3 STRUCTURES AND STRATEGIES FOR STATE SPACE SEARCH

In order to cope, an organism must either armor itself (like a tree or a clam) and "hope for the best," or else develop methods for getting out of harm's way and into the better neighborhoods of the vicinity. If you follow this later course, you are confronted with the primordial problem that every agent must continually solve: Now what do I do?

—DANIEL C. DENNETT, *"Consciousness Explained"*

Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;
Then took the other. . .

—ROBERT FROST, *"The Road Not Taken"*

3.0 Introduction

Chapter 2 introduced predicate calculus as an example of an artificial intelligence representation language. Well-formed predicate calculus expressions provide a means of describing objects and relations in a problem domain, and inference rules such as modus ponens allow us to infer new knowledge from these descriptions. These inferences define a space that is searched to find a problem solution. Chapter 3 introduces the theory of state space search.

To successfully design and implement search algorithms, a programmer must be able to analyze and predict their behavior. Questions that need to be answered include:

Is the problem solver guaranteed to find a solution?

Will the problem solver always terminate? Can it become caught in an infinite loop?

When a solution is found, is it guaranteed to be optimal?

What is the complexity of the search process in terms of time usage? Memory usage?

How can the interpreter most effectively reduce search complexity?

How can an interpreter be designed to most effectively utilize a representation language?

The theory of *state space search* is our primary tool for answering these questions. By representing a problem as a *state space graph*, we can use *graph theory* to analyze the structure and complexity of both the problem and the search procedures that we employ to solve it.

A graph consists of a set of *nodes* and a set of *arcs* or *links* connecting pairs of nodes. In the state space model of problem solving, the nodes of a graph are taken to represent discrete *states* in a problem-solving process, such as the results of logical inferences or the different configurations of a game board. The arcs of the graph represent transitions between states. These transitions correspond to logical inferences or legal moves of a game. In expert systems, for example, states describe our knowledge of a problem instance at some stage of a reasoning process. Expert knowledge, in the form of *if . . . then* rules, allows us to generate new information; the act of applying a rule is represented as an arc between states.

Graph theory is our best tool for reasoning about the structure of objects and relations; indeed, this is precisely the need that led to its creation in the early eighteenth century. The Swiss mathematician Leonhard Euler invented graph theory to solve the "bridges of Königsberg problem." The city of Königsberg occupied both banks and two islands of a river. The islands and the riverbanks were connected by seven bridges, as indicated in Figure 3.1.

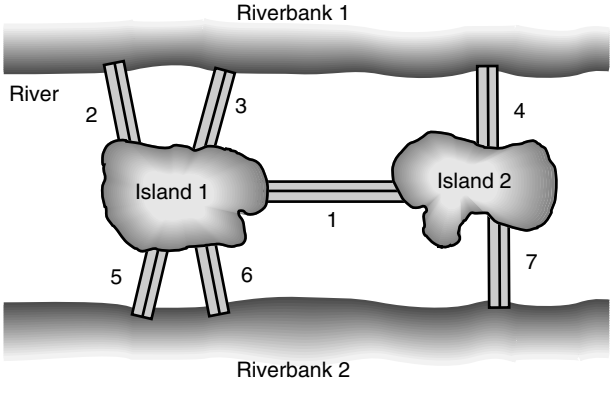


Figure 3.1 The city of Königsberg.

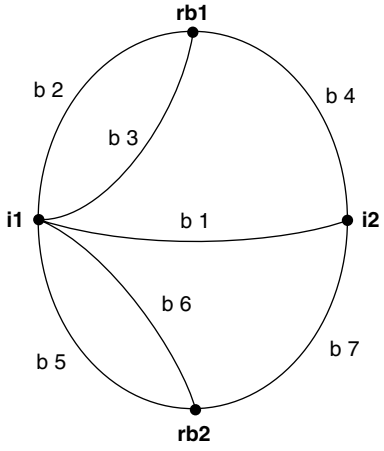


Figure 3.2 Graph of the Königsberg bridge system.

The bridges of Königsberg problem asks if there is a walk around the city that crosses each bridge exactly once. Although the residents had failed to find such a walk and doubted that it was possible, no one had proved its impossibility. Devising a form of graph theory, Euler created an alternative representation for the map, presented in Figure 3.2. The riverbanks (rb1 and rb2) and islands (i1 and i2) are described by the nodes of a graph; the bridges are represented by labeled arcs between nodes (b1, b2, , b7). The graph representation preserves the essential structure of the bridge system, while ignoring extraneous features such as bridge lengths, distances, and order of bridges in the walk.

Alternatively, we may represent the Königsberg bridge system using predicate calculus. The connect predicate corresponds to an arc of the graph, asserting that two land masses are connected by a particular bridge. Each bridge requires two connect predicates, one for each direction in which the bridge may be crossed. A predicate expression, connect(X, Y, Z) = connect (Y, X, Z), indicating that any bridge can be crossed in either direction, would allow removal of half the following connect facts:

```
connect(i1, i2, b1)        connect(i2, i1, b1)
connect(rb1, i1, b2)       connect(i1, rb1, b2)
connect(rb1, i1, b3)       connect(i1, rb1, b3)
connect(rb1, i2, b4)       connect(i2, rb1, b4)
connect(rb2, i1, b5)       connect(i1, rb2, b5)
connect(rb2, i1, b6)       connect(i1, rb2, b6)
connect(rb2, i2, b7)       connect(i2, rb2, b7)
```

The predicate calculus representation is equivalent to the graph representation in that the connectedness is preserved. Indeed, an algorithm could translate between the two representations with no loss of information. However, the structure of the problem can be visualized more directly in the graph representation, whereas it is left implicit in the predicate calculus version. Euler's proof illustrates this distinction.

In proving that the walk was impossible, Euler focused on the *degree* of the nodes of the graph, observing that a node could be of either *even* or *odd* degree. An *even* degree node has an even number of arcs joining it to neighboring nodes. An *odd* degree node has an odd number of arcs. With the exception of its beginning and ending nodes, the desired walk would have to leave each node exactly as often as it entered it. Nodes of odd degree could be used only as the beginning or ending of the walk, because such nodes could be crossed only a certain number of times before they proved to be a dead end. The traveler could not exit the node without using a previously traveled arc.

Euler noted that unless a graph contained either exactly zero or two nodes of odd degree, the walk was impossible. If there were two odd-degree nodes, the walk could start at the first and end at the second; if there were no nodes of odd degree, the walk could begin and end at the same node. The walk is not possible for graphs containing any other number of nodes of odd degree, as is the case with the city of Königsberg. This problem is now called finding an *Euler path* through a graph.

Note that the predicate calculus representation, though it captures the relationships between bridges and land in the city, does not suggest the concept of the degree of a node. In the graph representation there is a single instance of each node with arcs between the nodes, rather than multiple occurrences of constants as arguments in a set of predicates. For this reason, the graph representation suggests the concept of node degree and the focus of Euler's proof. This illustrates graph theory's power for analyzing the structure of objects, properties, and relationships.

In Section 3.1 we review basic graph theory and then present finite state machines and the state space description of problems. In section 3.2 we introduce graph search as a problem-solving methodology. Depth-first and breadth-first search are two strategies for searching a state space. We compare these and make the added distinction between goal-driven and data-driven search. Section 3.3 demonstrates how state space search may be used to characterize reasoning with logic. Throughout the chapter, we use graph theory to analyze the structure and complexity of a variety of problems.

3.1 Structures for State Space Search

3.1.1 Graph Theory (optional)

A *graph* is a set of *nodes* or *states* and a set of *arcs* that connect the nodes. A *labeled* graph has one or more descriptors (labels) attached to each node that distinguish that node from any other node in the graph. In a *state space graph*, these descriptors identify states in a problem-solving process. If there are no descriptive differences between two nodes, they are considered the same. The arc between two nodes is indicated by the labels of the connected nodes.

The arcs of a graph may also be labeled. Arc labels are used to indicate that an arc represents a named relationship (as in a semantic network) or to attach weights to arcs (as in the traveling salesperson problem). If there are different arcs between the same two nodes (as in Figure 3.2), these can also be distinguished through labeling.

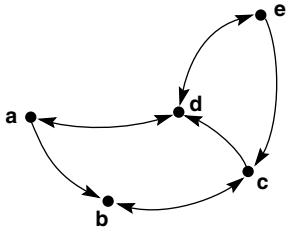
A graph is *directed* if arcs have an associated directionality. The arcs in a directed graph are usually drawn as arrows or have an arrow attached to indicate direction. Arcs that can be crossed in either direction may have two arrows attached but more often have no direction indicators at all. Figure 3.3 is a labeled, directed graph: arc (a, b) may only be crossed from node a to node b, but arc (b, c) is crossable in either direction.

A *path* through a graph connects a sequence of nodes through successive arcs. The path is represented by an ordered list that records the nodes in the order they occur in the path. In Figure 3.3, [a, b, c, d] represents the path through nodes a, b, c, and d, in that order.

A *rooted* graph has a unique node, called the *root*, such that there is a path from the root to all nodes within the graph. In drawing a rooted graph, the root is usually drawn at the top of the page, above the other nodes. The state space graphs for games are usually rooted graphs with the start of the game as the root. The initial moves of the tic-tac-toe game graph are represented by the rooted graph of Figure II.5. This is a directed graph with all arcs having a single direction. Note that this graph contains no cycles; players cannot (as much as they might sometimes wish!) undo a move.

A *tree* is a graph in which two nodes have at most one path between them. Trees often have roots, in which case they are usually drawn with the root at the top, like a rooted graph. Because each node in a tree has only one path of access from any other node, it is impossible for a path to *loop* or *cycle* through a sequence of nodes.

For rooted trees or graphs, relationships between nodes include *parent*, *child*, and *sibling*. These are used in the usual familial fashion with the parent preceding its child along a directed arc. The children of a node are called *siblings*. Similarly, an *ancestor* comes before a *descendant* in some path of a directed graph. In Figure 3.4, b is a *parent* of nodes e and f (which are, therefore, *children* of b and *siblings* of each other). Nodes a and c are *ancestors* of states g, h, and i, and g, h, and i are *descendants* of a and c.



Nodes = **{a,b,c,d,e}**
Arcs = **{(a,b),(a,d),(b,c),(c,b),(c,d),(d,a),(d,e),(e,c),(e,d)}**

Figure 3.3 A labeled directed graph.

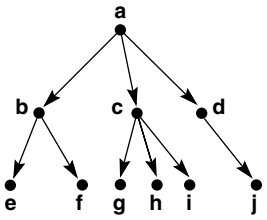


Figure 3.4 A rooted tree, exemplifying family relationships.

Before introducing the state space representation of problems we formally define these concepts.

DEFINITION

GRAPH

A graph consists of:

A set of *nodes* N_1, N_2, N_3, ..., N_n, ..., which need not be finite.

A set of *arcs* that connect pairs of nodes.

Arcs are ordered pairs of nodes; i.e., the arc (N_3, N_4) connects node N_3 to node N_4. This indicates a direct connection from node N_3 to N_4 but not from N_4 to N_3, unless (N_4, N_3) is also an arc, and then the arc joining N_3 and N_4 is *undirected*.

If a *directed* arc connects N_j and N_k, then N_j is called the *parent* of N_k and N_k, the *child* of N_j. If the graph also contains an arc (N_j, N_l), then N_k and N_l are called *siblings*.

A *rooted* graph has a unique node N_s from which all paths in the graph originate. That is, the root has no parent in the graph.

A *tip* or *leaf* node is a node that has no children.

An ordered sequence of nodes [N_1, N_2, N_3, ..., N_n], where each pair N_i, N_{i+1} in the sequence represents an arc, i.e., (N_i, N_{i+1}), is called a *path* of length $n - 1$.

On a path in a rooted graph, a node is said to be an *ancestor* of all nodes positioned after it (to its right) as well as a *descendant* of all nodes before it.

A path that contains any node more than once (some N_j in the definition of path above is repeated) is said to contain a *cycle* or *loop.*

A *tree* is a graph in which there is a unique path between every pair of nodes. (The paths in a tree, therefore, contain no cycles.)

The edges in a rooted tree are directed away from the root. Each node in a rooted tree has a unique parent.

Two nodes are said to be *connected* if a path exists that includes them both.

Next we introduce the finite state machine, an abstract representation for computational devices, that may be viewed as an automaton for traversing paths in a graph.

3.1.2 The Finite State Machine (optional)

We can think of a machine as a system that accepts input values, possibly produces output values, and that has some sort of internal mechanism (states) to keep track of information about previous input values. A *finite state machine* (FSM) is a finite, directed, connected graph, having a set of states, a set of input values, and a state transition function that describes the effect that the elements of the input stream have on the states of the graph. The stream of input values produces a path within the graph of the states of this finite machine. Thus the FSM can be seen as an abstract model of computation.

