3

Generate and Test Means-Ends Analysis and Problem Reduction

In this chapter, you learn about three powerful problem-solving methods: *generate and test*, *means–ends analysis*, and *problem reduction*. You also learn about two new representations, both of which can be viewed as special cases of the semantic-net representation introduced in Chapter 2. One is the *state-space representation*, introduced in the discussion of means–ends analysis, and another is the *goal tree*, introduced in the discussion of problem reduction.

By way of illustration, you see how a program can find the combination to a safe, plan a route from one city to another, and solve motion-planning problems in a world populated by a child's blocks.

Once you have finished this chapter, you will be able to identify and deploy three more problem-solving methods and two more representations, thus adding considerably to your personal representation and method collections. You will also begin to see that you yourself use similar representations and methods daily as you solve problems.

THE GENERATE-AND-TEST METHOD

Problem solvers adhering to the **generate-and-test paradigm** use two basic modules, as illustrated in figure 3.1. One module, the **generator**,

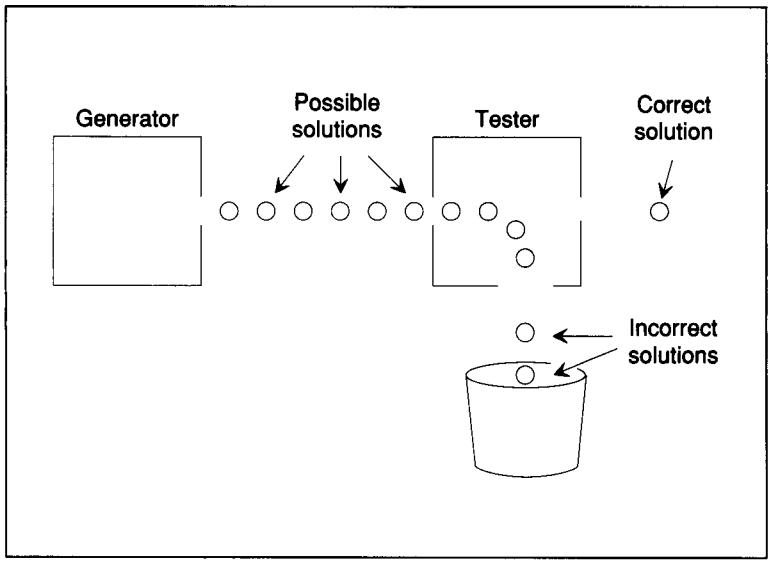


Figure 3.1 The generate-and-test method involves a generator and a tester.

enumerates possible solutions. The second, the **tester**, evaluates each proposed solution, either accepting or rejecting that solution.

The generator may generate all possible solutions before the tester takes over; more commonly, however, generation and testing are interdigitated. Action may stop when one acceptable solution is found, or action may continue until some satisfactory number of solutions is found, or action may continue until all possible solutions are found. Here is the interdigitated, stop-when-acceptable version in procedural English:

To perform generate and test,

- ▷ Until a satisfactory solution is found or no more candidate solutions can be generated,
  - ▷ Generate a candidate solution.
  - ▷ Test the candidate solution.
- ▷ If an acceptable solution is found, announce it; otherwise, announce failure.

In the rest of this section, you learn more about the generate-and-test method, you learn which sort of problems the generate-and-test method solves, and you learn several criteria that good generators always satisfy.

Generate-and-Test Systems Often Do Identification

The generate-and-test paradigm is used most frequently to solve identification problems. In identification problems, the generator is said to produce *hypotheses.*

