Artificial Intelligence Search Algorithms

Richard E. Korf*
Computer Science Department
University of California, Los Angeles
Los Angeles, Ca. 90095

June 23, 1998

1 Introduction

Search is a universal problem-solving mechanism in artificial intelligence (AI). In AI problems, the sequence of steps required for solution of a problem are not known a priori, but often must be determined by a systematic 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**, **two-player games**, and **constraint-satisfaction problems**.

Classic examples in the AI literature of pathfinding problems are the sliding-tile puzzles, including the 3×3 Eight Puzzle (see Fig. 1) and its larger relatives the 4×4 Fifteen 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[43]. Other well-known examples of single-agent pathfinding problems include Rubik's Cube and theorem proving. Real-world examples include the travelling salesman problem, vehicle navigation, and the wiring of VLSI circuits. In each case, the task is to find a sequence of operations that map an initial state to a goal state.

A second class of search problems include two-player perfect-information games, such as chess, checkers, and Othello. The third category is constraint-satisfaction problems, such as the Eight Queens Problem. The task is to place eight queens on an 8×8 chessboard, such that no two queens are on the same row, column or diagonal. Real-world examples of constraint-satisfaction problems are ubiquitous, including 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, and bidirectional search. Next, various heuristic searches are examined 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 two-player game searches, including minimax and alpha-beta pruning. Finally, we examine constraint-satisfaction algorithms, such as backtracking,

*This work was supported in part by NSF Grant IRI-9619447, and a grant from Rockwell International.

constraint recording, and heuristic repair. The efficiency 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.

2 Problem Space Model

A *problem space* is the environment in which a search takes place [34]. 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 Eight Queens Problem, the goal state is not given explicitly, but rather implicitly specified by certain properties that must be satisfied by a 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 1 shows a small part of the Eight Puzzle problem-space graph.

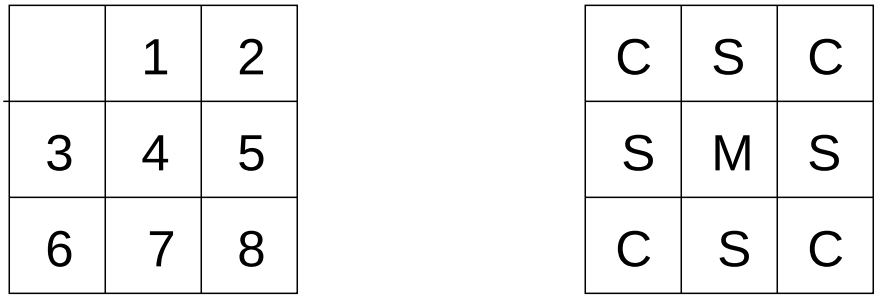

Figure 1: Eight Puzzle Search Tree Fragment

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, increasing the size of the tree. The benefit of a tree is that the absence of cycles greatly simplifies many search algorithms. In this survey, we will restrict our attention to trees, but there exist graph versions of most of the algorithms we describe as well.

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. If the goal were in the bottom row of figure 1, the depth of the problem instance represented by the initial state at the root would be three moves.

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, and bidirectional search. In the descriptions of the algorithms below, 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.

3.1 Breadth-First Search

Breadth-first search expands nodes in order of their distance from the root, generating one level of the tree at a time until a solution is found (see Figure 2). It is most easily implemented by maintaining a 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 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 saved 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.

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 also known as Dijkstra's single-source shortest-path algorithm[6].

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 $O(b^{c/m})$, where c is the cost of an optimal solution, and m is the minimum edge cost. Unfortunately, it also suffers the same memory limitation as breadth-first search.

3.3 Depth-First Search

Depth-first search remedies the space limitation of breadth-first search by always generating next a child of the deepest unexpanded node (see Figure 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.

The advantage of depth-first search is that its space requirement is only linear with respect to the 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 disadvantage of depth-first search is that it may not terminate on an infinite tree, but simply go down the left-most path forever. Even a finite graph can generate an infinite tree. 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.

3.4 Depth-First Iterative-Deepening

Depth-first iterative-deepening (DFID) combines the best features of breadth-first and depth-first search[48, 18]. 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 4).

Since it never generates a node until all shallower nodes have been generated, the first solution found by DFID is guaranteed to be along 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 it appears that DFID wastes a great deal of time in the iterations prior to 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}, 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 become 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 on the previous iteration whose cost exceeded the previous threshold.

On a graph with cycles, however, breadth-first search may be much more efficient than any

depth-first search. The reason is that a breadth-first search can check for duplicate nodes whereas a depth-first search cannot. Thus, the complexity of breadth-first search grows only as the number of nodes 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 [49] and [7].

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[40]. 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.

3.6 Combinatorial Explosion

The problem with all brute-force search algorithms is that their time complexities 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, the Fifteen Puzzle contains over 10^{13} states, and hence cannot be solved with brute-force techniques on current machines. Faster machines will not have a significant impact on this problem, since the 5×5 Twenty-Four Puzzle contains almost 10^{25} states, for example.

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 isn't 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.*

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 example, 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 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. For

a fixed goal state, a heuristic evaluation is a function of a node, $h(n)$, that estimates the distance from node n to the given goal state.