The primary use for such a machine is to recognize components of a formal language. These components are often strings of characters ("words" made from characters of an "alphabet"). In Section 5.3 we extend this definition to a probabilistic finite state machine. These state machines have an important role in analyzing expressions in languages, whether computational or human, as we see in Sections 5.3, 9.3, and Chapter 15.

DEFINITION

FINITE STATE MACHINE (FSM)

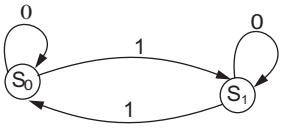
A *finite state machine* is an ordered triple (S, I, F), where:

S is a finite set of *states* in a connected graph $s_1, s_2, s_3, \ldots s_n$.

I is a finite set of *input* values $i_1, i_2, i_3, \ldots i_m$.

F is a state transition function that for any $i \in I$, describes its effect on the states S of the machine, thus $\forall\, i \in I$, $F_i : (S \rightarrow S)$. If the machine is in state s_j and input i occurs, the next state of the machine will be $F_i(s_j)$.

For a simple example of a finite state machine, let $S = \{s_0, s_1\}$, $I = \{0,1\}$, $f_0(s_0) = s_0$, $f_0(s_1) = (s_1)$, $f_1(s_0) = s_1$, and $f_1(s_1) = s_0$. With this device, sometimes called a *flip-flop*, an input value of zero leaves the state unchanged, while input 1 changes the state of the machine. We may visualize this machine from two equivalent perspectives, as a finite graph with labelled, directed arcs, as in Figure 3.5a, or as a transition matrix, Figure 3.5b. In the transition matrix, input values are listed along the top row, the states are in the left-most column, and the output for an input applied to a state is at the intersection point.



(a)

	0	1
S_0	S_0	S_1
S_1	S_1	S_0

(b)

Figure 3.5 (a) The finite state graph for a flip-flop and (b) its transition matrix.

A second example of a finite state machine is represented by the directed graph of Figure 3.6a and the (equivalent) transition matrix of Figure 3.6b. One might ask what the finite state machine of Figure 3.6 could represent. With two assumptions, this machine could be seen as a recognizer of all strings of characters from the alphabet {a, b, c, d} that contain the exact sequence "abc". The two assumptions are, first, that state s_0 has a special role as the *starting state*, and second, that s_3 is the *accepting state*. Thus, the input stream will present its first element to state s_0. If the stream later terminates with the machine in state s_3, it will have recognized that there is the sequence "abc" within that input stream.

What we have just described is a *finite state accepting machine*, sometimes called a *Moore machine*. We use the convention of placing an arrow from no state that terminates in the starting state of the Moore machine, and represent the accepting state (or states) as special, often using a doubled circle, as in Figure 3.6. We now present a formal definition of the Moore machine:

DEFINITION

FINITE STATE ACCEPTOR (MOORE MACHINE)

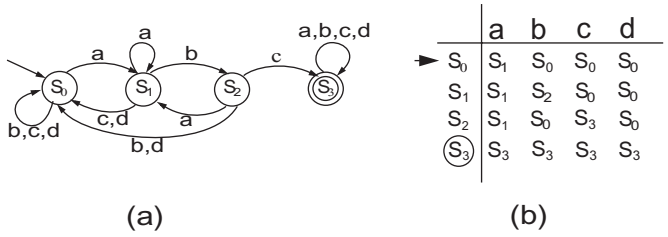
A *finite state acceptor* is a finite state machine (S, I, *F*), where:

$\exists\, s_0 \in S$ such that the input stream starts at s_0, and

$\exists\, s_n \in S$, an *accept* state. The input stream is accepted if it terminates in that state. In fact, there may be a set of accept states.

The finite state acceptor is represented as $(S, s_0, \{s_n\}, I, F)$

We have presented two fairly simple examples of a powerful concept. As we will see in natural language understanding (Chapter 15), finite state recognizers are an important tool for determining whether or not patterns of characters, words, or sentences have desired properties. We will see that a finite state accepter implicitly defines a formal language on the basis of the sets of letters (characters) and words (strings) that it accepts.



	a	b	c	d
→ S_0	S_1	S_0	S_0	S_0
S_1	S_1	S_2	S_0	S_0
S_2	S_1	S_0	S_3	S_0
(S_3)	S_3	S_3	S_3	S_3

Figure 3.6 (a) The finite state graph and (b) the transition matrix for string recognition example.

We have also shown only *deterministic* finite state machines, where the transition function for any input value to a state gives a unique next state. *Probabilistic finite state machines*, where the transition function defines a distribution of output states for each input to a state, are also an important modeling technique. We consider these in Section 5.3 and again in Chapter 15. We next consider a more general graphical representation for the analysis of problem solving: the state space.

3.1.3 The State Space Representation of Problems

In the *state space representation* of a problem, the nodes of a graph correspond to partial problem solution *states* and the arcs correspond to steps in a problem-solving process. One or more *initial states*, corresponding to the given information in a problem instance, form the root of the graph. The graph also defines one or more *goal* conditions, which are solutions to a problem instance. *State space search* characterizes problem solving as the process of finding a *solution path* from the start state to a goal.

A goal may describe a state, such as a winning board in tic-tac-toe (Figure II.5) or a goal configuration in the 8-puzzle (Figure 3.7). Alternatively, a goal can describe some property of the solution path itself. In the traveling salesperson problem (Figures 3.9 and 3.10), search terminates when the "shortest" path is found through all nodes of the graph. In the parsing problem (Section 3.3), the solution path is a successful analysis of a sentence's structure.

Arcs of the state space correspond to steps in a solution process and paths through the space represent solutions in various stages of completion. Paths are searched, beginning at the start state and continuing through the graph, until either the goal description is satisfied or they are abandoned. The actual generation of new states along the path is done by applying operators, such as "legal moves" in a game or inference rules in a logic problem or expert system, to existing states on a path. The task of a search algorithm is to find a solution path through such a problem space. Search algorithms must keep track of the paths from a start to a goal node, because these paths contain the series of operations that lead to the problem solution.

We now formally define the state space representation of problems:

DEFINITION

STATE SPACE SEARCH

A *state space* is represented by a four-tuple [N,A,S,GD], where:

N is the set of nodes or states of the graph. These correspond to the states in a problem-solving process.

A is the set of arcs (or links) between nodes. These correspond to the steps in a problem-solving process.

S, a nonempty subset of N, contains the start state(s) of the problem.

GD, a nonempty subset of N, contains the goal state(s) of the problem. The states in GD are described using either:

1. A measurable property of the states encountered in the search.
2. A measurable property of the path developed in the search, for example, the sum of the transition costs for the arcs of the path.

A *solution path* is a path through this graph from a node in S to a node in GD.

One of the general features of a graph, and one of the problems that arise in the design of a graph search algorithm, is that states can sometimes be reached through different paths. For example, in Figure 3.3 a path can be made from state a to state d either through b and c or directly from a to d. This makes it important to choose the *best* path according to the needs of a problem. In addition, multiple paths to a state can lead to loops or cycles in a solution path that prevent the algorithm from reaching a goal. A blind search for goal state e in the graph of Figure 3.3 might search the sequence of states abcdabcdabcd . . . forever!

If the space to be searched is a tree, as in Figure 3.4, the problem of cycles does not occur. It is, therefore, important to distinguish between problems whose state space is a tree and those that may contain loops. General graph search algorithms must detect and eliminate loops from potential solution paths, whereas tree searches may gain efficiency by eliminating this test and its overhead.

Tic-tac-toe and the 8-puzzle exemplify the state spaces of simple games. Both of these examples demonstrate termination conditions of type 1 in our definition of state space search. Example 3.1.3, the traveling salesperson problem, has a goal description of type 2, the total cost of the path itself.

EXAMPLE 3.1.1: TIC-TAC-TOE

The state space representation of tic-tac-toe appears in Figure II.5, pg 42. The start state is an empty board, and the termination or goal description is a board state having three Xs in a row, column, or diagonal (assuming that the goal is a win for X). The path from the start state to a goal state gives the series of moves in a winning game.

The states in the space are all the different configurations of Xs and Os that the game can have. Of course, although there are 3^9 ways to arrange {blank, X, O} in nine spaces, most of them would never occur in an actual game. Arcs are generated by legal moves of the game, alternating between placing an X and an O in an unused location. The state space is a graph rather than a tree, as some states on the third and deeper levels can be reached by different paths. However, there are no cycles in the state space, because the directed arcs of the graph do not allow a move to be undone. It is impossible to "go back up" the structure once a state has been reached. No checking for cycles in path generation is necessary. A graph structure with this property is called a *directed acyclic graph*, or *DAG*, and is common in state space search and in graphical models, Chapter 13.

The state space representation provides a means of determining the complexity of the problem. In tic-tac-toe, there are nine first moves with eight possible responses to each of them, followed by seven possible responses to each of these, and so on. It follows that $9 \times 8 \times 7 \times ...$ or 9! different paths can be generated. Although it is not impossible for a computer to search this number of paths (362,880) exhaustively, many important problems also exhibit factorial or exponential complexity, although on a much larger scale. Chess has 10^{120} possible game paths; checkers has 10^{40}, some of which may never occur in an actual game. These spaces are difficult or impossible to search exhaustively. Strategies for searching such large spaces often rely on heuristics to reduce the complexity of the search (Chapter 4).

EXAMPLE 3.1.2: THE 8-PUZZLE

In the *15-puzzle* of Figure 3.7, 15 differently numbered tiles are fitted into 16 spaces on a grid. One space is left blank so that tiles can be moved around to form different patterns. The goal is to find a series of moves of tiles into the blank space that places the board in a goal configuration. This is a common game that most of us played as children. (The version I remember was about 3 inches square and had red and white tiles in a black frame.)

A number of interesting aspects of this game have made it useful to researchers in problem solving. The state space is large enough to be interesting but is not completely intractable (16! if symmetric states are treated as distinct). Game states are easy to represent. The game is rich enough to provide a number of interesting heuristics (see Chapter 4).

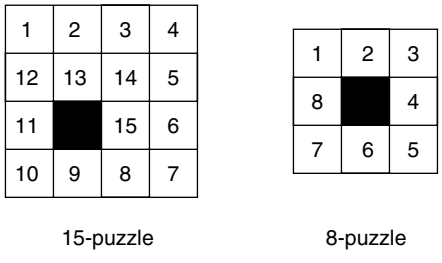


Figure 3.7 The 15-puzzle and the 8-puzzle.

The *8-puzzle* is a 3 × 3 version of the 15-puzzle in which eight tiles can be moved around in nine spaces. Because the 8-puzzle generates a smaller state space than the full 15-puzzle and its graph fits easily on a page, it is used for many examples in this book.

Although in the physical puzzle moves are made by moving tiles ("move the 7 tile right, provided the blank is to the right of the tile" or "move the 3 tile down"), it is much simpler to think in terms of "moving the blank space". This simplifies the definition of move rules because there are eight tiles but only a single blank. In order to apply a move, we must make sure that it does not move the blank off the board. Therefore, all four moves are not applicable at all times; for example, when the blank is in one of the corners only two moves are possible.

The legal moves are:

move the blank up ↑
move the blank right →
move the blank down ↓
move the blank left ←

If we specify a beginning state and a goal state for the 8-puzzle, it is possible to give a state space accounting of the problem-solving process (Figure 3.8). States could be

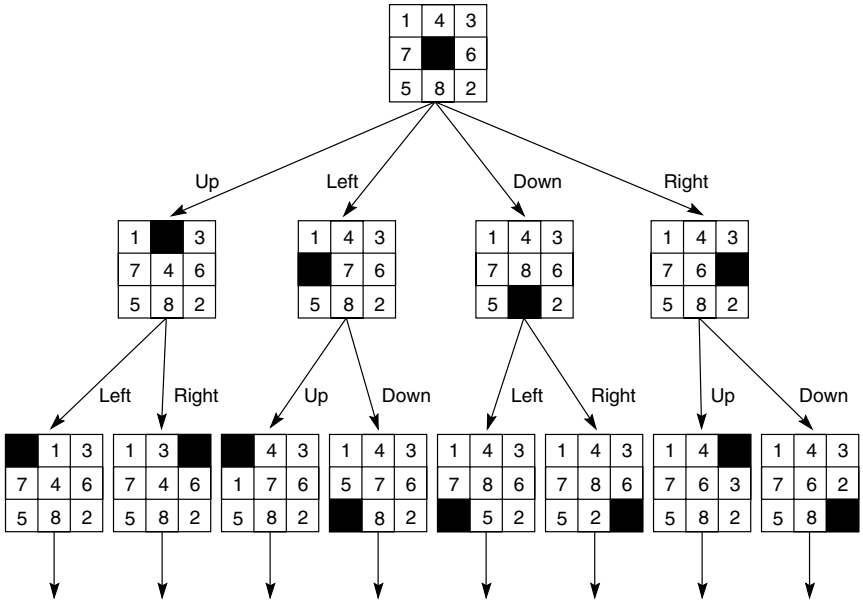


Figure 3.8 State space of the 8-puzzle generated by move blank operations.

represented using a simple 3 × 3 array. A predicate calculus representation could use a "state" predicate with nine parameters (for the locations of numbers in the grid). Four procedures, describing each of the possible moves of the blank, define the arcs in the state space.