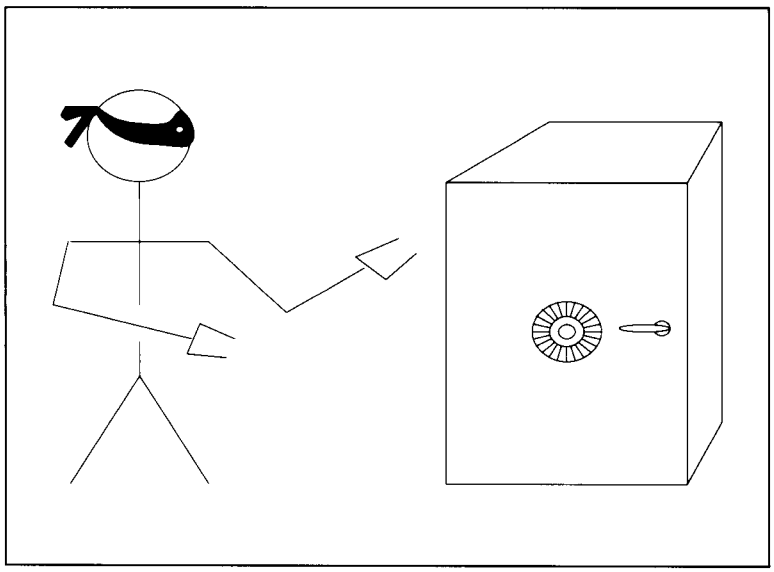


Figure 3.2 Burgling a safe using the generate-and-test paradigm. The generator is the procedure that the burglar uses to select and dial combinations. The tester is the procedure that the burglar uses to work the handle. Careful safecrackers make sure that they try all possibilities, without any repeats, until a twist of the handle opens the safe.

To use the generate-and-test paradigm to identify, say, a tree, you can reach for a tree book, then thumb through it page by page, stopping when you find a picture that looks like the tree to be identified. Thumbing through the book is the generation procedure; matching the pictures to the tree is the testing procedure.

To use generate and test to burgle a three-number, two-digit safe, you can start with the combination 00-00-00, move to 00-00-01, and continue on through all possible combinations until the door opens. Of course, the counting is the generation procedure, and the twist of the safe handle is the testing procedure.

The burglar in figure 3.2 may take some time to crack the safe with this approach, however, for there are $100^3 = 1$ million combinations. At three per minute, figuring that he will have to go through half of the combinations, on average, to succeed, the job will take about 16 weeks, if he works 24 hours per day.

Good Generators Are Complete, Nonredundant, and Informed

It is obvious that good generators have three properties:

- Good generators are complete: They eventually produce all possible solutions.
- Good generators are nonredundant: They never compromise efficiency by proposing the same solution twice.
- Good generators are informed: They use possibility-limiting information, restricting the solutions that they propose accordingly.

Informability is important, because otherwise there are often too many solutions to go through. Consider the tree-identification example. If it is

winter and a tree you are trying to identify is bare, you do not bother going through a tree book's conifer section.

Similarly, if a burglar knows, somehow, that all of the numbers in a safe combination are prime numbers in the range from 0 to 99, then he can confine himself to $25^3 = 15625$ numbers, getting the safe open in less than 2 days, on the average, instead of in 16 weeks.

THE MEANS-ENDS ANALYSIS METHOD

The **state** of a system is a description that is sufficient to determine the future. In a **state space**, each node denotes a **state**, and each link denotes a possible one-step **transition** from one state to another state:

A **state space** is a representation

That is a semantic net

In which

- ▷ The nodes denote states.
- ▷ The links denote transitions between states.

Thus, a state space is a member of the semantic-net family of representations introduced in Chapter 2.

In the context of problem solving, states correspond to where you are or might be in the process of solving a problem. Hence, the **current state** corresponds to where you are, the **goal state** corresponds to where you want to be, and the problem is to find a sequence of transitions that leads from the **initial state** to the goal state.

In the rest of this section, you learn about means–ends analysis, a standard method for selecting transitions. You also learn about one popular way to implement means–ends analysis using a simple table.

The Key Idea in Means-Ends Analysis Is to Reduce Differences

The purpose of **means–ends analysis** is to identify a procedure that causes a transition from the current state to the goal state, or at least to an intermediate state that is closer to the goal state. Thus, the identified procedure reduces the observed **difference** between the current state and the goal state.

Consider the states shown in figure 3.3. Solid-line nodes identify the current state and the goal state. Dotted-line nodes correspond to states that are not yet known to exist. Descriptions of the current state, or of the goal state, or of the difference between those states, may contribute to the identification of a difference-reducing procedure.