The key properties of a heuristic evaluation function are that it estimate actual cost, and that it be inexpensive to compute. 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. In addition, most heuristic functions are derived from relaxations of the original problem, and hence are lower bounds on actual cost, a property referred to as *admissibility*. For example, airline 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, 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 distance in grid units from its goal position.

A number of algorithms make use of heuristic functions, including pure heuristic search, the A* algorithm, iterative-deepening-A*, depth-first branch-and-bound, and the heuristic path algorithm. In addition, heuristic information can be employed in bidirectional search as well.

4.2 Pure Heuristic Search

The simplest of these algorithms, pure heuristic search, expands nodes in order of their heuristic values $h(n)$[9]. It maintains a *Closed list* of those nodes that have already been expanded, and an *Open list* of those nodes that have been generated but not yet expanded. The algorithm begins with just the initial state on the Open list. At each cycle, 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 the children, and they are placed on the Open list in order of their heuristic values. The algorithm continues until a goal state 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.

Breadth-first search, uniform-cost search, and pure heuristic search are all special cases of a more general algorithm called *best-first search*. In each cycle 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, and $h(n)$ for pure heuristic search.

4.3 A* Algorithm

The *A* algorithm*[13] 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. Thus, $f(n)$ estimates the lowest total cost of any solution path going through node n. At each point a node with lowest f value is chosen for expansion. Ties among nodes of equal f value should be 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[13]. 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[4].

The main drawback of A*, and indeed of any best-first search, is its memory requirement. Since at least 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. For example, while it can be run successfully on the Eight Puzzle, it exhausts available memory in a matter of minutes on the Fifteen Puzzle.

4.4 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, its path is cut off, and the search backtracks before continuing. The cost threshold is initialized to the heuristic estimate of the initial state, and in each successive iteration is increased to the total cost of the lowest-cost node that was pruned during the previous iteration. The algorithm terminates when a goal state is reached whose total cost does not exceed the current threshold.

Since IDA* performs a series of depth-first searches, its memory requirement is linear with respect to 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 DFID, 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. 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.

4.5 Depth-First Branch-and-Bound

For many problems, the maximum search depth is known in advance, or the search tree is finite. For example, consider the travelling salesman problem (TSP) of visiting each of a given set of cities and returning 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, etc. 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 with respect to the number of cities.

The idea of depth-first branch-and-bound (DFBnB) is to make this search more efficient by keeping track of the lowest-cost solution found so far. Since the cost of a partial tour is the sum of the costs of the edges travelled 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

carefully ordering the children of a given node from smallest to largest estimated total cost, a lower-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.

4.6 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 uniform-cost search, which has exponential complexity.

In general, the time complexity of A* and IDA* is an exponential function of the error in the heuristic function[36]. For example, if the heuristic has constant absolute error, meaning that it never underestimates by more than a constant amount regardless of the magnitude of the estimate, then the running time of A* is linear with respect to the solution cost[11]. A more realistic assumption is constant relative error, which means that the error is a fixed percentage of the quantity being estimated. In that case, the running times of A* and IDA* are exponential[38]. The base of the exponent, however, is smaller than the brute-force branching factor, reducing the asymptotic complexity and allowing larger problems to be solved. For example, using appropriate admissible heuristic functions, IDA* can optimally solve random instances of the Twenty-Four Puzzle[27] and Rubik's Cube[28].

4.7 Heuristic Path Algorithm

Since the complexity of finding optimal solutions to these problems is generally exponential in practice, 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)*[39]. HPA is a best-first search algorithm, where the figure of merit of a node n is $f(n) = (1-w)*g(n) + w*h(n)$. Varying w produces a range of algorithms from uniform-cost search ($w = 0$), through A* ($w = 1/2$), to pure heuristic search ($w = 1$). Increasing w beyond $1/2$ generally decreases the amount of computation, while increasing the cost of the solution generated. This tradeoff is often quite favorable, with small increases in solution cost yielding huge savings in computation[23]. Furthermore, it can be shown that the solutions found by this algorithm are guaranteed to be no more than a factor of $w/(1-w)$ greater than optimal[3], but often are significantly better.

4.8 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 $w \geq 1/2$,

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)[23]. RBFS is a best-first search that runs in space that is linear with respect to 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 immediate 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 deepest 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 [23] for full details on RBFS.

5 Interleaving Search and Execution

In the discussion above, it is assumed that a complete solution can be computed, before even the first step of the solution need be 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, one move must be computed at a time, and that move executed before computing the next. Below we consider algorithms designed for this scenario.

5.1 Minimin Search

Minimin search determines individual single-agent moves in constant time per move[20]. 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 the node. A single move is then made to the 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 five relative to brute-force minimin search on sliding-tile puzzles[20].

Minimin search with branch-and-bound is an algorithm for evaluating the immediate neighbors of the current node. As such, it is run until the 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.

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[20].

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 the neighbor with the minimum $f(n)$ value, and stores with 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[20]. Furthermore, on a tree, RTA* makes locally-optimal decisions given the information it has seen so far.

5.3 Learning-Real-Time-A*

If a problem is to be solved repeatedly with the same goal state but different initial states, one would like an algorithm that improves 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.

6 Two-Player Games

The second major application of heuristic search algorithms in AI is two-player games. 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[50], a goal recently achieved.