As with tic-tac-toe, the state space for the 8-puzzle is a graph (with most states having multiple parents), but unlike tic-tac-toe, cycles are possible. The GD or goal description of the state space is a particular state or board configuration. When this state is found on a path, the search terminates. The path from start to goal is the desired series of moves.

It is interesting to note that the complete state space of the 8- and 15-puzzles consists of two disconnected (and in this case equal-sized) subgraphs. This makes half the possible states in the search space impossible to reach from any given start state. If we exchange (by prying loose!) two immediately adjacent tiles, states in the other component of the space become reachable.

EXAMPLE 3.1.3: THE TRAVELING SALESPERSON

Suppose a salesperson has five cities to visit and then must return home. The goal of the problem is to find the shortest path for the salesperson to travel, visiting each city, and then returning to the starting city. Figure 3.9 gives an instance of this problem. The nodes of the graph represent cities, and each arc is labeled with a weight indicating the cost of traveling that arc. This cost might be a representation of the miles necessary in car travel or cost of an air flight between the two cities. For convenience, we assume the salesperson lives in city A and will return there, although this assumption simply reduces the problem of N cities to a problem of (N - 1) cities.

The path [A,D,C,B,E,A], with associated cost of 450 miles, is an example of a possible circuit. The goal description requires a complete circuit with minimum cost. Note

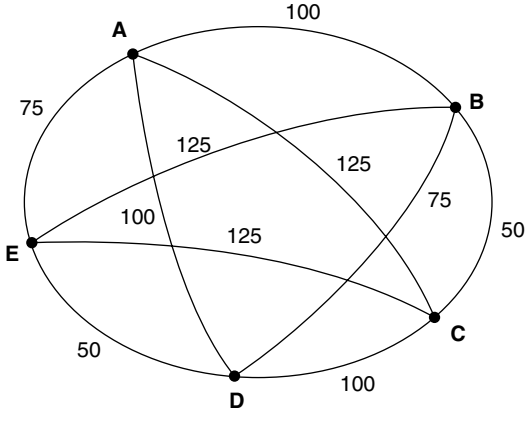


Figure 3.9 An instance of the traveling salesperson problem.

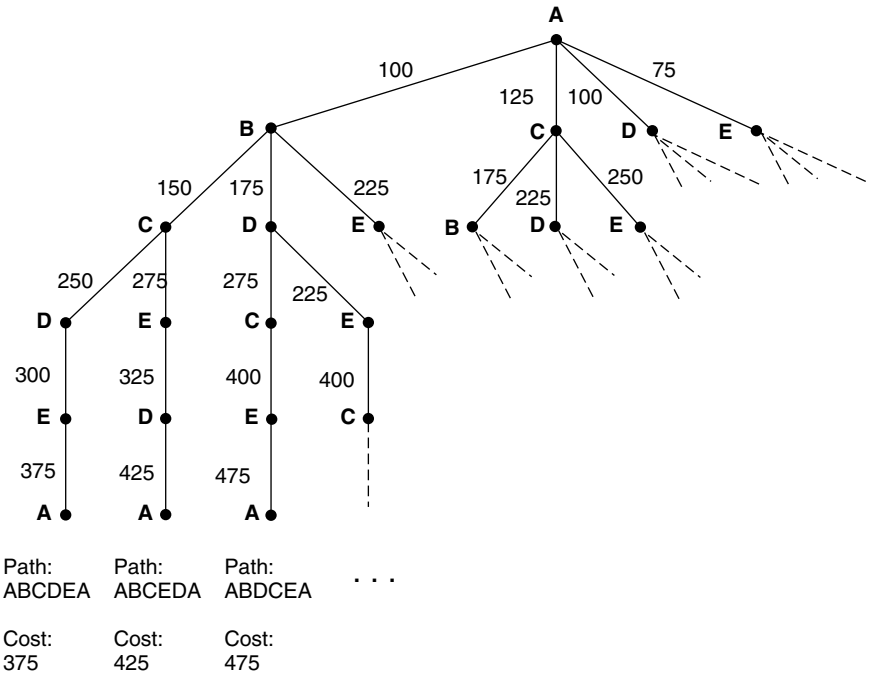


Figure 3.10 Search of the traveling salesperson problem. Each arc is marked with the total weight of all paths from the start node (A) to its endpoint.

that the goal description is a property of the entire path, rather than of a single state. This is a goal description of type 2 from the definition of state space search.

Figure 3.10 shows one way in which possible solution paths may be generated and compared. Beginning with node A, possible next states are added until all cities are included and the path returns home. The goal is the lowest-cost path.

As Figure 3.10 suggests, the complexity of exhaustive search in the traveling salesperson problem is (N - 1)!, where N is the number of cities in the graph. For 9 cities we may exhaustively try all paths, but for any problem instance of interesting size, for example with 50 cities, simple exhaustive search cannot be performed within a practical length of time. In fact complexity costs for an N! search grow so fast that very soon the search combinations become intractable.

Several techniques can reduce this search complexity. One is called *branch and bound* (Horowitz and Sahni 1978). Branch and bound generates paths one at a time, keeping track of the best circuit found so far. This value is used as a *bound* on future candidates. As paths are constructed one city at a time, the algorithm examines each partially completed path. If the algorithm determines that the best possible extension to a path, the branch, will have greater cost than the bound, it eliminates that partial path and *all* of its possible extensions. This reduces search considerably but still leaves an exponential number of paths (1.26^N rather than N!).

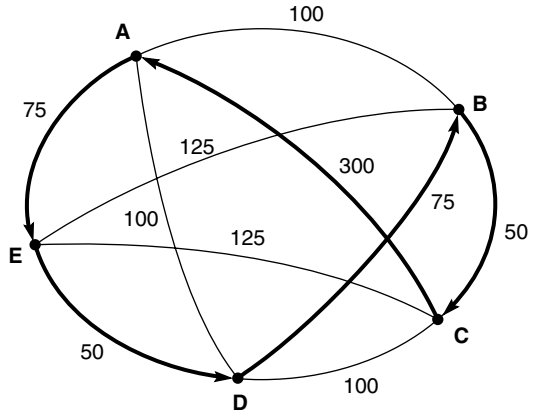


Figure 3.11 An instance of the traveling salesperson problem with the nearest neighbor path in bold. Note that this path (A, E, D, B, C, A), at a cost of 550, is not the shortest path. The comparatively high cost of arc (C, A) defeated the heuristic.

Another strategy for controlling search constructs the path according to the rule "go to the closest unvisited city." The *nearest neighbor* path through the graph of Figure 3.11 is [A,E,D,B,C,A], at a cost of 375 miles. This method is highly efficient, as there is only one path to be tried! The nearest neighbor, sometimes called *greedy*, heuristic is fallible, as graphs exist for which it does not find the shortest path, see Figure 3.11, but it is a possible compromise when the time required makes exhaustive search impractical.

Section 3.2 examines strategies for state space search.

3.2 Strategies for State Space Search

3.2.1 Data-Driven and Goal-Driven Search

A state space may be searched in two directions: from the given data of a problem instance toward a goal or from a goal back to the data.

In *data-driven search*, sometimes called *forward chaining*, the problem solver begins with the given facts of the problem and a set of legal moves or rules for changing state. Search proceeds by applying rules to facts to produce new facts, which are in turn used by the rules to generate more new facts. This process continues until (we hope!) it generates a path that satisfies the goal condition.

An alternative approach is possible: take the goal that we want to solve. See what rules or legal moves could be used to generate this goal and determine what conditions must be true to use them. These conditions become the new goals, or *subgoals*, for the search. Search continues, working backward through successive subgoals until (we hope!)

it works back to the facts of the problem. This finds the chain of moves or rules leading from data to a goal, although it does so in backward order. This approach is called *goal-driven* reasoning, or *backward chaining*, and it recalls the simple childhood trick of trying to solve a maze by working back from the finish to the start.

To summarize: data-driven reasoning takes the facts of the problem and applies the rules or legal moves to produce new facts that lead to a goal; goal-driven reasoning focuses on the goal, finds the rules that could produce the goal, and chains backward through successive rules and subgoals to the given facts of the problem.

In the final analysis, both data-driven and goal-driven problem solvers search the same state space graph; however, the order and actual number of states searched can differ. The preferred strategy is determined by the properties of the problem itself. These include the complexity of the rules, the "shape" of the state space, and the nature and availability of the problem data. All of these vary for different problems.

As an example of the effect a search strategy can have on the complexity of search, consider the problem of confirming or denying the statement "I am a descendant of Thomas Jefferson." A solution is a path of direct lineage between the "I" and Thomas Jefferson. This space may be searched in two directions, starting with the "I" and working along ancestor lines to Thomas Jefferson or starting with Thomas Jefferson and working through his descendants.

Some simple assumptions let us estimate the size of the space searched in each direction. Thomas Jefferson was born about 250 years ago; if we assume 25 years per generation, the required path will be about length 10. As each person has exactly two parents, a search back from the "I" would examine on the order of 2^{10} ancestors. A search that worked forward from Thomas Jefferson would examine more states, as people tend to have more than two children (particularly in the eighteenth and nineteenth centuries). If we assume an average of only three children per family, the search would examine on the order of 3^{10} nodes of the family tree. Thus, a search back from the "I" would examine fewer nodes. Note, however, that both directions yield exponential complexity.

The decision to choose between data- and goal-driven search is based on the structure of the problem to be solved. Goal-driven search is suggested if:

1. A goal or hypothesis is given in the problem statement or can easily be formulated. In a mathematics theorem prover, for example, the goal is the theorem to be proved. Many diagnostic systems consider potential diagnoses in a systematic fashion, confirming or eliminating them using goal-driven reasoning.
2. There are a large number of rules that match the facts of the problem and thus produce an increasing number of conclusions or goals. Early selection of a goal can eliminate most of these branches, making goal-driven search more effective in pruning the space (Figure 3.12). In a theorem prover, for example, the total number of rules used to produce a given theorem is usually much smaller than the number of rules that may be applied to the entire set of axioms.
3. Problem data are not given but must be acquired by the problem solver. In this case, goal-driven search can help guide data acquisition. In a medical diagnosis program, for example, a wide range of diagnostic tests can be applied. Doctors order only those that are necessary to confirm or deny a particular hypothesis.

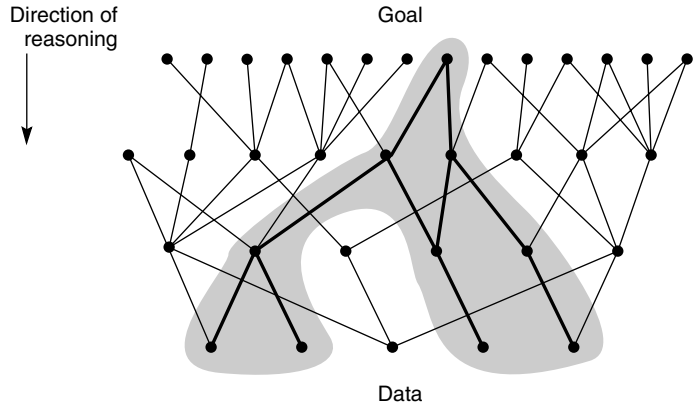


Figure 3.12 State space in which goal-directed search effectively prunes extraneous search paths.

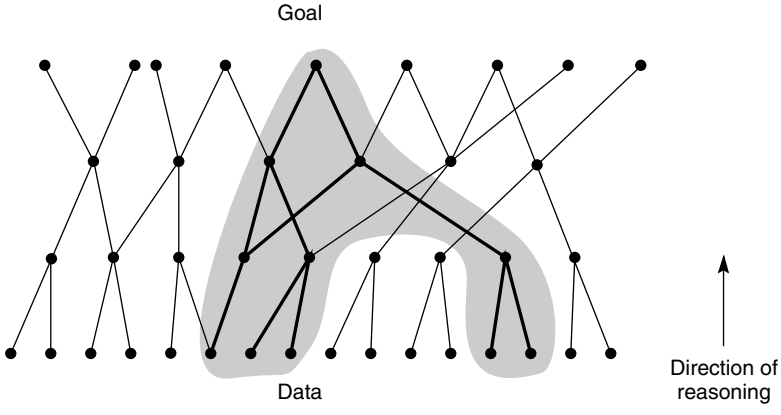


Figure 3.13 State space in which data-directed search prunes irrelevant data and their consequents and determines one of a number of possible goals.