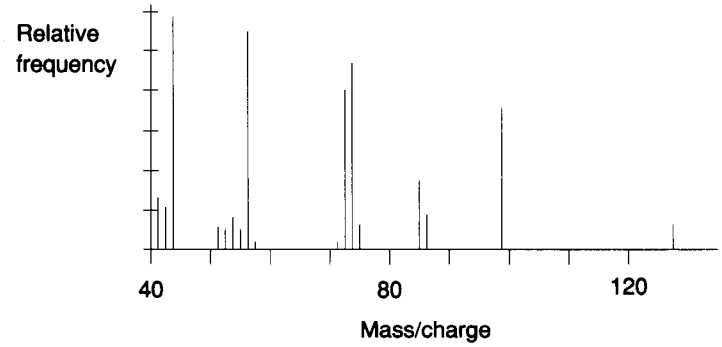
In figure 3.4, a sequence of procedures P1, ..., P5 cause transitions from state to state, starting from the initial current state. Each of the

DENDRAL Analyzes Mass Spectrograms

DENDRAL is one of the great classic application programs. To see what it does, suppose that an organic chemist wants to know the chemical nature of some substance newly created in the test tube. The first step, not the one of concern here, is to determine the number of atoms of various kinds in one molecule of the stuff. This step determines the chemical formula, such as $C_8H_{16}O$. The notation indicates that each molecule has eight atoms of carbon, 16 of hydrogen, and one of oxygen.

Once a sample's chemical formula is known, the chemist may use the sample's mass spectrogram to work out the way the atoms are arranged in the chemical's structure, thus identifying the isomer of the chemical.

The spectrogram machine bombards a sample with high energy electrons, causing the molecules to break up into charged chunks of various sizes. Then, the machine sorts the chunks by passing them through a magnetic field that deflects the high-charge, low-weight ones more than it does the low-charge, high-weight ones. The deflected chunks are collected, forming a spectrogram like the following:



The purpose of DENDRAL is to work, like a knowledgeable chemist, from a chemical formula and spectrogram to a deduced structure, producing a chemical structure like this:

$$\begin{array}{c} CH_3 - CH_2 - C - CH_2 - CH_2 - CH_2 - CH_2 - CH_3 \\ \qquad\qquad\quad \| \\ \qquad\qquad\quad O \end{array}$$

The DENDRAL program works out structures from chemical formulas and mass spectrograms using the generate-and-test method. The generator consists of a structure enumerator and a synthesizer that produces a synthetic mass spectrogram by simulating the action of a real mass spectrometer on each enumerated structure.

The structure enumerator ensures that the overall generator is complete and nonredundant because the structure enumerator uses a provably complete and nonredundant structure-enumeration procedure. The overall generator is also informed, because the structure enumerator uses the chemical formula and knowledge about necessary and forbidden substructures.

The tester compares the real mass spectrogram with those produced by the generator. The possible structures are those whose synthetic spectrograms match the real one adequately. The structure judged correct is the one whose synthetic spectrogram most closely matches the real one.

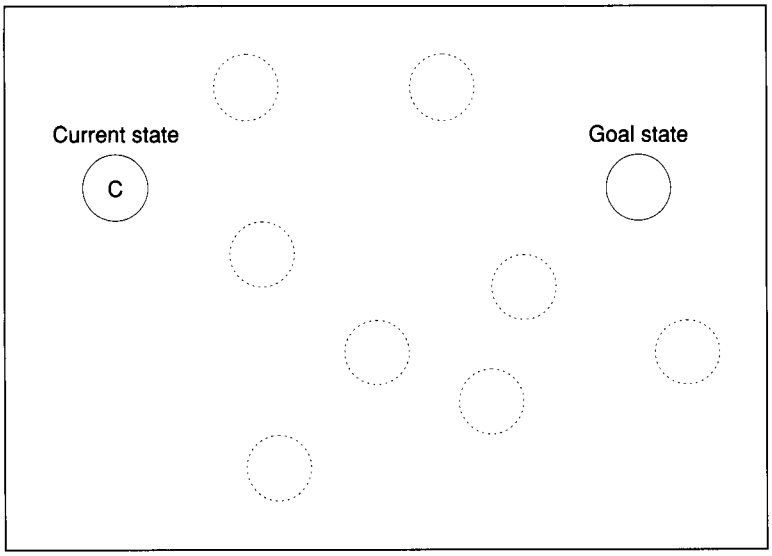


Figure 3.3 Means–ends analysis involves states and procedures for reducing differences between states. The current state and goal state are shown solid; other states, not yet encountered, are shown dotted.

