moves among the players. This paradigm extends in a straightforward way to more than two players, but alpha-beta becomes much less effective [20].

Games with chance elements, such as the roll of the dice in backgammon, for example, tend to limit search algorithms because of the need to search over all possible chance outcomes. In addition to chance, card games have information that is available to some players but hidden from others, such as cards in different hands in Bridge. Poker is a very difficult challenge in this area, combining all of the above complexities, as well as active deception and the need to model the opponents.

One technique that has been effective in handling hidden information is Monte-Carlo sampling [12]. Given a decision to be made, such as the play of a card in bridge, we can randomly generate a set of hands that is consistent with the information known about those hands at the current point in time. Given this particular set of hands, we then use perfect-information techniques, such as alpha-beta minimax, to determine the optimal play in this case. We then repeat the experiment, generating another random set of hands, consistent with the information available, and compute the optimal move in that case. After generating about 100 different random hands, we then play the card that was most often the optimal card to play over all the randomly generated hands.

22.7 Constraint-Satisfaction Problems

In addition to single-agent path-finding problems and game playing, the third major application of heuristic search is constraint-satisfaction problems. The N-queens problem and the popular Sudoku puzzle are classic examples. Other examples include graph coloring, boolean satisfiability, and scheduling problems.

Constraint-satisfaction problems are usually modelled as follows: There is a set of variables, a set of values for each variable, and a set of constraints on the values that the variables can be assigned.

A unary constraint on a variable specifies a subset of all possible values that can be assigned to that variable. A binary constraint between two variables specifies which possible combinations of assignments to the pair of variables is legal. For example, in the N-queens problem, the variables would represent individual queens, and the values would be their positions on the board. The constraints are binary constraints on each pair of queens that prohibit them from occupying positions on the same row, column, or diagonal.

22.7.1 Chronological Backtracking

The brute-force approach to constraint satisfaction is called chronological backtracking. One selects an order for the variables, and an order for the values, and starts assigning values to the variables one at a time. Each assignment is made so that all constraints involving any of the instantiated variables are satisfied. The reason for this is that once a constraint is violated, no assignment to the remaining variables can possibly satisfy that constraint. Once a variable is reached which has no remaining legal assignments, then the last variable that was assigned is reassigned to its next legal value. The algorithm continues until either one or all complete, consistent assignments are found, resulting in success, or all possible assignments are shown to violate some constraint, resulting in failure. Figure 22.6 shows the tree generated by chronological backtracking to find all solutions to the four-queens problem. The tree is searched depth-first to minimize memory requirements.

22.7.2 Intelligent Backtracking

One can improve the performance of chronological backtracking using a number of techniques, such as variable ordering, value ordering, backjumping, and forward checking.

The order in which variables are instantiated can have a large effect on the size of a search tree. The idea of variable ordering is to order the variables from most constrained to least constrained [10,37].

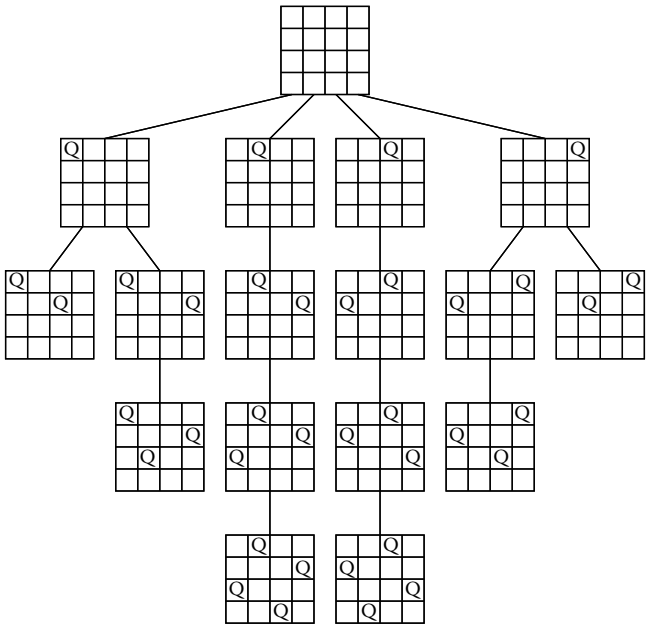

FIGURE 22.6 Tree generated to solve the four queens problem.

For example, if any variable has only a single value remaining that is consistent with the previously instantiated variables, it should be assigned that value immediately. In general, the variables should be instantiated in increasing order of the size of their remaining domains. This can either be done statically at the beginning of the search, or dynamically, reordering the remaining variables each time a variable is assigned a new value.