Goal-driven search thus uses knowledge of the desired goal to guide the search through relevant rules and eliminate branches of the space.

Data-driven search (Figure 3.13) is appropriate for problems in which:

1. All or most of the data are given in the initial problem statement. Interpretation problems often fit this mold by presenting a collection of data and asking the system to provide a high-level interpretation. Systems that analyze particular data (e.g., the PROSPECTOR or Dipmeter programs, which interpret geological data

or attempt to find what minerals are likely to be found at a site) fit the data-driven approach.

2. There are a large number of potential goals, but there are only a few ways to use the facts and given information of a particular problem instance. The DENDRAL program, an expert system that finds the molecular structure of organic compounds based on their formula, mass spectrographic data, and knowledge of chemistry, is an example of this. For any organic compound, there are an enormous number of possible structures. However, the mass spectrographic data on a compound allow DENDRAL to eliminate all but a few of these.
3. It is difficult to form a goal or hypothesis. In using DENDRAL, for example, little may be known initially about the possible structure of a compound.

Data-driven search uses the knowledge and constraints found in the given data of a problem to guide search along lines known to be true.

To summarize, there is no substitute for careful analysis of the particular problem to be solved, considering such issues as the *branching factor* of rule applications (see Chapter 4; on average, how many new states are generated by rule applications in both directions?), availability of data, and ease of determining potential goals.

3.2.2 Implementing Graph Search

In solving a problem using either goal- or data-driven search, a problem solver must find a path from a start state to a goal through the state space graph. The sequence of arcs in this path corresponds to the ordered steps of the solution. If a problem solver were given an oracle or other infallible mechanism for choosing a solution path, search would not be required. The problem solver would move unerringly through the space to the desired goal, constructing the path as it went. Because oracles do not exist for interesting problems, a problem solver must consider different paths through the space until it finds a goal. *Backtracking* is a technique for systematically trying all paths through a state space.

We begin with backtrack because it is one of the first search algorithms computer scientists study, and it has a natural implementation in a stack oriented recursive environment. We will present a simpler version of the backtrack algorithm with *depth-first search* (Section 3.2.3).

Backtracking search begins at the start state and pursues a path until it reaches either a goal or a "dead end." If it finds a goal, it quits and returns the solution path. If it reaches a dead end, it "backtracks" to the most recent node on the path having unexamined siblings and continues down one of these branches, as described in the following recursive rule:

> If the present state S does not meet the requirements of the goal description, then generate its first descendant S_{child1}, and apply the backtrack procedure recursively to this node. If backtrack does not find a goal node in the subgraph rooted at S_{child1}, repeat the procedure for its sibling, S_{child2}. Continue until either some descendant of a child is a goal node or all the children have been searched. If none of the children of S leads to a goal, then backtrack "fails back" to the parent of S, where it is applied to the siblings of S, and so on.

The algorithm continues until it finds a goal or exhausts the state space. Figure 3.14 shows the backtrack algorithm applied to a hypothetical state space. The direction of the dashed arrows on the tree indicates the progress of search up and down the space. The number beside each node indicates the order in which it is visited. We now define an algorithm that performs a backtrack, using three lists to keep track of nodes in the state space:

SL, for state list, lists the states in the current path being tried. If a goal is found, SL contains the ordered list of states on the solution path.

NSL, for new state list, contains nodes awaiting evaluation, i.e., nodes whose descendants have not yet been generated and searched.

DE, for dead ends, lists states whose descendants have failed to contain a goal. If these states are encountered again, they will be detected as elements of DE and eliminated from consideration immediately.

In defining the backtrack algorithm for the general case (a graph rather than a tree), it is necessary to detect multiple occurrences of any state so that it will not be reentered and cause (infinite) loops in the path. This is accomplished by testing each newly generated state for membership in any of these three lists. If a new state belongs to any of these lists, then it has already been visited and may be ignored.

```
function backtrack;
begin
  SL := [Start];  NSL := [Start];  DE := [ ];  CS := Start;                    % initialize:
  while NSL ≠ [ ] do                                  % while there are states to be tried
    begin
      if CS = goal (or meets goal description)
        then return SL;                          % on success, return list of states in path.
      if CS has no children (excluding nodes already on DE, SL, and NSL)
        then begin
          while SL is not empty and CS = the first element of SL do
            begin
              add CS to DE;                                % record state as dead end
              remove first element from SL;                                %backtrack
              remove first element from NSL;
              CS := first element of NSL;
            end
          add CS to SL;
        end
        else begin
          place children of CS (except nodes already on DE, SL, or NSL) on NSL;
          CS := first element of NSL;
          add CS to SL
        end
    end;
    return FAIL;
end.
```

In backtrack, the state currently under consideration is called CS for current state. CS is always equal to the state most recently added to SL and represents the "frontier" of the solution path currently being explored. Inference rules, moves in a game, or other appropriate problem-solving operators are ordered and applied to CS. The result is an ordered set of new states, the children of CS. The first of these children is made the new current state and the rest are placed in order on NSL for future examination. The new current state is added to SL and search continues. If CS has no children, it is removed from SL (this is where the algorithm "backtracks") and any remaining children of its predecessor on SL are examined.

A trace of backtrack on the graph of Figure 3.14 is given by:

Initialize: SL = [A]; NSL = [A]; DE = [ ]; CS = A;

AFTER ITERATION	CS	SL	NSL	DE
0	A	[A]	[A]	[ ]
1	B	[B A]	[B C D A]	[ ]
2	E	[E B A]	[E F B C D A]	[ ]
3	H	[H E B A]	[H I E F B C D A]	[ ]
4	I	[I E B A]	[I E F B C D A]	[H]
5	F	[F B A]	[F B C D A]	[E I H]
6	J	[J F B A]	[J F B C D A]	[E I H]
7	C	[C A]	[C D A]	[B F J E I H]
8	G	[G C A]	[G C D A]	[B F J E I H]

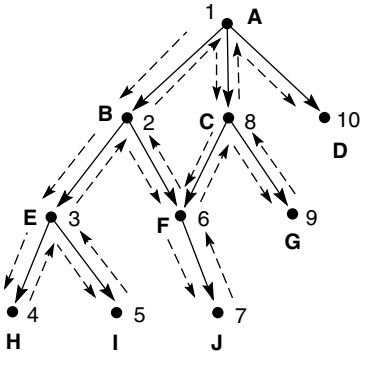


Figure 3.14 Backtracking search of a hypothetical state space.

As presented here, backtrack implements data-driven search, taking the root as a start state and evaluating its children to search for the goal. The algorithm can be viewed as a goal-driven search by letting the goal be the root of the graph and evaluating descendants back in an attempt to find a start state. If the goal description is of type 2 (see Section 3.1.3), the algorithm must determine a goal state by examining the path on SL.

backtrack is an algorithm for searching state space graphs. The graph search algorithms that follow, including depth-first, breadth-first, and best-first search, exploit the ideas used in backtrack, including:

1. The use of a list of unprocessed states (NSL) to allow the algorithm to return to any of these states.
2. A list of "bad" states (DE) to prevent the algorithm from retrying useless paths.
3. A list of nodes (SL) on the current solution path that is returned if a goal is found.
4. Explicit checks for membership of new states in these lists to prevent looping.

The next section introduces search algorithms that, like backtrack, use lists to keep track of states in a search space. These algorithms, including *depth-first*, *breadth-first*, and *best-first* (Chapter 4) search, differ from backtrack in providing a more flexible basis for implementing alternative graph search strategies.

3.2.3 Depth-First and Breadth-First Search

In addition to specifying a search direction (data-driven or goal-driven), a search algorithm must determine the order in which states are examined in the tree or the graph. This section considers two possibilities for the order in which the nodes of the graph are considered: *depth-first* and *breadth-first* search.

Consider the graph represented in Figure 3.15. States are labeled (A, B, C, . . .) so that they can be referred to in the discussion that follows. In depth-first search, when a state is examined, all of its children and their descendants are examined before any of its siblings. Depth-first search goes deeper into the search space whenever this is possible. Only when no further descendants of a state can be found are its siblings considered. Depth-first search examines the states in the graph of Figure 3.15 in the order A, B, E, K, S, L, T, F, M, C, G, N, H, O, P, U, D, I, Q, J, R. The backtrack algorithm of Section 3.2.2 implemented depth-first search.

Breadth-first search, in contrast, explores the space in a level-by-level fashion. Only when there are no more states to be explored at a given level does the algorithm move on to the next deeper level. A breadth-first search of the graph of Figure 3.15 considers the states in the order A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U.

We implement breadth-first search using lists, open and closed, to keep track of progress through the state space. open, like NSL in backtrack, lists states that have been generated but whose children have not been examined. The order in which states are removed from open determines the order of the search. closed records states already examined. closed is the union of the DE and SL lists of the backtrack algorithm.

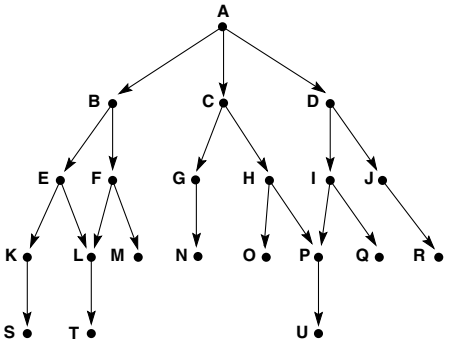


Figure 3.15 Graph for breadth- and depth-first search examples.

```
function breadth_first_search;

begin
   open := [Start];                                                    % initialize
   closed := [ ];
   while open ≠ [ ] do                                                 % states remain
      begin
         remove leftmost state from open, call it X;
            if X is a goal then return SUCCESS                        % goal found
               else begin
                  generate children of X;
                  put X on closed;
                  discard children of X if already on open or closed;  % loop check
                  put remaining children on right end of open          % queue
               end
      end
   return FAIL                                                         % no states left
end.
```

Child states are generated by inference rules, legal moves of a game, or other state transition operators. Each iteration produces all children of the state X and adds them to open. Note that open is maintained as a *queue*, or first-in-first-out (FIFO) data structure. States are added to the right of the list and removed from the left. This biases search toward the states that have been on open the longest, causing the search to be breadth-first. Child states that have already been discovered (already appear on either open or closed) are discarded. If the algorithm terminates because the condition of the "while" loop is no longer satisfied (open = [ ]) then it has searched the entire graph without finding the desired goal: the search has failed.

A trace of breadth_first_search on the graph of Figure 3.15 follows. Each successive number, 2,3,4, . . . , represents an iteration of the "while" loop. U is the goal state.

1. open = [A]; closed = [ ]
2. open = [B,C,D]; closed = [A]
3. open = [C,D,E,F]; closed = [B,A]
4. open = [D,E,F,G,H]; closed = [C,B,A]
5. open = [E,F,G,H,I,J]; closed = [D,C,B,A]
6. open = [F,G,H,I,J,K,L]; closed = [E,D,C,B,A]
7. open = [G,H,I,J,K,L,M] (as L is already on open); closed = [F,E,D,C,B,A]
8. open = [H,I,J,K,L,M,N]; closed = [G,F,E,D,C,B,A]
9. and so on until either U is found or open = [ ].

Figure 3.16 illustrates the graph of Figure 3.15 after six iterations of breadth_first_search. The states on open and closed are highlighted. States not shaded have not been discovered by the algorithm. Note that open records the states on the "frontier" of the search at any stage and that closed records states already visited.

Because breadth-first search considers every node at each level of the graph before going deeper into the space, all states are first reached along the shortest path from the start state. Breadth-first search is therefore guaranteed to find the shortest path from the start state to the goal. Furthermore, because all states are first found along the shortest path, any states encountered a second time are found along a path of equal or greater length. Because there is no chance that duplicate states were found along a better path, the algorithm simply discards any duplicate states.

It is often useful to keep other information on open and closed besides the names of the states. For example, note that breadth_first_search does not maintain a list of states on the current path to a goal as backtrack did on the list SL; all visited states are kept on closed. If a solution path is required, it can not be returned by this algorithm. The solution can be found by storing ancestor information along with each state. A state may be saved along with a record of its parent state, e.g., as a (state, parent) pair. If this is done in the search of Figure 3.15, the contents of open and closed at the fourth iteration would be:

open = [(D,A), (E,B), (F,B), (G,C), (H,C)]; closed = [(C,A), (B,A), (A,nil)]

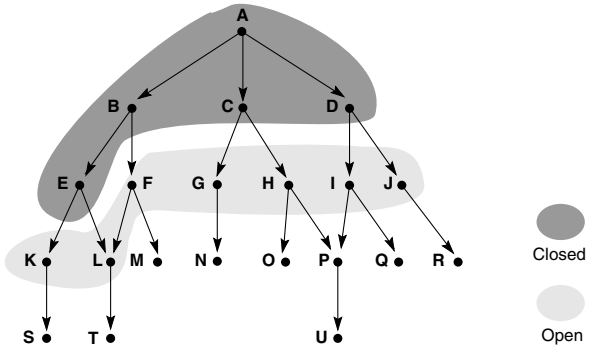


Figure 3.16 Graph of Figure 3.15 at iteration 6 of breadth-first search. States on open and closed are highlighted.