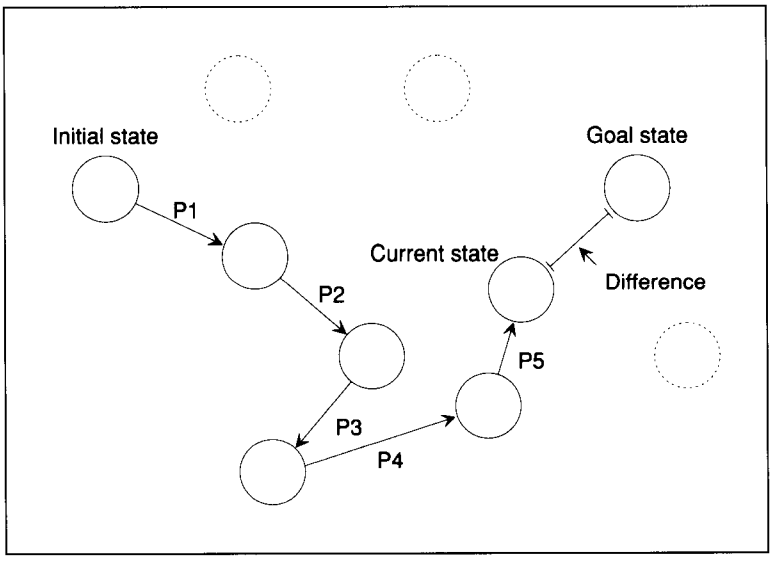


Figure 3.4 Means–ends analysis produces a path through state space. The current state, the goal state, and a description of their difference determine which procedure to try next. Note that the procedures are expected, but not guaranteed, to cause a transition to a state that is nearer the goal state than is the current state.

procedures is selected because it is believed to be relevant to reducing the difference between the state in which it was applied and the goal state. Note, however, that procedure P3 takes the problem solver farther away from the goal; there is no built-in mechanism preventing backward steps in the most general form of means–ends analysis. Fortunately, procedure P4 and procedure P5 take the problem solver back toward the goal.

In summary, here is the means–ends procedure expressed, more precisely, in procedural English:

To perform means–ends analysis,

- ▷ Until the goal is reached or no more procedures are available,
  - ▷ Describe the current state, the goal state, and the difference between the two.
  - ▷ Use the difference between the current state and goal state, possibly with the description of the current state or goal state, to select a promising procedure.
  - ▷ Use the promising procedure and update the current state.
- ▷ If the goal is reached, announce success; otherwise, announce failure.

Difference-Procedure Tables Often Determine the Means

Whenever the description of the difference between the current state and the goal state is the key to which procedure to try next, a simple **difference-procedure table** may suffice to connect difference descriptions to preferred procedures.†

Consider, for example, a travel situation in which the problem is to find a way to get from one city to another. One traveler's preferences might link the preferred transportation procedure to the difference between states, described in terms of the distance between the cities involved, via the following difference-procedure table:

Distance	Airplane	Train	Car
More than 300 miles	√		
Between 100 and 300 miles		√	
Less than 100 miles			√

Thus, the difference-procedure table determines generally what to do, leaving descriptions of the current state and destination state with no purpose other than to specify the origin and destination for the appropriate procedure.

THE PROBLEM-REDUCTION METHOD

Sometimes, it is possible to convert difficult goals into one or more easier-to-achieve subgoals. Each subgoal, in turn, may be divided still more finely into one or more lower-level subgoals.

†Because transition-causing procedures are often called *operators*, a difference-procedure table is called a *difference-operator table* in some circles.