6.1 Minimax Search

The standard algorithm for two-player perfect-information games, such as chess, checkers or Othello, is minimax search with heuristic static evaluation[46]. 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 function is applied to the frontier nodes. In this case, a heuristic evaluation 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, appropriately 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 large negative values 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 to the child with the maximum or minimum value is made, depending on whose turn it is to move.

6.2 Alpha-Beta Pruning

One of the most elegant of all AI search algorithms is alpha-beta pruning. While it was in use in the late 1950s, a thorough treatment of the algorithm can be found in [29]. 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 5 shows an example of alpha-beta pruning. Only the labelled 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 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. Thus, we label node g as $<= 3$. 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 we have determined the minimax value of node c, we do not need to evaluate or even generate the brother of node h.

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 achieve another cutoff.

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 q 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 the brother of node q. A similar situation exists after the evaluation of node r at 2. At that point, the value of node o is less than or equal to 1, and the value of node p is less than or equal to 2, hence the value of node n, which is the maximum of the values of nodes o and p, 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 its brother, 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 brother of node n to be pruned, since the value of the root node a is greater than or equal to 4. Thus, we computed the minimax value of the root of the tree to be 4, by generating only seven of sixteen 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 with the same amount of computation. This raises the question of how much does alpha-beta improve performance? The best way to characterize the efficiency of a pruning algorithm is in terms of its *effective branching factor*. The effective branching factor is the d^{th} root of the number of frontier nodes that must be evaluated in a search to depth d, in the limit of large d.

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 effective branching factor is b, the brute-force branching factor.

On the other hand, there is an optimal or perfect ordering in which every possible cutoff is realized. In that case, the effective branching factor is reduced from b to $b^{1/2}$, the square root of the brute-force branching factor. 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 effective branching factor to approximately $b^{3/4}$[35]. This means that one can search 4/3 as deep with alpha-beta, yielding a 33% improvement in search depth.

In practice, however, the effective branching factor of alpha-beta is closer to the best case of $b^{1/2}$ due to *node ordering*. The idea of node ordering is that instead of generating the tree left-to-right, we can reorder the tree based on static evaluations of the interior nodes. In other words, 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.

6.3 Quiescence, Iterative-Deepening, and Transposition Tables

Two other important ideas are quiescence and iterative-deepening. 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 always exploring any capture moves one level deeper.

Iterative-deepening is used to solve the problem of where to set the search horizon[47], 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.

The search graphs of most games, such as chess, 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 is important to detect when a node has already been searched, in order to avoid researching it. 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 searching the tree below the node.

Almost all two-player game programs use full-width, fixed-depth, alpha-beta minimax search with node ordering, quiescence, iterative-deepening, and transposition tables, among other techniques.

6.4 Special Purpose Hardware

While the basic algorithms are described above, much of the performance advances in computer chess have come 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 today are based on special-purpose hardware designed and built only to play chess. For example, DeepBlue is a chess machine that can evaluate about 200 million chess positions per second[17]. In May 1997, it defeated Gary Kasparov, the world champion, in a six-game tournament.

6.5 Multi-Player 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 less effective[21]. Games with chance elements, such as the roll of the dice in backgammon for example, tend to foil 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 the cards in the different hands in bridge. Perhaps poker is one of the ultimate challenges in this area, combining all of the above complexities as well as active deception and the need to model the opponents.

7 Constraint-Satisfaction Problems

In addition to single-agent path-finding problems and two-player games, the third major application of heuristic search is constraint-satisfaction problems. The Eight Queens Problem mentioned previously is a classic example. Other examples include graph coloring, boolean satisfiability, and scheduling problems.

Constraint-satisfaction problems are 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 satisfy the constraint. For example, in a map or graph coloring problem, the variables would represent regions or nodes, and the values would represent colors. The constraints are binary constraints on each pair of adjacent regions or nodes that prohibit them from being assigned the same color.

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 variables that have already been assigned are satisfied. The reason for this is that once a constraint is violated, no assignment to the remaining variables can possibly resatisfy 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 a complete, consistent assignment is found, resulting in success, or all possible assignments are shown to violate some constraint, resulting in failure. Figure 6 shows the tree generated by brute-force backtracking to find all solutions to the Four Queens problem. The tree is searched depth-first to minimize memory requirements.

7.2 Limited Discrepancy Search

Limited discrepancy search (LDS)[14, 26] is a completely general tree-search algorithm, but is most useful in the context of constraint-satisfaction problems in which the entire tree is too large to search exhaustively. In that case, we would like to search that subset of the tree that is most likely to yield a solution in the time available. Assume that we can heuristically order a binary tree so that at any node, the left branch is more likely to lead to a solution than the right branch. LDS then proceeds in a series of depth-first iterations. The first iteration explores just the left-most path in the tree. The second iteration explores those root-to-leaf paths with exactly one right branch, or discrepancy, in them. In general, each iteration explores those paths with exactly k discrepancies, with k ranging from zero to the depth of the tree. The last iteration explores just the rightmost branch. Under certain assumptions, one can show that LDS is likely to find a solution sooner than a strict left-to-right depth-first search.

7.3 Intelligent Backtracking

One can improve the performance of brute-force 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 the search tree. The idea of variable ordering is to order the variables from most constrained to least constrained[10, 42]. For example, if a 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 doesn't 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. In general, one should order the values from least constraining to most constraining, in order to minimize the time required to find a first solution[5, 12].