The path (A, B, F) that led from A to F could easily be constructed from this information. When a goal is found, the algorithm can construct the solution path by tracing back along parents from the goal to the start state. Note that state A has a parent of nil, indicating that it is a start state; this stops reconstruction of the path. Because breadth-first search finds each state along the shortest path and retains the first version of each state, this is the shortest path from a start to a goal.

Figure 3.17 shows the states removed from open and examined in a breadth-first search of the graph of the 8-puzzle. As before, arcs correspond to moves of the blank up, to the right, down, and to the left. The number next to each state indicates the order in which it was removed from open. States left on open when the algorithm halted are not shown.

Next, we create a depth-first search algorithm, a simplification of the backtrack algorithm already presented in Section 3.2.3. In this algorithm, the descendant states are added and removed from the *left* end of open: open is maintained as a *stack*, or last-in-first-out (LIFO) structure. The organization of open as a stack directs search toward the most recently generated states, producing a depth-first search order:

```
function depth_first_search;

begin
   open := [Start];                                                    % initialize
   closed := [ ];
   while open ≠ [ ] do                                                 % states remain
      begin
         remove leftmost state from open, call it X;
         if X is a goal then return SUCCESS                            % goal found
            else begin
               generate children of X;
               put X on closed;
               discard children of X if already on open or closed;     % loop check
               put remaining children on left end of open              % stack
            end
      end;
   return FAIL                                                         % no states left
end.
```

A trace of depth_first_search on the graph of Figure 3.15 appears below. Each successive iteration of the "while" loop is indicated by a single line (2, 3, 4, . . .). The initial states of open and closed are given on line 1. Assume U is the goal state.

1. open = [A]; closed = [ ]
2. open = [B,C,D]; closed = [A]
3. open = [E,F,C,D]; closed = [B,A]
4. open = [K,L,F,C,D]; closed = [E,B,A]
5. open = [S,L,F,C,D]; closed = [K,E,B,A]
6. open = [L,F,C,D]; closed = [S,K,E,B,A]
7. open = [T,F,C,D]; closed = [L,S,K,E,B,A]

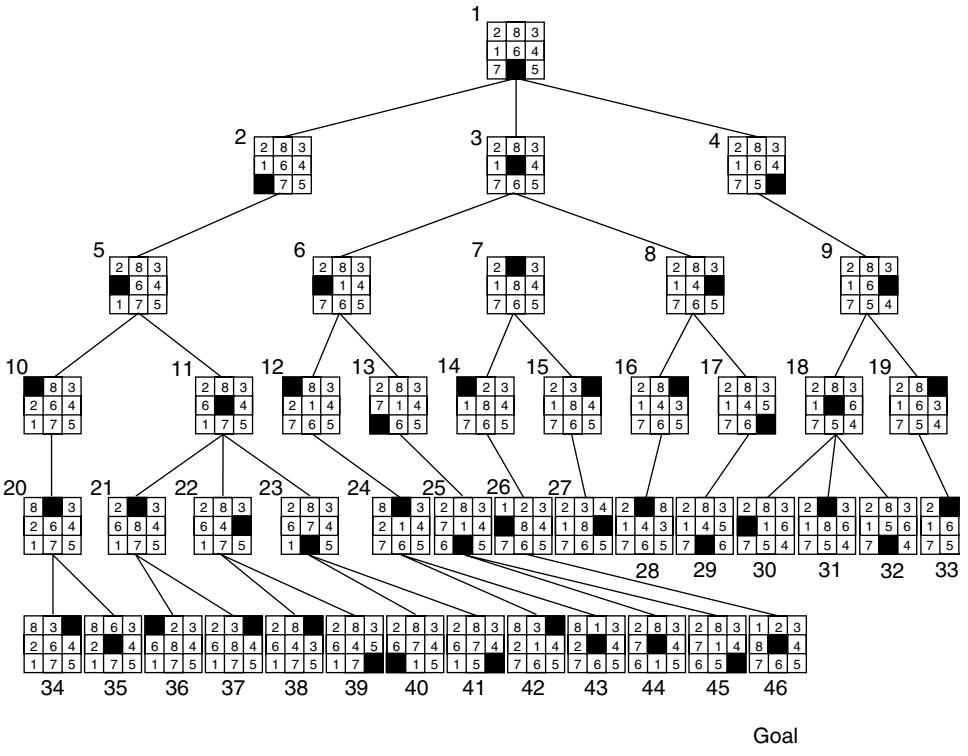


Figure 3.17 Breadth-first search of the 8-puzzle, showing order in which states were removed from open.

8. open = [F,C,D]; closed = [T,L,S,K,E,B,A]
9. open = [M,C,D], (as L is already on closed); closed = [F,T,L,S,K,E,B,A]
10. open = [C,D]; closed = [M,F,T,L,S,K,E,B,A]
11. open = [G,H,D]; closed = [C,M,F,T,L,S,K,E,B,A]

and so on until either U is discovered or open = [ ].

As with breadth_first_search, open lists all states discovered but not yet evaluated (the current "frontier" of the search), and closed records states already considered. Figure 3.18 shows the graph of Figure 3.15 at the sixth iteration of the depth_first_search. The contents of open and closed are highlighted. As with breadth_first_search, the algorithm could store a record of the parent along with each state, allowing the algorithm to reconstruct the path that led from the start state to a goal.

Unlike breadth-first search, a depth-first search is not guaranteed to find the shortest path to a state the first time that state is encountered. Later in the search, a different path may be found to any state. If path length matters in a problem solver, when the algorithm encounters a duplicate state, the algorithm should save the version reached along the shortest path. This could be done by storing each state as a triple: (state, parent,

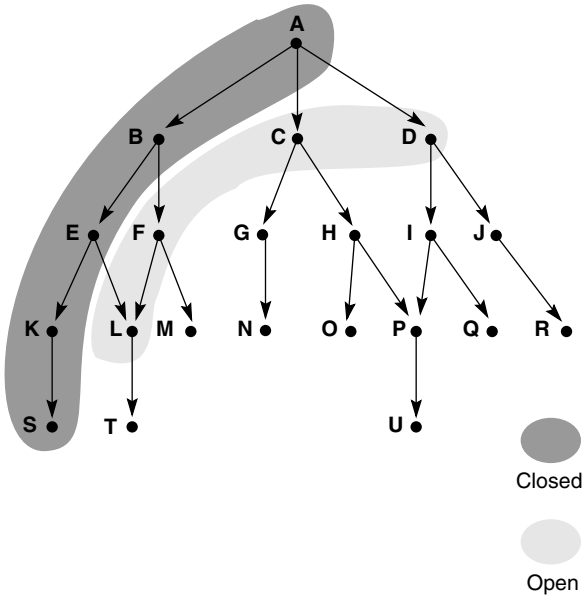


Figure 3.18 Graph of Figure 3.15 at iteration 6 of depth-first search. States on open and closed are highlighted.

length_of_path). When children are generated, the value of the path length is simply incremented by one and saved with the child. If a child is reached along multiple paths, this information can be used to retain the best version. This is treated in more detail in the discussion of *algorithm A* in Chapter 4. Note that retaining the best version of a state in a simple depth-first search does not guarantee that a goal will be reached along the shortest path.

Figure 3.19 gives a depth-first search of the 8-puzzle. As noted previously, the space is generated by the four "move blank" rules (up, down, left, and right). The numbers next to the states indicate the order in which they were considered, i.e., removed from open. States left on open when the goal is found are not shown. A depth bound of 5 was imposed on this search to keep it from getting lost deep in the space.

As with choosing between data- and goal-driven search for evaluating a graph, the choice of depth-first or breadth-first search depends on the specific problem being solved. Significant features include the importance of finding the shortest path to a goal, the branching factor of the space, the available compute time and space resources, the average length of paths to a goal node, and whether we want all solutions or only the first solution. In making these decisions, there are advantages and disadvantages for each approach.

Breadth-First Because it always examines all the nodes at level n before proceeding to level n + 1, breadth-first search always finds the shortest path to a goal node. In a problem where it is known that a simple solution exists, this solution will be found. Unfortunately, if there is a bad branching factor, i.e., states have a high average number of children, the

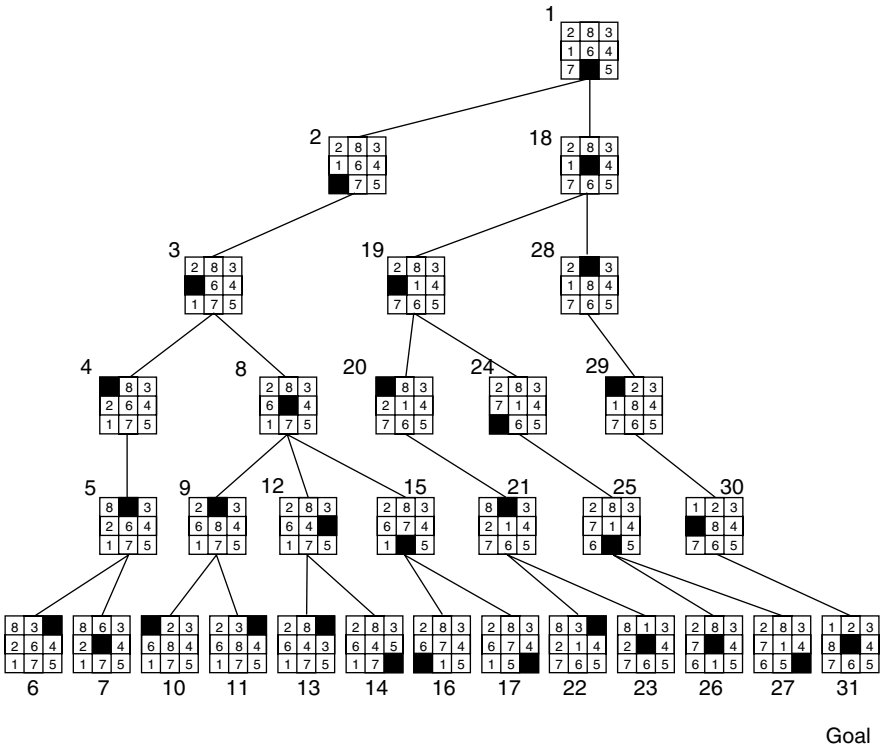


Figure 3.19 Depth-first search of the 8-puzzle with a depth bound of 5.

combinatorial explosion may prevent the algorithm from finding a solution using available memory. This is due to the fact that all unexpanded nodes for each level of the search must be kept on open. For deep searches, or state spaces with a high branching factor, this can become quite cumbersome.

The space utilization of breadth-first search, measured in terms of the number of states on open, is an exponential function of the length of the path at any time. If each state has an average of B children, the number of states on a given level is B times the number of states on the previous level. This gives B^n states on level n. Breadth-first search would place all of these on open when it begins examining level n. This can be prohibitive if solution paths are long, in the game of chess, for example.

Depth-First Depth-first search gets quickly into a deep search space. If it is known that the solution path will be long, depth-first search will not waste time searching a large number of "shallow" states in the graph. On the other hand, depth-first search can get "lost" deep in a graph, missing shorter paths to a goal or even becoming stuck in an infinitely long path that does not lead to a goal.

Depth-first search is much more efficient for search spaces with many branches because it does not have to keep all the nodes at a given level on the open list. The space

usage of depth-first search is a linear function of the length of the path. At each level, open retains only the children of a single state. If a graph has an average of B children per state, this requires a total space usage of $B \times n$ states to go n levels deep into the space.

The best answer to the "depth-first versus breadth-first" issue is to examine the problem space and consult experts in the area. In chess, for example, breadth-first search simply is not possible. In simpler games, breadth-first search not only may be possible but, because it gives the shortest path, may be the only way to avoid losing.

3.2.4 Depth-First Search with Iterative Deepening

A nice compromise on these trade-offs is to use a depth bound on depth-first search. The depth bound forces a failure on a search path once it gets below a certain level. This causes a breadth-first like sweep of the search space at that depth level. When it is known that a solution lies within a certain depth or when time constraints, such as those that occur in an extremely large space like chess, limit the number of states that can be considered; then a depth-first search with a depth bound may be most appropriate. Figure 3.19 showed a depth-first search of the 8-puzzle in which a depth bound of 5 caused the sweep across the space at that depth.