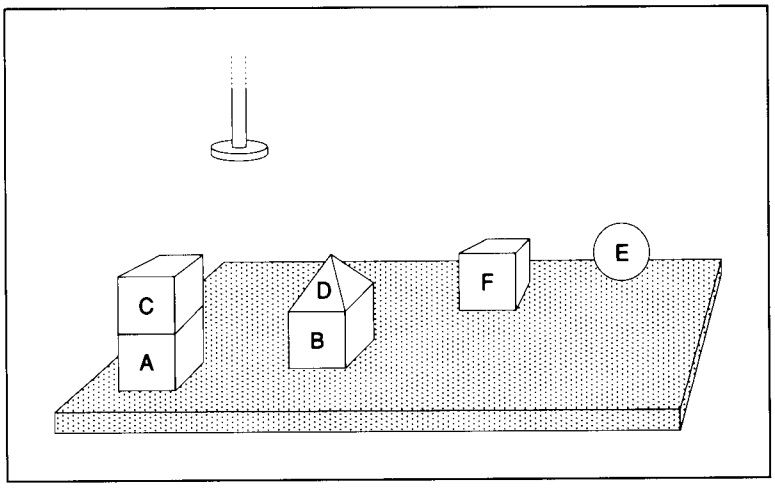


Figure 3.5 MOVER is a procedure for planning motion sequences in the world of bricks, pyramids, balls, and a robot hand.

When using the **problem-reduction method**, you generally recognize goals and convert them into appropriate subgoals. When so used, problem reduction is often called, equivalently, **goal reduction**.

In the rest of this section, you learn more about the problem-reduction method, you learn which problems the problem-reduction method solves, and you learn how the problem-reduction method makes it easy to answer certain "why?" and "how?" questions.

Moving Blocks Illustrates Problem Reduction

The MOVER procedure solves problems in block manipulation and answers questions about its own behavior. MOVER works with blocks such as the one shown in figure 3.5, obeying commands such as the following:

Put <block name> on <another block name>.

To obey, MOVER plans a motion sequence for a one-handed robot that picks up only one block at a time. MOVER consists of procedures that reduce given problems to simpler problems, thus engaging in what is called **problem reduction**. Conveniently, the names of these procedures are mnemonics for the problems that the procedures reduce. Figure 3.6 shows how the procedures fit together.

- PUT-ON arranges to place one block on top of another block. It works by activating other procedures that find a specific place on the top of the target block, grasping the traveling block, moving it, and ungrasping it at the specific place.
- GET-SPACE finds space on the top of a target block for a traveling block.
- MAKE-SPACE helps GET-SPACE, when necessary, by moving obstructions until there is enough room for a traveling block.
- GRASP grasps blocks. If the robot hand is holding a block when GRASP is invoked, GRASP must arrange for the robot hand to get rid of that

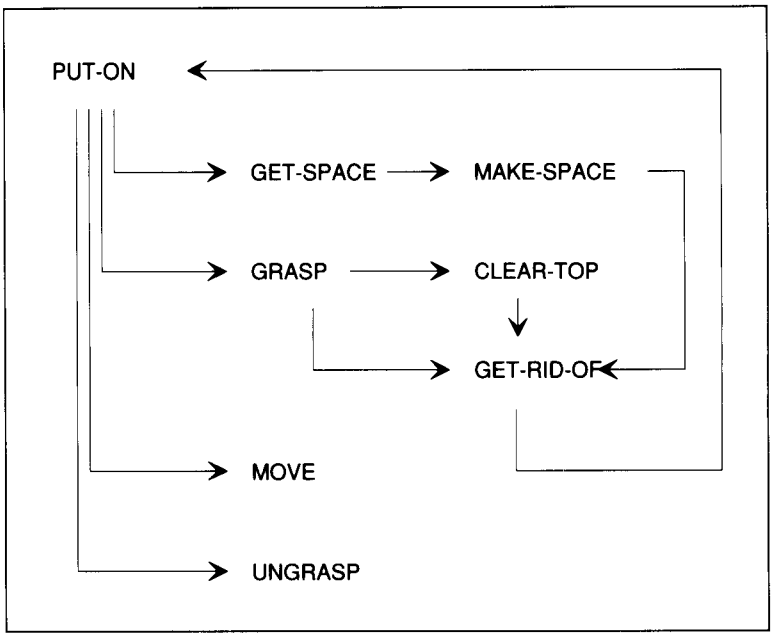


Figure 3.6 Specialists for moving blocks.

block. Also, GRASP must arrange to clear off the top of the object to be grasped.