The order in which the values of a given variable are chosen determines the order in which the tree is searched. Since it does not effect the size of the tree, it makes no difference if all solutions are to be found. If only a single solution is required, however, value ordering can decrease the time required to find a solution. Thus, one should order the values from least constraining to most constraining, in order to minimize the time required to find a first solution [6,13].

The idea of backjumping is that when an impasse is reached, instead of simply undoing the last decision made, the decision that actually caused the failure should be modified [11]. For example, consider a three-variable problem where the variables are instantiated in the order x, y, z. Assume that values have been chosen for both x and y, but that all possible values for z conflict with the value chosen for x. In chronological backtracking, the value chosen for y would be changed, and then all the possible values for z would be tested again, to no avail. A better strategy in this case is to go back to the source of the failure, and change the value of x, before trying different values for y and z.

When a variable is assigned a value, forward checking checks each remaining uninstantiated variable to make sure that there is at least one assignment for each of them that is consistent with the current assignments. If not, the original variable is reassigned to its next value. Forward checking subsumes backjumping.

22.7.3 Constraint Recording

In a constraint-satisfaction problem, some constraints are explicitly specified, and others are implied by explicit constraints. Implicit constraints may be discovered either during a backtracking search, or in advance in a preprocessing phase. The idea of constraint recording is that once these implicit constraints are discovered, they should be saved explicitly so that they do not have to be rediscovered.

A simple example of constraint recording in a preprocessing phase is called arc consistency [10, 29,31]. For each pair of variables x and y that are related by a binary constraint, we remove from the domain of x any values that do not have at least one corresponding consistent assignment to y, and vice versa. In general, several iterations may be required to achieve complete arc consistency. Path consistency is a generalization of arc consistency where instead of considering pairs of variables, we examine triples of constrained variables. The effect of performing arc or path consistency before backtracking is that the resulting search space can be dramatically reduced. In some cases, this preprocessing of the constraints can eliminate the need for search entirely.

22.7.4 Local Search Algorithms

Backtracking searches a space of consistent partial assignments to variables, in the sense that all constraints among instantiated variables are satisfied, and looks for a complete consistent assignment to the variables, or in other words a solution. An alternative approach is to search a space of inconsistent but complete assignments to the variables, until a consistent complete assignment is found. This approach is known as heuristic repair [30] or more generally stochastic local search. For example, in the N-queens problem, this algorithm places all N queens on the board at the same time, and then moves queens one at a time until a solution is found. The natural heuristic, called min-conflicts, moves a queen that is in conflict with the most other queens, to a position where it conflicts with the fewest other queens. What is surprising about this simple strategy is how well it performs, relative to backtracking. While backtracking techniques can solve on the order of

hundred-queen problems, heuristic repair can solve million-queen problems, often with only about 50 individual queen moves!

This strategy has been extensively explored in the context of boolean satisfiability. The problem of boolean satisfiability starts with a formula in propositional logic in conjunctive normal form. This is an AND of a set of clauses, each of which is an OR of a set of literals, each of which is either an atomic proposition or a negated atomic proposition. For example, the boolean formula $(a + b) \cdot (a' + b')$ is in conjunctive normal form if $+$ represents OR, $\cdot$ represents AND, and $'$ represents negation. The problem is to determine if there exists some assignment to the variables such that the entire formula evaluates to true, or in other words, all the clauses are satisfied. For example, the above formula is satisfiable, since assigning a to true and b to false will satisfy both clauses. If each clause has three literals, called 3-SAT, the problem is NP-complete.

The greedy satisfiability (GSAT) algorithm [40] starts with a random assignment of boolean values to the variables of a satisfiability problem, and flips the assignment of a variable that results in the largest net increase in the number of satisfied clauses. It continues until either a solution is found, or until a specified number of flips are performed, at which point it starts over with a different random initial assignment. WalkSAT [41] adds another random element to this algorithm. With a certain probability, it flips a variable that GSAT would flip, and with one minus that probability, it chooses a random unsatisfied clause, and flips a randomly chosen variable in that clause.

The main drawback of these local-search approaches is that they are not complete, in that they are not guaranteed to find a solution in a finite amount of time, even if one exists. If there is no solution, these algorithms will run forever, whereas complete backtracking algorithms will eventually discover that a problem is not solvable.