This insight leads to a search algorithm that remedies many of the drawbacks of both depth-first and breadth-first search. *Depth-first iterative deepening* (Korf 1987) performs a depth-first search of the space with a depth bound of 1. If it fails to find a goal, it performs another depth-first search with a depth bound of 2. This continues, increasing the depth bound by one each time. At each iteration, the algorithm performs a complete depth-first search to the current depth bound. No information about the state space is retained between iterations.

Because the algorithm searches the space in a level-by-level fashion, it is guaranteed to find a shortest path to a goal. Because it does only depth-first search at each iteration, the space usage at any level n is $B \times n$, where B is the average number of children of a node.

Interestingly, although it seems as if depth-first iterative deepening would be much less efficient than either depth-first or breadth-first search, its time complexity is actually of the same order of magnitude as either of these: $O(B^n)$. An intuitive explanation for this seeming paradox is given by Korf (1987):

> Since the number of nodes in a given level of the tree grows exponentially with depth, almost all the time is spent in the deepest level, even though shallower levels are generated an arithmetically increasing number of times.

Unfortunately, all the search strategies discussed in this chapter—depth-first, breadth-first, and depth-first iterative deepening—may be shown to have worst-case exponential time complexity. This is true for all *uninformed* search algorithms. The only approaches to search that reduce this complexity employ heuristics to guide search. *Best-first search* is a search algorithm that is similar to the algorithms for depth- and breadth-first search just presented. However, best-first search orders the states on the

open list, the current fringe of the search, according to some measure of their heuristic merit. At each iteration, it considers neither the deepest nor the shallowest but the "best" state. Best-first search is the main topic of Chapter 4.

3.3 Using the State Space to Represent Reasoning with the Propositional and Predicate Calculus

3.3.1 State Space Description of a Logic System

When we defined state space graphs in Section 3.1, we noted that nodes must be distinguishable from one another, with each node representing some state of the solution process. The propositional and predicate calculus can be used as the formal specification language for making these distinctions as well as for mapping the nodes of a graph onto the state space. Furthermore, inference rules can be used to create and describe the arcs between states. In this fashion, problems in the predicate calculus, such as determining whether a particular expression is a logical consequence of a given set of assertions, may be solved using search.

The soundness and completeness of predicate calculus inference rules can guarantee the correctness of conclusions derived through this form of graph-based reasoning. This ability to produce a formal proof of the integrity of a solution through the same algorithm that produces the solution is a unique attribute of much artificial intelligence and theorem proving based problem solving.

Although many problems' states, e.g., tic-tac-toe, can be more naturally described by other data structures, such as arrays, the power and generality of logic allow much of AI problem solving to use the propositional and predicate calculus descriptions and inference rules. Other AI representations such as rules (Chapter 8), semantic networks, or frames (Chapter 7) also employ search strategies similar to those presented in Section 3.2.

EXAMPLE 3.3.1: THE PROPOSITIONAL CALCULUS

The first example of how a set of logic relationships may be viewed as defining a graph is from the propositional calculus. If p, q, r,... are propositions, assume the assertions:

$q \rightarrow p$
$r \rightarrow p$
$v \rightarrow q$
$s \rightarrow r$
$t \rightarrow r$
$s \rightarrow u$
s
t

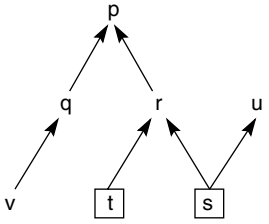


Figure 3.20 State space graph of a set of implications in the propositional calculus.

From this set of assertions and the inference rule modus ponens, certain propositions (p, r, and u) may be inferred; others (such as v and q) may not be so inferred and indeed do not logically follow from these assertions. The relationship between the initial assertions and these inferences is expressed in the directed graph in Figure 3.20.

In Figure 3.20 the arcs correspond to logical implications (→). Propositions that are given as true (s and t) correspond to the given data of the problem. Propositions that are logical consequences of this set of assertions correspond to the nodes that may be reached along a directed path from a state representing a true proposition; such a path corresponds to a sequence of applications of modus ponens. For example, the path [s, r, p] corresponds to the sequence of inferences:

s and s → r yields r.
r and r → p yields p.

Given this representation, determining whether a given proposition is a logical consequence of a set of propositions becomes a problem of finding a path from a boxed node (the start node) to the proposition (the goal node). Thus, the task can be cast as a graph search problem. The search strategy used here is data-driven, because it proceeds from what is known (the true propositions) toward the goal. Alternatively, a goal-directed strategy could be applied to the same state space by starting with the proposition to be proved (the goal) and searching back along arcs to find support for the goal among the true propositions. We can also search this space of inferences in either a depth-first or breadth-first fashion.

3.3.2 And/Or Graphs

In the propositional calculus example of Section 3.3.1, all of the assertions were implications of the form p → q. We did not discuss the way in which the logic operators and and or could be represented in such a graph. Expressing the logic relationships defined by these operators requires an extension to the basic graph model defined in Section 3.1 to what we call the *and/or graph*. And/or graphs are an important tool for describing the search spaces generated by many AI problems, including those solved by logic-based theorem provers and expert systems.

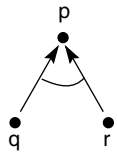


Figure 3.21 And/or graph of the expression $q \wedge r \rightarrow p$.

In expressions of the form $q \wedge r \rightarrow p$, both q and r must be true for p to be true. In expressions of the form $q \vee r \rightarrow p$, the truth of either q or r is sufficient to prove p is true. Because implications containing disjunctive premises may be written as separate implications, this expression is often written as $q \rightarrow p$, $r \rightarrow p$. To represent these different relationships graphically, and/or graphs distinguish between and nodes and or nodes. If the premises of an implication are connected by an $\wedge$ operator, they are called and nodes in the graph and the arcs from this node are joined by a curved link. The expression $q \wedge r \rightarrow p$ is represented by the and/or graph of Figure 3.21.

The link connecting the arcs in Figure 3.21 captures the idea that both q and r must be true to prove p. If the premises are connected by an or operator, they are regarded as or nodes in the graph. Arcs from or nodes to their parent node are not so connected (Figure 3.22). This captures the notion that the truth of any one of the premises is independently sufficient to determine the truth of the conclusion.

An and/or graph is actually a specialization of a type of graph known as a *hypergraph*, which connects nodes by sets of arcs rather than by single arcs. A hypergraph is defined as follows:

DEFINITION

HYPERGRAPH

A hypergraph consists of:

N, a set of nodes.

H, a set of hyperarcs defined by ordered pairs in which the first element of the pair is a single node from N and the second element is a subset of N.

An ordinary graph is a special case of hypergraph in which all the sets of descendant nodes have a cardinality of 1.

Hyperarcs are also known as *k-connectors*, where k is the cardinality of the set of descendant nodes. If $k = 1$, the descendant is thought of as an or node. If $k > 1$, the elements of the set of descendants may be thought of as and nodes. In this case, the connector is drawn between the individual edges from the parent to each of the descendant nodes; see, for example, the curved link in Figure 3.21.

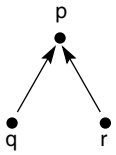


Figure 3.22 And/or graph of the expression $q \vee r \rightarrow p$.

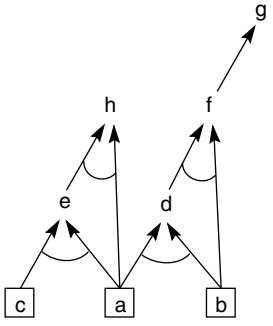


Figure 3.23 And/or graph of a set of propositional calculus expressions.

EXAMPLE 3.3.2: AND/OR GRAPH SEARCH

The second example is also from the propositional calculus but generates a graph that contains both and and or descendants. Assume a situation where the following propositions are true:

a
b
c
$a \wedge b \rightarrow d$
$a \wedge c \rightarrow e$
$b \wedge d \rightarrow f$
$f \rightarrow g$
$a \wedge e \rightarrow h$

This set of assertions generates the and/or graph in Figure 3.23.

Questions that might be asked (answers deduced by the search of this graph) are:

1. Is h true?
2. Is h true if b is no longer true?

3. What is the shortest path (i.e., the shortest sequence of inferences) to show that X (some proposition) is true?

4. Show that the proposition p (note that p is not supported) is false. What does this mean? What would be necessary to achieve this conclusion?

And/or graph search requires only slightly more record keeping than search in regular graphs, an example of which was the backtrack algorithm, previously discussed. The or descendants are checked as they were in backtrack: once a path is found connecting a goal to a start node along or nodes, the problem will be solved. If a path leads to a failure, the algorithm may backtrack and try another branch. In searching and nodes, however, all of the and descendants of a node must be solved (or proved true) to solve the parent node.

In the example of Figure 3.23, a goal-directed strategy for determining the truth of h first attempts to prove both a and e. The truth of a is immediate, but the truth of e requires the truth of both c and a; these are given as true. Once the problem solver has traced all these arcs down to true propositions, the true values are recombined at the and nodes to verify the truth of h.

A data-directed strategy for determining the truth of h, on the other hand, begins with the known facts (c, a, and b) and begins adding new propositions to this set of known facts according to the constraints of the and/or graph. e or d might be the first proposition added to the set of facts. These additions make it possible to infer new facts. This process continues until the desired goal, h, has been proved.

One way of looking at and/or graph search is that the $\wedge$ operator (hence the and nodes of the graph) indicates a problem decomposition in which the problem is broken into subproblems such that all of the subproblems must be solved to solve the original problem. An $\vee$ operator in the predicate calculus representation of the problem indicates a selection, a point in the problem solution at which a choice may be made between alternative problem-solving paths or strategies, any one of which, if successful, is sufficient to solve the problem.

3.3.3 Further Examples and Applications

EXAMPLE 3.3.3: MACSYMA

One natural example of an and/or graph is a program for symbolically integrating mathematical functions. MACSYMA is a well-known program that is used extensively by mathematicians. The reasoning of MACSYMA can be represented as an and/or graph. In performing integrations, one important class of strategies involves breaking an expression into sub-expressions that may be integrated independently of one another, with the result being combined algebraically into a solution expression. Examples of this strategy include the rule for integration by parts and for decomposing the integral of a sum into the sum of the integrals of the individual terms. These strategies, representing the decomposition of a problem into independent subproblems, can be represented by and nodes in the graph.

Another class of strategies involves the simplification of an expression through various algebraic substitutions. Because any given expression may allow a number of

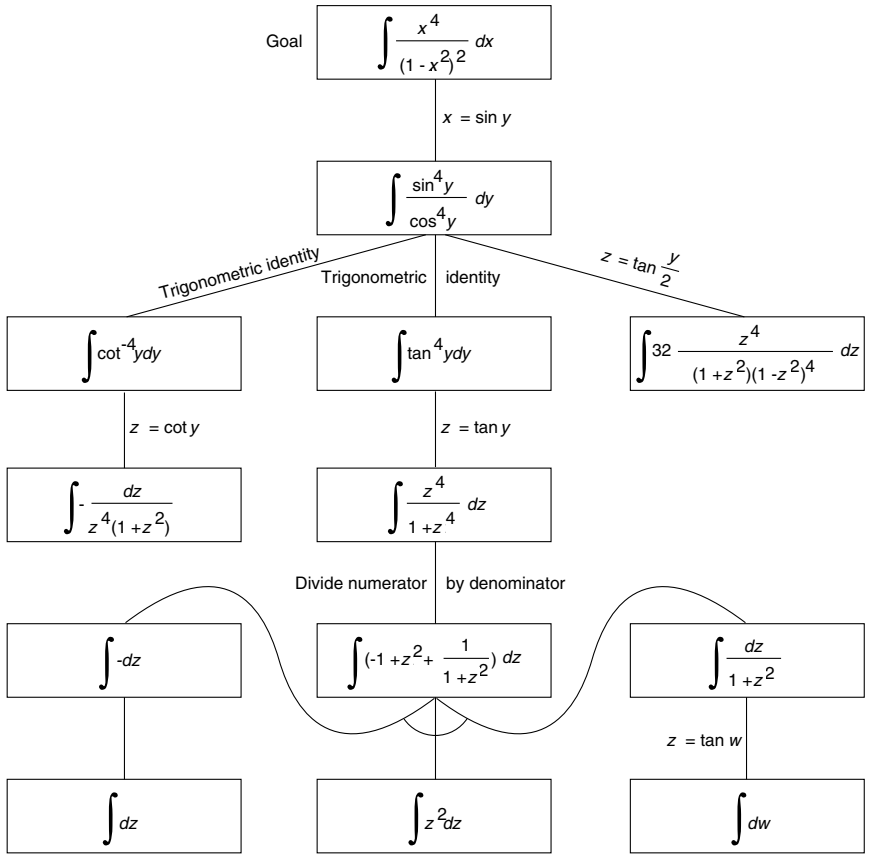


Figure 3.24 And/or graph of part of the state space for integrating a function, from Nilsson (1971).

different substitutions, each representing an independent solution strategy, these strategies are represented by or nodes of the graph. Figure 3.24 illustrates the space searched by such a problem solver. The search of this graph is goal-directed, in that it begins with the query "find the integral of a particular function" and searches back to the algebraic expressions that define that integral. Note that this is an example in which goal-directed search is the obvious strategy. It would be practically impossible for a problem solver to determine the algebraic expressions that formed the desired integral without working back from the query.