- CLEAR-TOP does the top clearing. It works by getting rid of everything on top of the object to be grasped.
- GET-RID-OF gets rid of an obstructing object by putting it on the table.
- UNGRASP makes the robot's hand let go of whatever it is holding.
- MOVE moves objects, once they are held, by moving the robot hand.

Now imagine that there is a request to put block A on block B, given the situation shown in figure 3.5. Plainly, the following sequence suffices:

- Grasp D.
- Move D to some location on the table.
- Ungrasp D.
- Grasp C.
- Move C to some location on the table.
- Ungrasp C.
- Grasp A.
- Move A to some location on B.
- Ungrasp A.

The question is, How do the procedures in MOVER find the appropriate sequence? Here is the answer:

First, PUT-ON asks GET-SPACE to identify a place for block A on top of block B. GET-SPACE appeals to MAKE-SPACE because block D is in the way.

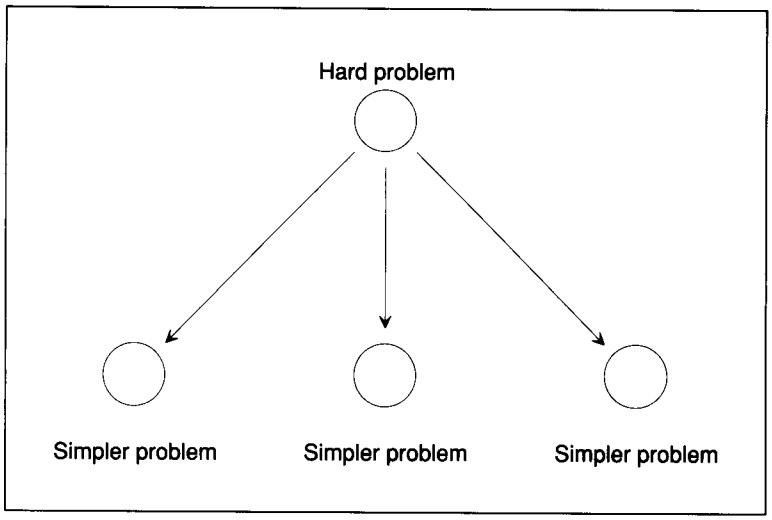


Figure 3.7 A goal tree. Branches connect supergoals to simpler subgoals.

MAKE-SPACE asks GET-RID-OF to help by getting rid of block D. GET-RID-OF obliges by finding a place for block D on the table and by moving block D to that place using PUT-ON.

Note that PUT-ON, at work placing block A on block B, eventually produces a new job for PUT-ON itself, this time to put block D on the table. When a procedure uses itself, the procedure is said to **recurse**. Systems in which procedures use themselves are said to be **recursive**.

With block D gone, MAKE-SPACE can find a place for block A to go on top of block B. Recall that MAKE-SPACE was asked to do this by GET-SPACE because PUT-ON has the duty of putting block A on B. PUT-ON can proceed now, asking GRASP to grasp block A. But GRASP realizes that it cannot grasp block A because block C is in the way. GRASP asks CLEAR-TOP for help. CLEAR-TOP, in turn, asks GET-RID-OF for help, whereupon GET-RID-OF arranges for block C to go on the table using PUT-ON.

With block A cleared, CLEAR-TOP is finished. But if there were many blocks on top of block A, rather than just one, CLEAR-TOP would appeal to GET-RID-OF many times, rather than just once.

Now GRASP can do its job, and PUT-ON can ask MOVE to move block A to the place found previously on top of block B. Finally, PUT-ON asks UNGRASP to let block A go.

The Key Idea in Problem Reduction Is to Explore a Goal Tree

A **semantic tree** is a semantic net with special links, called **branches**, each of which connects two nodes:

A **semantic tree** is a representation

That is a semantic net

In which

- ▷ Certain links are called branches. Each branch connects two nodes; the head node is called the **parent node** and the tail node is called the **child node**
- ▷ One node has no parent; it is called the **root node**. Other nodes have exactly one parent.
- ▷ Some nodes have no children; they are called **leaf nodes**.
- ▷ When two nodes are connected to each other by a chain of two or more branches, one is said to be the **ancestor**; the other is said to be the **descendant**.