An important idea, originally called backjumping, is that when an impass 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.

When a variable is assigned a value, the idea of forward checking is to check each remaining uninstantiated variable to make sure that there is at least one assignment for each of them that is consistent with the previous assignments. If not, the original variable is assigned its next value.

7.4 Constraint Recording

In a constraint-satisfaction problem, some constraints are explicitly specified, and others are implied by the 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 don't have to be rediscovered.

A simple example of constraint recording in a preprocessing phase is called arc consistency[10, 30, 33]. 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.

7.5 Heuristic Repair

Backtracking searches a space of consistent partial assignments to variables, in the sense that all constraints among instantiated variables are satisfied, looking 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[32]. For example, in the Eight Queens problem, this amounts to placing all eight queens on the board at the same time, and moving the queens one at a time until a solution is found. The natural heuristic, called min-conflicts, is to move a queen that is in conflict with the most other queens, and move it 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, where it is referred to as GSAT[45]. GSAT can satisfy difficult formulas with several thousand variables, whereas the best backtracking-based approach, the Davis-Putnam algorithm[2] with unit propagation, can only satisfy difficult formulas with several hundred variables.

The main drawback of this approach is that it is not complete, in that it is 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 backtracking will eventually discover that a problem is not solvable.

While constraint-satisfaction problems appear somewhat different from single-agent path-finding problems and two-player games, the 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 a 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.

8 Research Issues and Summary

8.1 Research Issues

The 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 computational limits. Thus, there is a continual demand for faster algorithms.

A related research area is the development of space-efficient algorithms[24]. While the exponential-space algorithms are clearly impractical, the linear-space algorithms use very little of the memory available on current machines. The primary issue here is given a fixed amount of memory, how to

make the best use of it to speed up a search as much as possible. In a two-player game search, the extra memory is used primarily in the transposition table. In single-agent path-finding problems, one of the most effective uses of additional memory is a form of bidirectional search known as perimeter search[8, 31, 15]. The idea is to search breadth-first backward from the goal state until memory is nearly exhausted. Then, the forward search proceeds until it encounters a state on the perimeter of the backward search. The main advantage to this approach is that the nodes on the perimeter can be used to refine the heuristic estimates in the forward search.

Another research area is the development of parallel search algorithms. Most search algorithms have a tremendous amount of potential parallelism, since the basic step of node generation and evaluation is often performed billions 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[44]. In the area of single-agent problems, a dominant theory is that the exact cost of a solution to a simplified version of a problem can be used as an admissible heuristic evaluation function for the original problem[36]. For example, 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, which is an admissible heuristic for the original problem. Automating this approach, however, is still a research problem[41].

Another important research area is the development of selective search algorithms for two-player games. While Deep Blue defeated Gary Kasparov, it did it by evaluating 200 million chess positions per second. Obviously, humans are much more selective in their choices of what positions to examine. The development of alternatives to full-width, fixed-depth minimax search is an active area of research. See [25] for one example of a selective search algorithm, along with pointers to other work in this area.

8.2 Summary

We have described search algorithms for three different classes of problems. In the first, 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 optimality. In the second area, two-player games, 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 the third class of problems, 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 searches and heuristic evaluation functions, can be applied to all three.

9 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 two-player 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, often systematic.

Search tree: A problem-space graph with no cycles.

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.

References

[1] Bolc, L., and J. Cytowski, *Search Methods for Artificial Intelligence*, Academic Press, London, 1992.

[2] Davis, M., and H. Putnam, A computing procedure for quantification theory, *Journal of the Association for Computing Machinery*, Vol. 7, 1960, pp. 201-215.

[3] 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, 1989, pp. 19-28.

[4] Dechter, R., and J. Pearl, Generalized best-first search strategies and the optimality of A*, *Journal of the Association for Computing Machinery*, Vol. 32, No. 3, July 1985, pp. 505-536.

[5] Dechter, R., Pearl, J. 1988. Network-Based Heuristics for Constraint-Satisfaction Problems, *Artificial Intelligence*, Vol. 34, No. 1, 1987, pp. 1-38.

[6] Dijkstra, E.W., A note on two problems in connexion with graphs, *Numerische Mathematik*, 1959, 1:269-71.

[7] Dillenburg, J.F., and P.C. Nelson, Improving the efficiency of depth-first search by cycle elimination, *Information Processing Letters*, Vol. 45, No. 1, pp. 5-10, 1993.

[8] Dillenburg, J.F., and P.C. Nelson, Perimeter search, *Artificial Intelligence*, Vol. 65, No. 1, Jan. 1994, pp. 165-178.

[9] Doran, J.E., and D. Michie, Experiments with the Graph Traverser program, *Proceedings of the Royal Society A*, Vol 294, 1966, pp. 235-259.

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

[11] Gaschnig, J. *Performance measurement and analysis of certain search algorithms*, Ph.D. thesis. Department of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa, 1979.

[12] Haralick, R.M., and G.L. Elliott, Increasing tree search efficiency for constraint satisfaction problems, *Artificial Intelligence*, Vol. 14, 1980, pp. 263-313.

[13] 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*, Vol. 4, no. 2, 1968, pp. 100-107.