EXAMPLE 3.3.4: GOAL-DRIVEN AND/OR SEARCH

This example is taken from the predicate calculus and represents a goal-driven graph search where the goal to be proved true in this situation is a predicate calculus expression containing variables. The axioms are the logical descriptions of a relationship between a

dog, Fred, and his master, Sam. We assume that a cold day is not a warm day, bypassing issues such as the complexity added by equivalent expressions for predicates, an issue discussed further in Chapters 7 and 14. The facts and rules of this example are given as English sentences followed by their predicate calculus equivalents:

1. Fred is a collie.
   collie(fred).
2. Sam is Fred's master.
   master(fred,sam).
3. The day is Saturday.
   day(saturday).
4. It is cold on Saturday.
   ¬ (warm(saturday)).
5. Fred is trained.
   trained(fred).
6. Spaniels are good dogs and so are trained collies.
   ∀ X[spaniel(X) ∨ (collie(X) ∧ trained(X)) → gooddog(X)]
7. If a dog is a good dog and has a master then he will be with his master.
   ∀ (X,Y,Z) [gooddog(X) ∧ master(X,Y) ∧ location(Y,Z) → location(X,Z)]
8. If it is Saturday and warm, then Sam is at the park.
   (day(saturday) ∧ warm(saturday)) → location(sam,park).
9. If it is Saturday and not warm, then Sam is at the museum.
   (day(saturday) ∧ ¬ (warm(saturday))) → location(sam,museum).

The goal is the expression ∃ X location(fred,X), meaning "where is fred?" A backward search algorithm examines alternative means of establishing this goal: if fred is a good dog and fred has a master and fred's master is at a location then fred is at that location also. The premises of this rule are then examined: what does it mean to be a gooddog, etc.? This process continues, constructing the and/or graph of Figure 3.25.

Let us examine this search in more detail, particularly because it is an example of goal-driven search using the predicate calculus and it illustrates the role of unification in the generation of the search space. The problem to be solved is "where is fred?" More formally, it may be seen as determining a substitution for the variable X, if such a substitution exists, under which location(fred,X) is a logical consequence of the initial assertions.

When it is desired to determine Fred's location, clauses are examined that have location as their conclusion, the first being clause 7. This conclusion, location(X,Z), is then unified with location(fred, X) by the substitutions {fred/X, X/Z}. The premises of this rule, under the same substitution set, form the and descendants of the top goal:

gooddog(fred) ∧ master(fred,Y) ∧ location(Y,X).

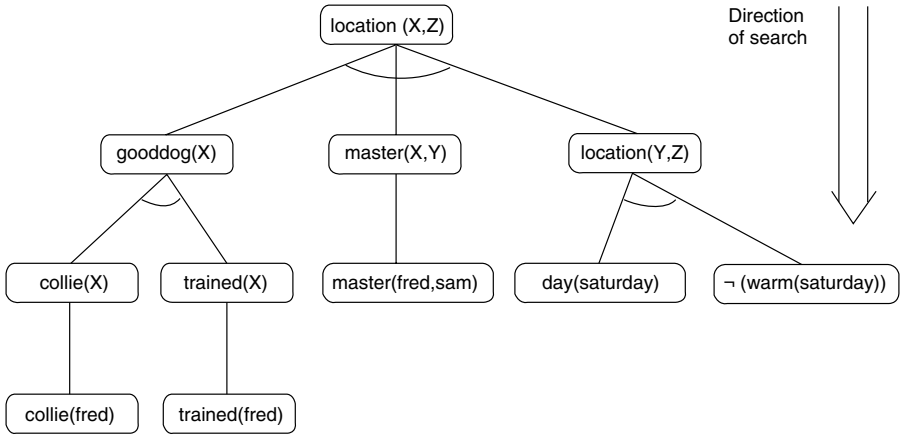


Figure 3.25 The solution subgraph showing that fred is at the museum.

This expression may be interpreted as meaning that one way to find Fred is to see if Fred is a good dog, find out who Fred's master is, and find out where the master is. The initial goal has thus been replaced by three subgoals. These are and nodes and all of them must be solved.

To solve these subgoals, the problem solver first determines whether Fred is a gooddog. This matches the conclusion of clause 6 using the substitution {fred/X}. The premise of clause 6 is the or of two expressions:

spaniel(fred) $\vee$ (collie(fred) $\wedge$ trained(fred))

The first of these or nodes is spaniel(fred). The database does not contain this assertion, so the problem solver must assume it is false. The other or node is (collie(fred) $\wedge$ trained(fred)), i.e., is Fred a collie and is Fred trained. Both of these need to be true, which they are by clauses 1 and 5.

This proves that gooddog(fred) is true. The problem solver then examines the second of the premises of clause 7: master(X,Y). Under the substitution {fred/X}, master(X,Y) becomes master(fred,Y), which unifies with the fact (clause 2) of master(fred,sam). This produces the unifying substitution of {sam/Y}, which also gives the value of sam to the third subgoal of clause 7, creating the new goal location(sam,X).

In solving this, assuming the problem solver tries rules in order, the goal location(sam,X) will first unify with the conclusion of clause 7. Note that the same rule is being tried with different bindings for X. Recall (Chapter 2) that X is a "dummy" variable and could have any name (any string beginning with an uppercase letter). Because the extent of the meaning of any variable name is contained within the clause in which it

appears, the predicate calculus has no global variables. Another way of saying this is that values of variables are passed to other clauses as parameters and have no fixed (memory) locations. Thus, the multiple occurrences of X in different rules in this example indicate *different* formal parameters (Section 14.3).

In attempting to solve the premises of rule 7 with these new bindings, the problem solver will fail because sam is not a gooddog. Here, the search will backtrack to the goal location(sam,X) and try the next match, the conclusion of rule 8. This will also fail, which will cause another backtrack and a unification with the conclusion of clause 9, at(sam,museum).

Because the premises of clause 9 are supported in the set of assertions (clauses 3 and 4), it follows that the conclusion of 9 is true. This final unification goes all the way back up the tree to finally answer ∃ X location(fred,X) with location(fred, museum).

It is important to examine carefully the nature of the goal-driven search of a graph and compare it with the data-driven search of Example 3.3.2. Further discussion of this issue, including a more rigorous comparison of these two methods of searching a graph, continues in the next example, but is seen in full detail only in the discussion of production systems in Chapter 6 and in the application to expert systems in Part IV. Another point implicit in this example is that the order of clauses affects the order of search. In the example above, the multiple location clauses were tried in order, with backtracking search eliminating those that failed to be proved true.

EXAMPLE 3.3.5: THE FINANCIAL ADVISOR REVISITED

In the last example of Chapter 2 we used predicate calculus to represent a set of rules for giving investment advice. In that example, modus ponens was used to infer a proper investment for a particular individual. We did not discuss the way in which a program might determine the appropriate inferences. This is, of course, a search problem; the present example illustrates one approach to implementing the logic-based financial advisor, using goal-directed, depth-first search with backtracking. The discussion uses the predicates found in Section 2.4; these predicates are not duplicated here.

Assume that the individual has two dependents, $20,000 in savings, and a steady income of $30,000. As discussed in Chapter 2, we can add predicate calculus expressions describing these facts to the set of predicate calculus expressions. Alternatively, the program may begin the search without this information and ask the user to add it as needed. This has the advantage of not requiring data that may not prove necessary for a solution. This approach, often taken with expert systems, is illustrated here.

In performing a consultation, the goal is to find an investment; this is represented with the predicate calculus expression ∃ X investment(X), where X in the goal variable we seek to bind. There are three rules (1, 2, and 3) that conclude about investments, because the query will unify with the conclusion of these rules. If we select rule 1 for initial exploration, its premise savings_account(inadequate) becomes the subgoal, i.e., the child node that will be expanded next.

In generating the children of savings_account(inadequate), the only rule that may be applied is rule 5. This produces the and node:

amount_saved(X) ∧ dependents(Y) ∧ ¬ greater(X,minsavings(Y)).

If we attempt to satisfy these in left-to-right order, amount_saved(X) is taken as the first subgoal. Because the system contains no rules that conclude this subgoal, it will query the user. When amount_saved(20000) is added the first subgoal will succeed, with unification substituting 20000 for X. Note that because an and node is being searched, a failure here would eliminate the need to examine the remainder of the expression.

Similarly, the subgoal dependents(Y) leads to a user query, and the response, dependents(2), is added to the logical description. The subgoal matches this expression with the substitution {2/Y}. With these substitutions, the search next evaluates the truth of

¬ greater(20000, minsavings(2)).

This evaluates to false, causing failure of the entire and node. The search then backtracks to the parent node, savings_account(inadequate), and attempts to find an alternative way to prove that node true. This corresponds to the generation of the next child in the search. Because no other rules conclude this subgoal, search fails back to the top-level goal, investment(X). The next rule whose conclusions unify with this goal is rule 2, producing the new subgoals

savings_account(adequate) ∧ income(adequate).

Continuing the search, savings_account(adequate) is proved true as the conclusion of rule 4, and income(adequate) follows as the conclusion of rule 6. Although the details of the remainder of the search will be left to the reader, the and/or graph that is ultimately explored appears in Figure 3.26.

EXAMPLE 3.3.6: AN ENGLISH LANGUAGE PARSER AND SENTENCE GENERATOR

Our final example is not from the predicate calculus but consists of a set of rewrite rules for parsing sentences in a subset of English grammar. Rewrite rules take an expression and transform it into another by replacing the pattern on one side of the arrow (↔) with the pattern on the other side. For example, a set of rewrite rules could be defined to change an expression in one language, such as English, into another language (perhaps French or a predicate calculus clause). The rewrite rules given here transform a subset of English sentences into higher level grammatical constructs such as noun phrase, verb phrase, and sentence. These rules are used to *parse* sequences of words, i.e., to determine whether they are well-formed sentences (whether they are grammatically correct or not) and to model the linguistic structure of the sentences.

Five rules for a simple subset of English grammar are:

1. sentence ↔ np vp
   (A sentence is a noun phrase followed by a verb phrase.
2. np ↔ n
   (A noun phrase is a noun.)

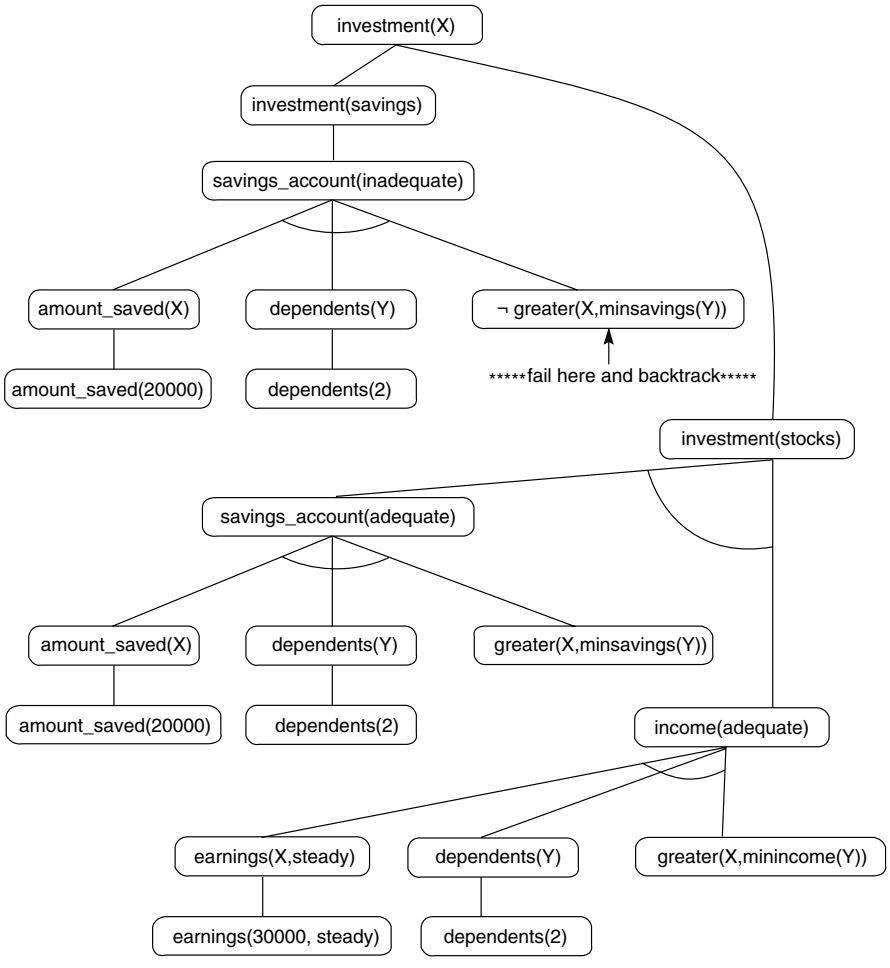


Figure 3.26 And/or graph searched by the financial advisor.