With constructors that

- ▷ Connect a parent node to a child node with a branch link

With readers that

- ▷ Produce a list of a given node's children
- ▷ Produce a given node's parent

A **goal tree**, like the one shown in figure 3.7, is a semantic tree in which nodes represent goals and branches indicate how you can achieve goals by solving one or more subgoals. Each node's children correspond to **immediate subgoals**; each node's parent corresponds to the **immediate supergoal**. The top node, the one with no parent, is the **root** goal.

Goal Trees Can Make Procedure Interaction Transparent

A goal tree, such as the one in figure 3.8, makes complicated MOVER scenarios transparent. Clearing the top of block A is shown as an immediate subgoal of grasping block A. Clearing the top of block A is also a subgoal of putting block A at a place on top of block B, but it is not an immediate subgoal.

All the goals shown in the example are satisfied only when all of their immediate subgoals are satisfied. Goals that are satisfied only when *all* of their immediate subgoals are satisfied are called And goals. The corresponding nodes are called And nodes, and you mark them by placing arcs on their branches.

Most goal trees also contain Or goals; these goals are satisfied when *any* of their immediate subgoals are satisfied. The corresponding, unmarked nodes are called Or nodes.

Finally, some goals are satisfied directly, without reference to any subgoals. These goals are called **leaf goals**, and the corresponding nodes are called **leaf nodes**.

Because goal trees always involve And nodes, or Or nodes, or both, they are often called And–Or trees.

To determine whether a goal has been achieved, you need a testing procedure. The key procedure, REDUCE, channels action into the REDUCE-AND and the REDUCE-OR procedures:

To determine, using REDUCE, whether a goal is achieved,

- ▷ Determine whether the goal is satisfied without recourse to subgoals:
  - ▷ If it is, announce that the goal is satisfied.
  - ▷ Otherwise, determine whether the goal corresponds to an And goal:
    - ▷ If it does, use the REDUCE-AND procedure to determine whether the goal is satisfied.
    - ▷ Otherwise, use the REDUCE-OR procedure to determine whether the goal is satisfied.

REDUCE uses two subprocedures: one deals with And goals, and the other deals with Or goals:

To determine, using REDUCE-AND, whether a goal has been satisfied,

- ▷ Use REDUCE on each immediate subgoal until there are no more subgoals, or until REDUCE finds a subgoal that is not satisfied.
- ▷ If REDUCE has found a subgoal that is not satisfied, announce that the goal is not satisfied; otherwise, announce that the goal is satisfied.

To determine, using REDUCE-OR, whether a goal has been satisfied,

- ▷ Use REDUCE on each subgoal until REDUCE finds a subgoal that is satisfied.
- ▷ If REDUCE has found a subgoal that is satisfied, announce that the goal is satisfied; otherwise, announce that the goal is not satisfied.

With REDUCE, REDUCE-AND, and REDUCE-OR in hand, it is a simple matter to test an entire And–Or tree: you just use REDUCE on the root node,