The best complete algorithms for boolean satisfiability are based on the DPLL algorithm [3]. These algorithms assign values to variables one at a time, and propagate the consequences of assignments using unit propagation, backtracking when a partial assignment cannot be extended any further. Both complete algorithms and stochastic local search are areas of active research for boolean satisfiability. Currently, local search algorithms seem to perform best on difficult random problems that are satisfiable, while the complete algorithms tend to perform better on more structured problems.

While constraint-satisfaction problems appear somewhat different from single-agent path-finding problems and two-player games, there is a strong similarity among the algorithms employed. For example, backtracking can be viewed as a form of branch-and-bound, where a node is pruned when any constraint is violated. Similarly, heuristic repair can be viewed as a heuristic search where the evaluation function is the total number of constraints that are violated, and the goal is to find a state with zero constraint violations.

22.8 Research Issues and Summary

22.8.1 Research Issues

A primary research problem in this area is the development of faster algorithms. All the above algorithms are limited by efficiency either in the size of problems that they can solve optimally, or in the quality of the decisions they can make, or solutions they can find within practical time limits. Thus, there is a continual demand for faster algorithms.

One approach to faster algorithms is parallel search. Most search algorithms have a tremendous amount of potential parallelism, since the basic step of node generation and evaluation is often performed billions or trillions of times. As a result, many such algorithms are readily parallelized with nearly linear speedups. The algorithms that are difficult to parallelize are branch-and-bound algorithms, such as alpha-beta pruning, because the results of searching one part of the tree determine whether another part of the tree needs to be examined at all.

Since the performance of a search algorithm depends critically on the quality of the heuristic evaluation function, another important research area is the automatic generation of such functions. This was pioneered in the area of two-player games by Arthur Samuel's landmark checkers program that learned to improve its evaluation function through repeated play [39]. The development of pattern databases, described above, is another approach to automatic heuristic construction.

While the first games to be addressed with AI techniques were two-player perfect information games such as chess and checkers, the field has branched out to consider games with random elements, such as backgammon, hidden information, such as card games, and multiple players. Recently the game of poker has seen a great deal of activity, both from search-based and game-theoretic approaches. The game of Go continues to be very challenging for computers, both because of its large branching factor, and the difficulty of evaluating nonterminal positions.

Another important area in computer game-playing is developing automated opponents for video games. The electronic entertainment industry is larger than the motion-picture industry, and most of these games have agents that must be computer controlled. In addition to low-level issues such as path planning for multiple agents, these games require high-level strategy as well.

22.8.2 Summary

We have described search algorithms for three different classes of problems. In single-agent path-finding problems, the task is to find a sequence of operators that map an initial state to a desired goal state. Much of the work in this area has focussed on finding optimal solutions to such problems, often making use of admissible heuristic functions to speed up the search without sacrificing solution optimality. In the area of game playing, finding optimal solutions is infeasible, and research has focussed on algorithms for making the best move decisions possible given a limited amount of computing time. This approach has also been applied to single-agent problems as well. In constraint-satisfaction problems, the task is to find a state that satisfies a set of constraints. While all three of these types of problems are different, the same set of ideas, such as brute-force search and heuristic evaluation functions, can be applied to all three.

22.9 Further Information

The classic reference in this area is [34]. A number of papers were collected in an edited volume devoted to search [16]. The constraint-satisfaction area is covered by [4], and stochastic local search is covered in [15]. Most new research in this area initially appears in the Proceedings of the Association for the Advancement of on Artificial Intelligence (AAAI) or the Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI). The Preeminent journals in this area include *Artificial Intelligence* (AIJ), and the *Journal of Artificial Intelligence Research* (JAIR).

Defining Terms

Admissible: A heuristic is said to be admissible if it never overestimates actual distance from a given state to a goal. An algorithm is said to be admissible if it always finds an optimal solution to a problem if one exists.

Branching factor: The average number of children of a node in a problem-space graph.

Constraint-satisfaction problem: A problem where the task is to identify a state that satisfies a set of constraints.

Depth: The length of a shortest path from the initial state to a goal state.

Heuristic evaluation function: A function from a state to a number. In a single-agent problem, it estimates the distance from the state to a goal. In a game, it estimates the merit of the position with respect to one player.

Node expansion: Generating all the children of a given state.

Node generation: Creating the data structure that corresponds to a problem state.

Operator: An action that maps one state into another state, such as a twist of Rubik's cube.

Problem instance: A problem space together with an initial state of the problem and a desired set of goal states.