[14] Harvey, W.D., and M.L. Ginsberg, Limited discrepancy search, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, Montreal, Canada, Aug. 1995, pp. 607-613.

[15] Kaindl, H., G. Kainz, A. Leeb, and H. Smetana, How to use limited memory in heuristic search, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, Montreal, Canada, Aug. 1995.

[16] Kanal, L., and V. Kumar (Eds.), *Search in Artificial Intelligence*, Springer-Verlag, New York, 1988.

[17] Keene, R., B. Jacobs, and T. Buzan, *Man v Machine: The ACM Chess Challenge: Garry Kasparov v IBM's Deep Blue*, B.B. Enterprises, Sussex, 1996.

[18] Korf, R.E., Depth-first iterative-deepening: An optimal admissible tree search, *Artificial Intelligence*, Vol. 27, No. 1, 1985, pp. 97-109.

[19] Korf, R.E., Search in AI: A survey of recent results, in *Exploring Artificial Intelligence*, H.E. Shrobe (Ed.), Morgan-Kaufmann, Los Altos, Ca. 1988.

[20] Korf, R.E., Real-time heuristic search, *Artificial Intelligence*, Vol. 42, No. 2-3, March 1990, pp. 189-211.

[21] Korf, R.E., Multi-player alpha-beta pruning, *Artificial Intelligence*, Vol. 48, No. 1, February, 1991, pp. 99-111.

[22] Korf, R.E., Search, in the *Encyclopedia of Artificial Intelligence, Second Edition*, John Wiley, New York, 1992, pp. 1460-1467.

[23] Korf, R.E., Linear-space best-first search, *Artificial Intelligence*, Vol. 62, No. 1, July 1993, pp. 41-78.

[24] Korf, R.E., Space-efficient search algorithms, *Computing Surveys*, Vol. 27, No. 3, Sept., 1995, pp. 337-339.

[25] Korf, R.E., and D.M. Chickering, Best-first minimax search, *Artificial Intelligence*, Vol. 84, No. 1-2, July 1996, pp. 299-337.

[26] Korf, R.E., Improved limited discrepancy search, *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, Portland, OR, Aug. 1996, pp. 286-291.

[27] Korf, R.E., and L.A. Taylor, Finding optimal solutions to the twenty-four puzzle, *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, Portland, OR, Aug. 1996, pp. 1202-1207.

[28] 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.

[29] Knuth, D.E., and R.E. Moore, An analysis of alpha-beta pruning, *Artificial Intelligence*, Vol. 6, No. 4, 1975, pp. 293-326.

[30] Mackworth, A.K., Consistency in networks of relations. *Artificial Intelligence* Vol. 8, No. 1, 1977, pp. 99-118.

[31] Manzini, G., BIDA*: An improved perimeter search algorithm, *Artificial Intelligence*, Vol. 75, No. 2, June 1995, pp. 347-360.

[32] 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*, Vol. 58, No. 1-3, December 1992, pp. 161-205.

[33] Montanari, U., Networks of constraints: Fundamental properties and applications to picture processing, *Information Science*, Vol. 7, 1974, pp. 95-132.

[34] Newell, A., and H.A. Simon, *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, N.J., 1972.

[35] Pearl, J., The solution for the branching factor of the Alpha-Beta pruning algorithm and its optimality, *Communications of the Association of Computing Machinery*, Vol. 25, No. 8, 1982, pp. 559-64.

[36] Pearl, J., *Heuristics*, Addison-Wesley, Reading, Ma., 1984.

[37] Pearl, J., and R.E. Korf, Search techniques, in *Annual Review of Computer Science*, Vol. 2, Annual Reviews Inc., Palo Alto, Ca., 1987, pp. 451-467.

[38] Pohl, I., First results on the effect of error in heuristic search, in *Machine Intelligence 5,* B. Meltzer and D. Michie (eds.), American Elsevier, New York, 1970, pp. 219-236.

[39] Pohl, I., Heuristic search viewed as path finding in a graph, *Artificial Intelligence*, Vol. 1, 1970, pp. 193-204.

[40] Pohl, I., Bi-directional search, in *Machine Intelligence 6*, B. Meltzer and D. Michie (eds.), American Elsevier, New York, 1971, pp. 127-140.

[41] Prieditis, A. E. Machine discovery of effective admissible heuristics, *Machine Learning*, Vol. 12, 1993, pp. 117-141.

[42] Purdom, P.W. 1983. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence* 21(1,2):117-33

[43] 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.

[44] 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.

[45] 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.

[46] Shannon, C.E., Programming a computer for playing chess, *Philosophical Magazine*, Vol. 41, 1950, pp. 256-275.

[47] Slate, D.J., and L.R. Atkin, CHESS 4.5 - The Northwestern University chess program, in *Chess Skill in Man and Machine*, Frey, P.W. (Ed.), Springer-Verlag, New York, 1977, pp. 82-118.

[48] 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.

[49] Taylor, L., and R.E. Korf, Pruning duplicate nodes in depth-first search, *Proceedings of the National Conference on Artificial Intelligence (AAAI-93)*, Washington D.C., July 1993, pp. 756-761.

[50] Turing, A.M., Computing machinery and intelligence, *Mind*, Vol. 59, October, 1950, pp. 433-460. Also in *Computers and Thought*, E. Feigenbaum and J. Feldman (Eds.), McGraw-Hill, New York, 1963.