3. np ↔ art n
   (A noun phrase is an article followed by a noun.)

4. vp ↔ v
   (A verb phrase is a verb.)

5. vp ↔ v np
   (A verb phrase is a verb followed by a noun phrase.)

In addition to these grammar rules, a parser needs a dictionary of words in the language. These words are called the *terminals* of the grammar. They are defined by their parts of speech using rewrite rules. In the following dictionary, "a," "the," "man," "dog," "likes," and "bites" are the terminals of our very simple grammar:

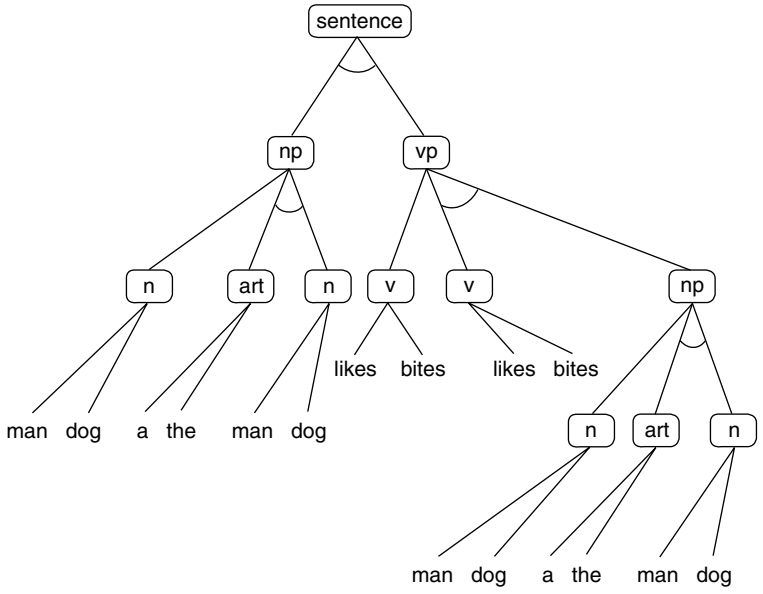


Figure 3.27 And/or graph for the grammar of Example 3.3.6. Some of the nodes (np, art, etc.) have been written more than once to simplify drawing the graph.

6. art ↔ a
7. art ↔ the
   ("a" and "the" are articles)
8. n ↔ man
9. n ↔ dog
   ("man" and "dog" are nouns)
10. v ↔ likes
11. v ↔ bites
    ("likes" and "bites" are verbs)

These rewrite rules define the and/or graph of Figure 3.27, where sentence is the root. The elements on the left of a rule correspond to and nodes in the graph. Multiple rules with the same conclusion form the or nodes. Notice that the leaf or terminal nodes of this graph are the English words in the grammar (hence, they are called *terminals*).

An expression is *well formed* in a grammar if it consists entirely of terminal symbols and there is a series of substitutions in the expression using rewrite rules that reduce it to the sentence symbol. Alternatively, this may be seen as constructing a *parse tree* that has the words of the expression as its leaves and the sentence symbol as its root.

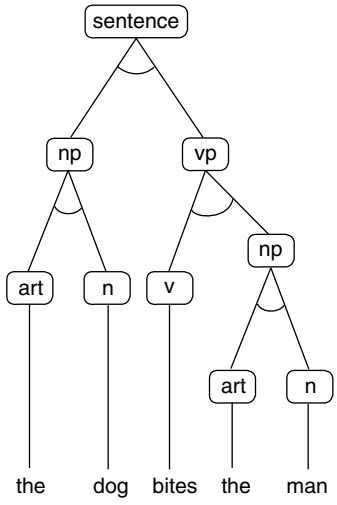


Figure 3.28 Parse tree for the sentence The dog bites the man.
Note that this is a subtree of the graph of Figure 3.27.

For example, we may parse the sentence the dog bites the man, constructing the parse tree of Figure 3.28. This tree is a subtree of the and/or graph of Figure 3.27 and is constructed by searching this graph. A *data-driven parsing* algorithm would implement this by matching right-hand sides of rewrite rules with patterns in the sentence, trying these matches in the order in which the rules are written. Once a match is found, the part of the expression matching the right-hand side of the rule is replaced by the pattern on the left-hand side. This continues until the sentence is reduced to the symbol sentence (indicating a successful parse) or no more rules can be applied (indicating failure). A trace of the parse of the dog bites the man:

1. The first rule that will match is 7, rewriting the as art. This yields: art dog bites the man.
2. The next iteration would find a match for 7, yielding art dog bites art man.
3. Rule 8 will fire, producing art dog bites art n.
4. Rule 3 will fire to yield art dog bites np.
5. Rule 9 produces art n bites np.
6. Rule 3 may be applied, giving np bites np.
7. Rule 11 yields np v np.
8. Rule 5 yields np vp.
9. Rule 1 reduces this to sentence, accepting the expression as correct.

The above example implements a data-directed depth-first parse, as it always applies the highest-level rule to the expression; e.g., art n reduces to np before bites reduces to v. Parsing could also be done in a goal-directed fashion, taking sentence as the starting string and finding a series of replacements of patterns that match left-hand sides of rules leading to a series of terminals that match the target sentence.

Parsing is important, not only for natural language (Chapter 15) but also for constructing compilers and interpreters for computer languages (Aho and Ullman 1977). The literature is full of parsing algorithms for all classes of languages. For example, many goal-directed parsing algorithms look ahead in the input stream to determine which rule to apply next.

In this example we have taken a very simple approach of searching the and/or graph in an uninformed fashion. One thing that is interesting in this example is the implementation of the search. This approach of keeping a record of the current expression and trying to match the rules in order is an example of using a *production system* to implement search. This is a major topic of Chapter 6.

Rewrite rules are also used to generate legal sentences according to the specifications of the grammar. Sentences may be generated by a goal-driven search, beginning with sentence as the top-level goal and ending when no more rules can be applied. This produces a string of terminal symbols that is a legal sentence in the grammar. For example:

A sentence is a np followed by a vp (rule 1).

np is replaced by n (rule 2), giving n vp.

man is the first n available (rule 8), giving man vp.

Now np is satisfied and vp is attempted. Rule 3 replaces vp with v, man v.

Rule 10 replaces v with likes.

man likes is found as the first acceptable sentence.

If it is desired to create all acceptable sentences, this search may be systematically repeated until all possibilities are tried and the entire state space has been searched exhaustively. This generates sentences including a man likes, the man likes, and so on. There are about 80 correct sentences that are produced by an exhaustive search. These include such semantic anomalies as the man bites the dog.

Parsing and generation can be used together in a variety of ways to handle different problems. For instance, if it is desired to find all sentences to complete the string "the man," then the problem solver may be given an incomplete string the man... . It can work upward in a data-driven fashion to produce the goal of completing the sentence rule (rule 1), where np is replaced by the man, and then work in a goal-driven fashion to determine all possible vps that will complete the sentence. This would create sentences such as the man likes, the man bites the man, and so on. Again, this example deals only with syntactic correctness. The issue of semantics (whether the string has a mapping into some "world" with "truth") is entirely different. Chapter 2 examined the issue of constructing a logic-based semantics for expressions; for verifing sentences in natural language, the issue is much more difficult and is discussed in Chapter 15.

In the next chapter we discuss the use of heuristics to focus search on the "best" possible portion of the state space. Chapter 6 discusses the production system and other software "architectures" for controlling state-space search.

3.4 Epilogue and References

Chapter 3 introduced the theoretical foundations of state space search, using graph theory to analyze the structure and complexity of problem-solving strategies. The chapter compared data-driven and goal-driven reasoning and depth-first and breadth-first search. And/or graphs allow us to apply state space search to the implementation of logic-based reasoning.

Basic graph search is discussed in a number of textbooks on computer algorithms. These include *Introduction to Algorithms* by Thomas Cormen, Charles Leiserson, and Ronald Rivest (1990), *Walls and Mirrors* by Paul Helman and Robert Veroff (1986), *Algorithms* by Robert Sedgewick (1983), and *Fundamentals of Computer Algorithms* by Ellis Horowitz and Sartaj Sahni (1978). Finite automata are presented in Lindenmayer and Rosenberg (1976). More algorithms for and/or search are presented in Chapter 14, *Automated Reasoning*.

The use of graph search to model intelligent problem solving is presented in *Human Problem Solving* by Alan Newell and Herbert Simon (1972). Artificial intelligence texts that discuss search strategies include Nils Nilsson's *Artificial Intelligence* (1998), Patrick Winston's *Artificial Intelligence* (1992), and *Artificial Intelligence* by Eugene Charniak and Drew McDermott (1985). *Heuristics* by Judea Pearl (1984) presents search algorithms and lays a groundwork for the material we present in Chapter 4. Developing new techniques for graph search are often topics at the annual AI conferences.

3.5 Exercises

1. A Hamiltonian path is a path that uses every node of the graph exactly once. What conditions are necessary for such a path to exist? Is there such a path in the Königsberg map?

2. Give the graph representation for the farmer, wolf, goat, and cabbage problem:

   > A farmer with his wolf, goat, and cabbage come to the edge of a river they wish to cross. There is a boat at the river's edge, but, of course, only the farmer can row. The boat also can carry only two things (including the rower) at a time. If the wolf is ever left alone with the goat, the wolf will eat the goat; similarly, if the goat is left alone with the cabbage, the goat will eat the cabbage. Devise a sequence of crossings of the river so that all four characters arrive safely on the other side of the river.

   Let nodes represent states of the world; e.g., the farmer and the goat are on the west bank and the wolf and cabbage on the east. Discuss the advantages of breadth-first and depth-first search for this problem.

3. Build a finite state acceptor that recognizes all strings of binary digits a) that contain "111", b) that end in "111", c) that contain "111" but not more that three consecutive "1"s.

4. Give an instance of the traveling salesperson problem for which the nearest-neighbor strategy fails to find an optimal path. Suggest another heuristic for this problem.
5. "Hand run" the backtrack algorithm on the graph in Figure 3.29. Begin from state A. Keep track of the successive values of NSL, SL, CS, etc.
6. Implement a backtrack algorithm in a programming language of your choice.
7. Determine whether goal-driven or data-driven search would be preferable for solving each of the following problems. Justify your answer.
   a. Diagnosing mechanical problems in an automobile.
   b. You have met a person who claims to be your distant cousin, with a common ancestor named John Doe. You would like to verify her claim.
   c. Another person claims to be your distant cousin. He does not know the common ancestor's name but knows that it was no more than eight generations back. You would like to either find this ancestor or determine that she did not exist.
   d. A theorem prover for plane geometry.
   e. A program for examining sonar readings and interpreting them, such as telling a large submarine from a small submarine from a whale from a school of fish.
   f. An expert system that will help a human classify plants by species, genus, etc.
8. Choose and justify a choice of breadth- or depth-first search for examples of Exercise 6.
9. Write a backtrack algorithm for and/or graphs.
10. Trace the goal-driven *good-dog* problem of Example 3.3.4 in a data-driven fashion.
11. Give another example of an and/or graph problem and develop part of the search space.
12. Trace a data-driven execution of the *financial advisor* of Example 3.3.5 for the case of an individual with four dependents, $18,000 in the bank, and a steady income of $25,000 per year. Based on a comparison of this problem and the example in the text, suggest a generally "best" strategy for solving the problem.
13. Add rules for adjectives and adverbs to the *English grammar* of Example 3.3.6.
14. Add rules for (multiple) prepositional phrases to the *English grammar* of Example 3.3.6.
15. Add grammar rules to Example 3.3.6 that allow complex sentences such as,

    sentence ↔ sentence AND sentence.

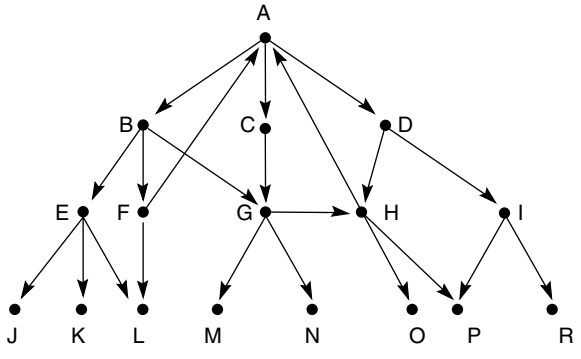


Figure 3.29 A graph to be searched.