Problem space: A theoretical construct in which a search takes place, consisting of a set of states and a set of operators.

Problem-space graph: A graphical representation of a problem space, where states are represented by nodes, and operators are represented by edges.

Search: A trial-and-error exploration of alternative solutions to a problem.

Search tree: A problem-space graph with a unique path to each node.

Single-agent path-finding problem: A problem where the task is to find a sequence of operators that map an initial state to a goal state.

State: A configuration of a problem, such as the arrangement of the parts of a Rubik's cube at a given point in time.

Acknowledgment

This work was supported in part by NSF Grant IIS-0713178.

References

1. Culberson, J. and J. Schaeffer, Pattern databases, *Computational Intelligence*, 14(3), 318–334, 1998.
2. Davis, H.W., A. Bramanti-Gregor, and J. Wang, The advantages of using depth and breadth components in heuristic search, in *Methodologies for Intelligent Systems 3*, Z.W. Ras and L. Saitta (Eds.), North-Holland, Amsterdam, the Netherlands, 1989, pp. 19–28.
3. Davis, M., G. Logemann, and D. Loveland, A machine program for theorem proving, *Journal of the Association for Computing Machinery*, 5(7), 394–397, 1962.
4. Dechter, R., *Constraint Processing*, Morgan-Kaufmann, San Francisco, CA, 2003.
5. Dechter, R. and J. Pearl, Generalized best-first search strategies and the optimality of A*, *Journal of the Association for Computing Machinery*, 32(3), 505–536, July 1985.
6. Dechter, R. and J. Pearl, Network-based heuristics for constraint-satisfaction problems, *Artificial Intelligence*, 34(1), 1–38, 1987.
7. Dijkstra, E.W., A note on two problems in connexion with graphs, *Numerische Mathematik*, 1, 269–271, 1959.
8. Dillenburg, J.F. and P.C. Nelson, Improving the efficiency of depth-first search by cycle elimination, *Information Processing Letters*, 45(1), 5–10, 1993.
9. Doran, J.E. and D. Michie, Experiments with the graph traverser program, *Proceedings of the Royal Society A*, 294, 235–259, 1966.
10. Freuder, E.C., A sufficient condition for backtrack-free search, *Journal of the Association for Computing Machinery*, 29(1), 24–32, 1982.

11. Gaschnig, J., Performance measurement and analysis of certain search algorithms, PhD thesis. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1979.
12. Ginsberg, M.L., GIB: Imperfect information in a computationally challenging game, *Journal of Artificial Intelligence Research*, 14, 303–358, 2001.
13. Haralick, R.M. and G.L. Elliott, Increasing tree search efficiency for constraint satisfaction problems, *Artificial Intelligence*, 14, 263–313, 1980.
14. Hart, P.E., N.J. Nilsson, and B. Raphael, A formal basis for the heuristic determination of minimum cost paths, *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 100–107, 1968.
15. Hoos, H.H. and T. Stutzle, *Stochastic Local Search: Foundations and Applications*, Morgan-Kaufmann, San Francisco, CA, 2004.
16. Kanal, L. and V. Kumar (Eds.), *Search in Artificial Intelligence*, Springer-Verlag, New York, 1988.
17. Campbell, M., A.J. Hoane, and F. Hsu, Deep blue, *Artificial Intelligence*, 134(1–2), 57–83, 2002.
18. Korf, R.E., Depth-first iterative-deepening: An optimal admissible tree search, *Artificial Intelligence*, 27(1), 97–109, 1985.
19. Korf, R.E., Real-time heuristic search, *Artificial Intelligence*, 42(2–3), 189–211, 1990.
20. Korf, R.E., Multi-player alpha-beta pruning, *Artificial Intelligence*, 48(1), 99–111, 1991.
21. Korf, R.E., Linear-space best-first search, *Artificial Intelligence*, 62(1), 41–78, 1993.
22. Korf, R.E., Finding optimal solutions to Rubik's cube using pattern databases, *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, Providence, RI, July 1997, pp. 700–705.
23. Korf, R.E., M. Reid, and S. Edelkamp, Time complexity of Iterative-Deepening-A*, *Artificial Intelligence*, 129(1–2), 199–218, 2001.
24. Korf, R.E. and A. Felner, Disjoint pattern database heuristics, *Artificial Intelligence*, 134(1–2), 9–22, 2002.
25. Korf, R.E, W. Zhang, I. Thayer, and H. Hohwald, Frontier search, *Journal of the Association for Computing Machinery*, 52(5), 715–748, 2005.
26. Korf, R.E. and P. Schultze, Large-scale, parallel breadth-first search, *Proceedings of the National Conference on Artificial Intelligence (AAAI-05)*, Pittsburgh, PA, July 2005, pp. 1380–1385.
27. Korf, R.E., Linear-time disk-based implicit graph search, *Journal of the ACM*, 55(6), 26-1–26-40, 2008.
28. Knuth, D.E. and R.E. Moore, An analysis of alpha-beta pruning, *Artificial Intelligence*, 6(4), 293–326, 1975.
29. Mackworth, A.K., Consistency in networks of relations. *Artificial Intelligence*, 8(1), 99–118, 1977.
30. Minton, S., M.D. Johnston, A.B. Philips, and P. Laird, Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems, *Artificial Intelligence*, 58(1–3), 161–205, 1992.
31. Montanari, U., Networks of constraints: Fundamental properties and applications to picture processing, *Information Science*, 7, 95–132, 1974.
32. Newell, A. and H.A. Simon, *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, NJ, 1972.
33. Pearl, J., The solution for the branching factor of the alpha-beta pruning algorithm and its optimality, *Communications of the Association of Computing Machinery*, 25(8), 559–564, 1982.
34. Pearl, J., *Heuristics*, Addison-Wesley, Reading, MA, 1984.
35. Pohl, I., Heuristic search viewed as path finding in a graph, *Artificial Intelligence*, 1, 193–204, 1970.
36. Pohl, I., Bi-directional search, in *Machine Intelligence 6*, B. Meltzer and D. Michie (Eds.), American Elsevier, New York, 1971, pp. 127–140.
37. Purdom, P.W., Search rearrangement backtracking and polynomial average time, *Artificial Intelligence*, 21(1–2), 117–133, 1983.
38. Ratner, D. and M. Warmuth, Finding a shortest solution for the NxN extension of the 15-puzzle is intractable, *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, Philadelphia, PA, 1986, pp. 168–172.