10 Further Information

The classic reference in this area is [36]. More recent survey articles include [37], [19], and [22]. Much of the material in this article was derived from these sources. A number of papers have been collected in an edited volume devoted to search[16]. The most recent book-length treatment of this area is [1]. Most new research in this area initially appears in the Proceedings of the National Conference on Artificial Intelligence (AAAI) or the Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI). Prominent journals in this area include *Artificial Intelligence* (AIJ), the Journal of Artificial Intelligence Research (JAIR), and the *IEEE Transactions on Pattern Analysis and Machine Intelligence* (IEEE TPAMI).

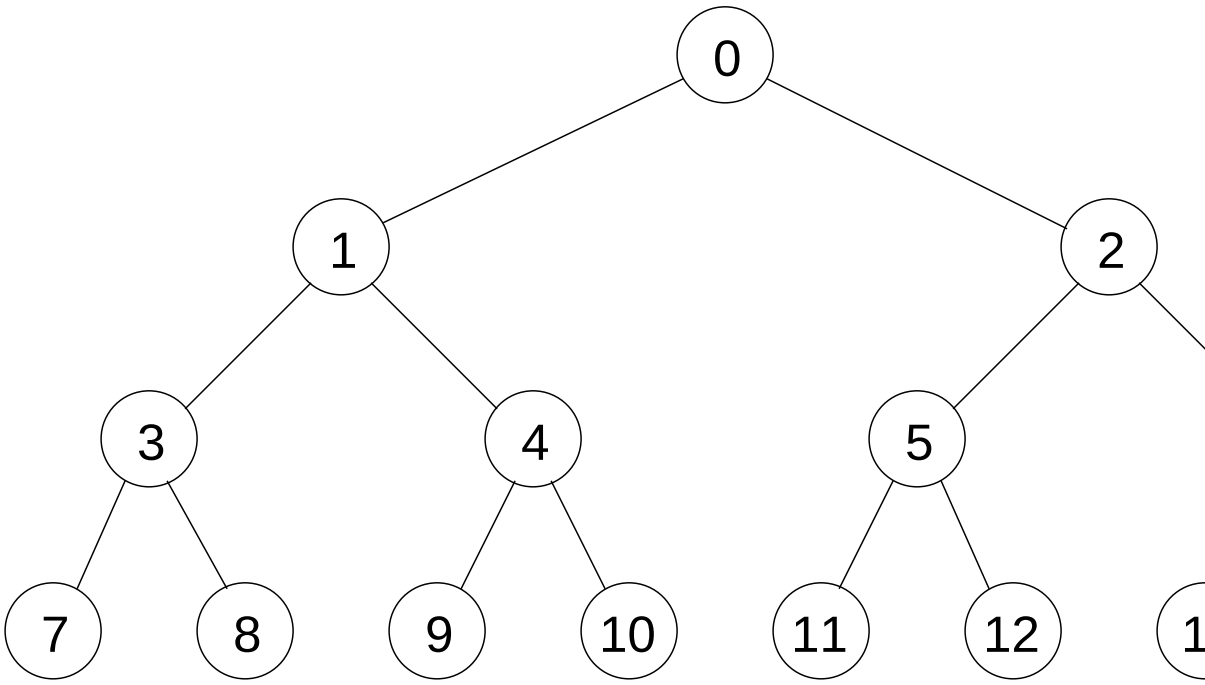

Figure 2: Order of Node Generation for Breadth-First Search

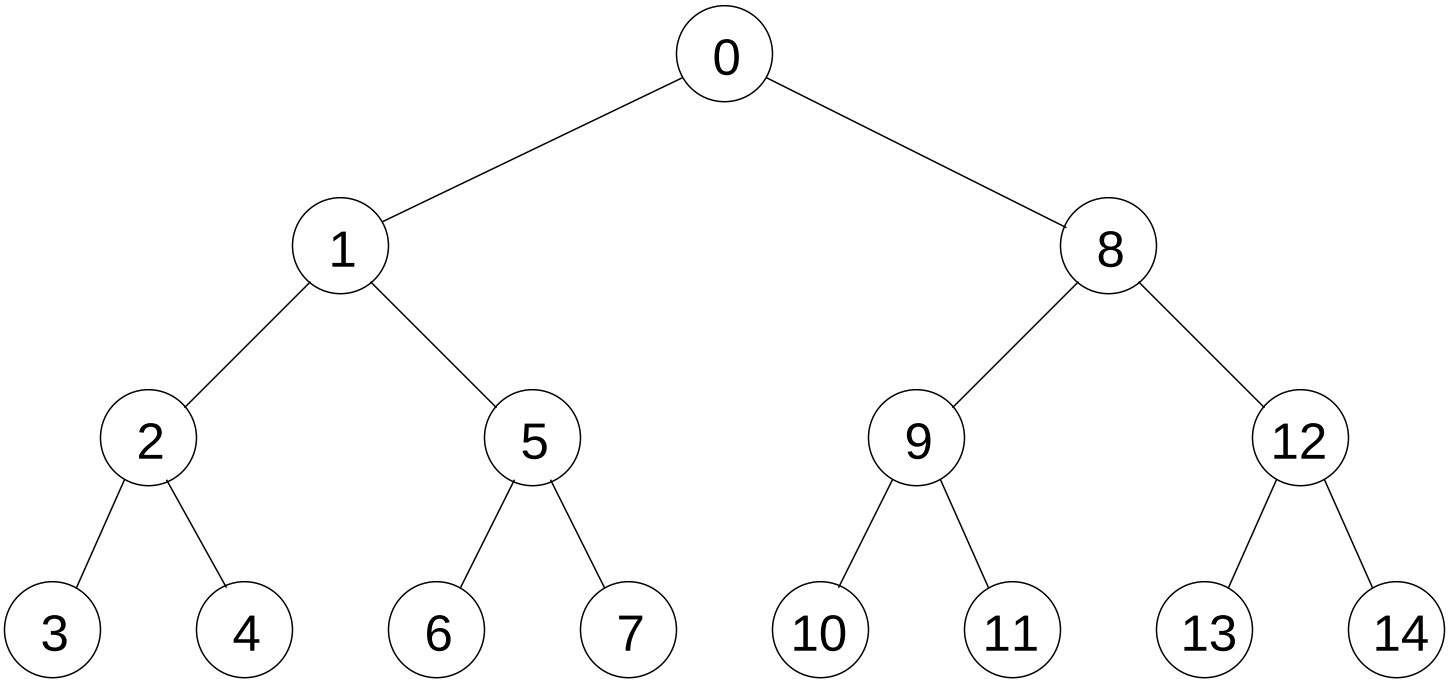

Figure 3: Order of Node Generation for Depth-First Search

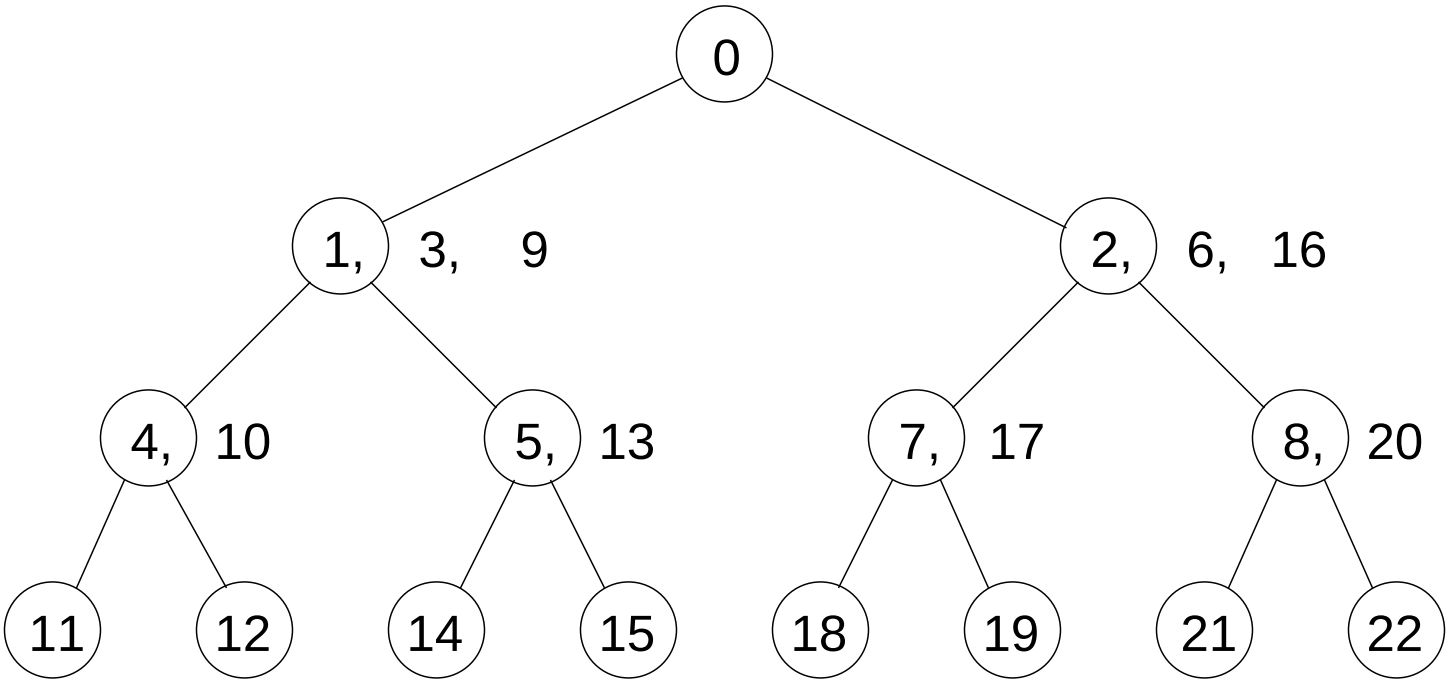

Figure 4: Order of Node Generation for Depth-First Iterative-Deepening Search

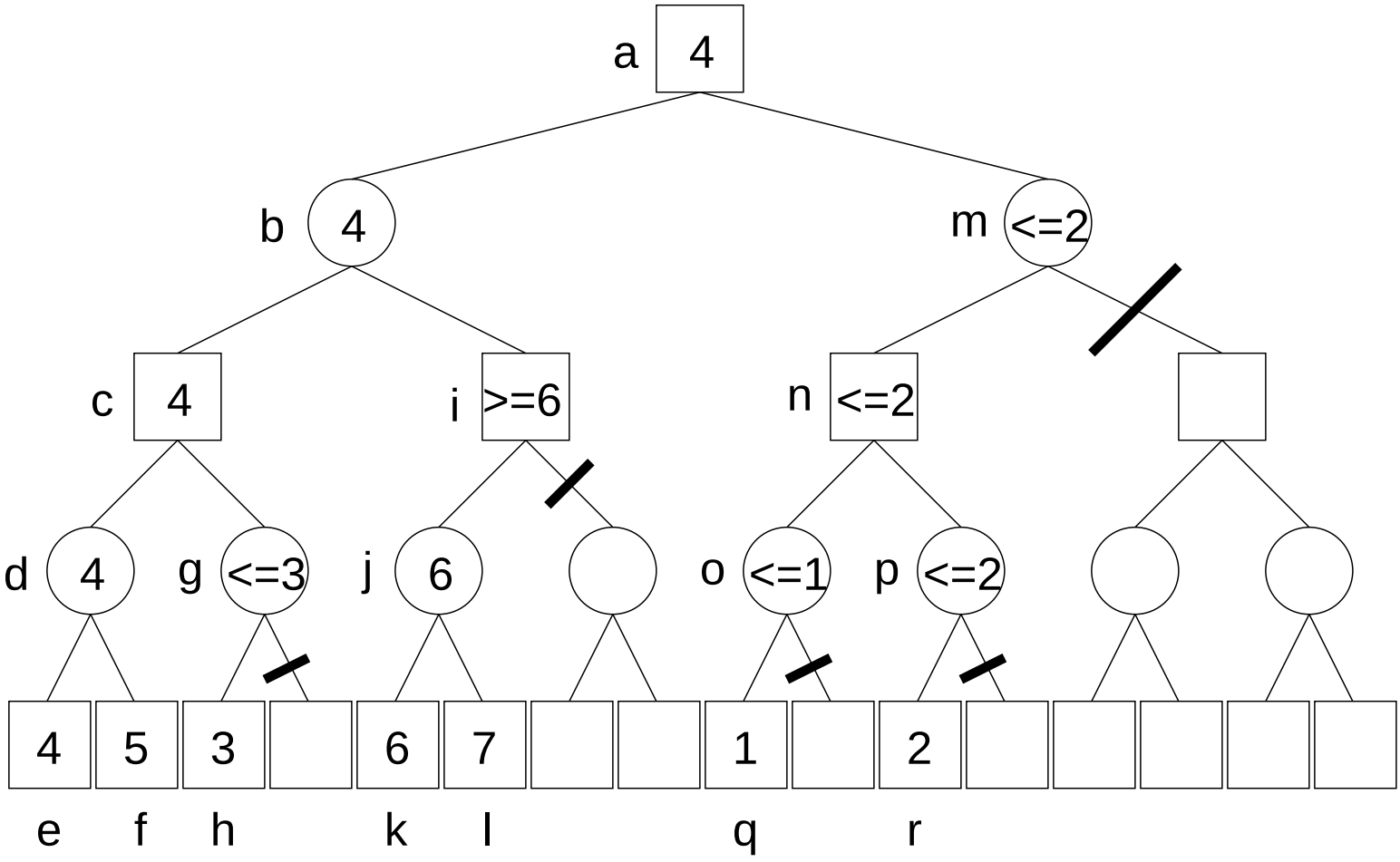

Figure 5: Alpha-Beta Pruning Example

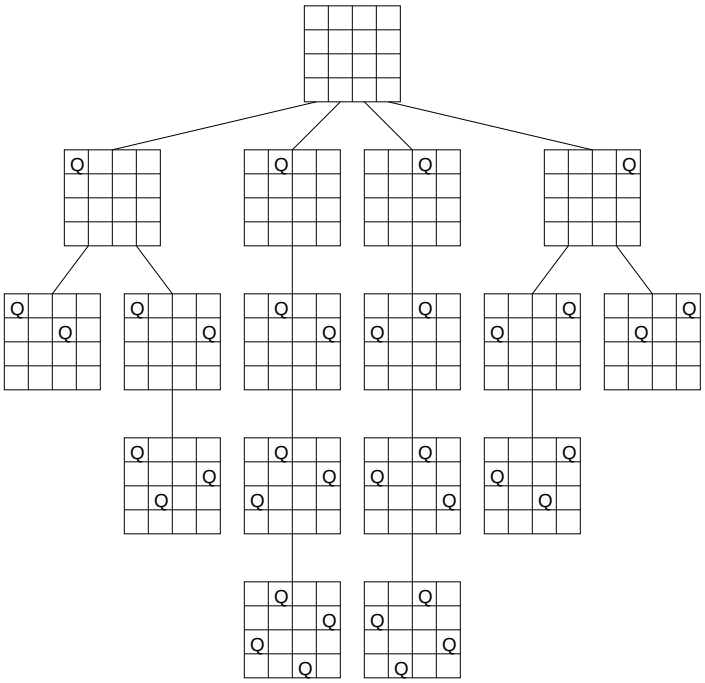

Figure 6: Tree Generated to Solve Four Queens Problem