39. Samuel, A.L., Some studies in machine learning using the game of checkers, in *Computers and Thought*, E. Feigenbaum and J. Feldman (Eds.), McGraw-Hill, New York, 1963, pp. 71–105.
40. Selman, B., H. Levesque, and D. Mitchell, A new method for solving hard satisfiability problems, *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, San Jose, CA, July 1992, pp. 440–446.
41. Selman, B., H. Kautz, and B. Cohen, Noise strategies for improving local search, *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, Seattle, WA, August 1994. pp. 337–343.
42. Schaeffer, J., Y. Bjornsson, N. Burch, R. Lake, P. Lu, and S. Sutphen, Building the checkers 10-piece endgame databases, in H.J. van den Herik, Iida, H., and Heinz, E.A., editors, *Advances in Computer Games 10*, Kluwer, Boston, MA, 2003, pp. 193–210.
43. Schaeffer, J., N. Burch, Y. Bjornsson, A. Kishimoto, M. Mueller, R. Lake, P. Lu, and S. Sutphen, Checkers is solved, *Science*, 317, 1518–1522, 2007.
44. Shannon, C.E., Programming a computer for playing chess, *Philosophical Magazine*, 41, 256–275, 1950.
45. Slate, D.J. and L.R. Atkin, CHESS 4.5 - The Northwestern University chess program, in *Chess Skill in Man and Machine*, P.W. Frey (Ed.), Springer-Verlag, New York, 1977, pp. 82–118.
46. Stickel, M.E. and W.M. Tyson, An analysis of consecutively bounded depth-first search with applications in automated deduction, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-85)*, Los Angeles, CA, August 1985, pp. 1073–1075.
47. Taylor, L. and R.E. Korf, Pruning duplicate nodes in depth-first search, *Proceedings of the National Conference on Artificial Intelligence (AAAI-93)*, Washington, DC, July 1993, pp. 756–761.
48. Turing, A.M., Computing machinery and intelligence, *Mind*, 59, 433–460, 1950. Also in *Computers and Thought*, E. Feigenbaum and J. Feldman (Eds.), McGraw-Hill, New York, 1963.
49. Zhou, R. and E.A. Hansen, Sparse-memory graph search, *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, Acapulco, Mexico, August 2003, pp. 1259–1266.
50. Zhou, R. and E.A. Hansen, Structured duplicate detection in external-memory graph search, *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04)*, San Jose, CA, July 2004, pp. 683–688.