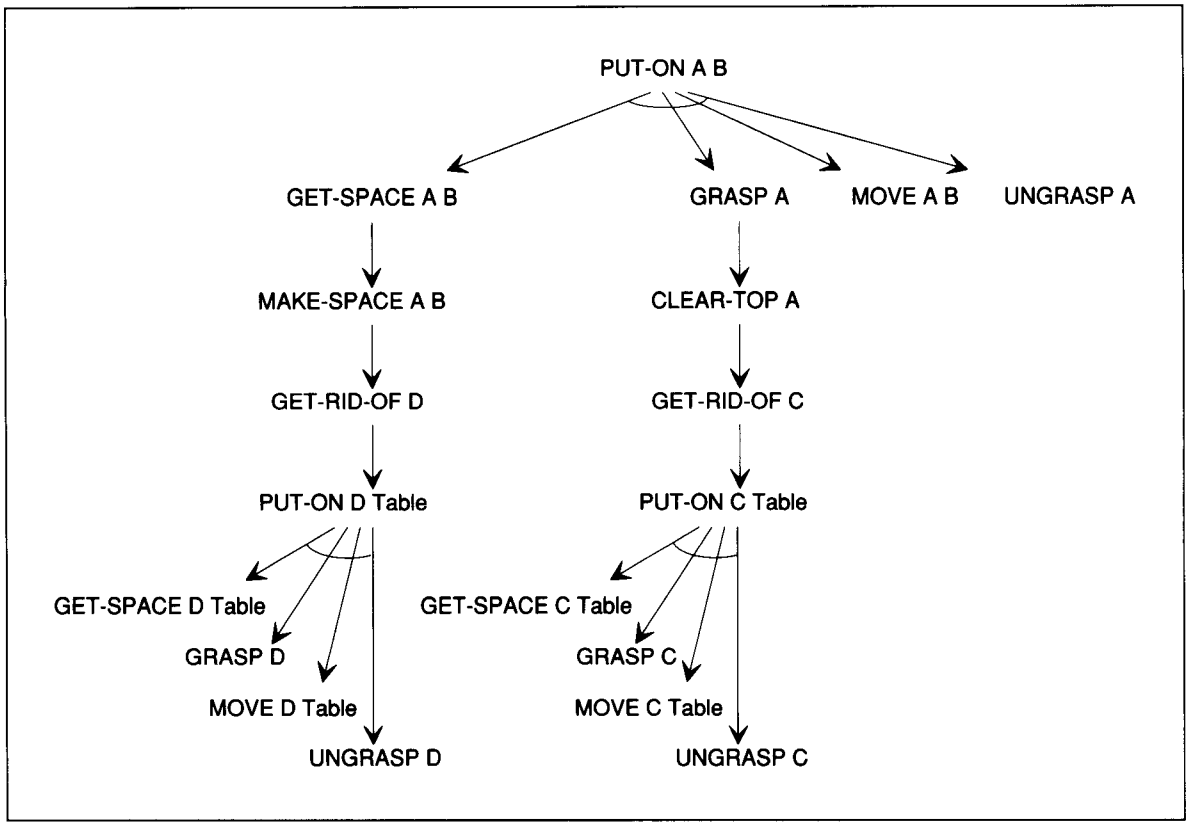


Figure 3.8 A goal tree. Branches joined by arcs are under And nodes; other branches are under Or nodes.

permitting the various procedures to call one another, as necessary, to work their way down the tree.

Goal Trees Enable Introspective Question Answering

The MOVER program is able to build And–Or trees because the specialists have a tight correspondence to identifiable goals. Indeed, MOVER's And–Or trees are so illuminating, they can be used to answer questions about *how* and *why* actions have been taken, giving MOVER a certain talent for introspection into its own behavior.

Suppose, for example, that MOVER has put block A on block B, producing the goal tree shown in figure 3.8.

Further suppose that someone asks, How did you clear the top of A? Plainly, a reasonable answer is, By getting rid of block C. On the other hand, suppose the question is, Why did you clear the top of A? Then a reasonable answer is, To grasp block A.

These examples illustrate general strategies. To deal with "how?" questions, you identify the goal involved in the And–Or tree. If the goal is an And goal, report all of the immediate subgoals. If the goal is an Or goal,

you report the immediate subgoal that was achieved. To deal with "why?" questions, you identify the goal and report the immediate supergoal.

Problem Reduction Is Ubiquitous in Programming

From a programming point of view, MOVER consists of a collection of specialized procedures. Each time one specialized procedure calls another, it effects a problem-reduction step.

More generally, whenever one procedure calls a subprocedure, there is a problem reduction step. Thus, problem reduction is the problem-solving method that all but the shortest programs exhibit in great quantity.

Problem-Solving Methods Often Work Together

Few real problems can be solved by a single problem-solving method. Accordingly, you often see problem-solving methods working together.

Suppose, for example, that you want to go from your house near Boston to a friend's house near Seattle. Earlier in this chapter, you learned that you can use means–ends analysis to decide what sort of transportation is most preferred for reducing the distance between where you are and where you want to be. Because the distance between Boston and Seattle is large, means–ends analysis doubtlessly would suggest that you take an airplane, but taking an airplane solves only part of your problem: You still have to figure out how get to the Boston airport from your house, and how to get from the Seattle airport to your friend's house. Thus, the initial goal, as shown in figure 3.9, becomes three subgoals, each of which can be handled, perhaps, by means–ends analysis.

SUMMARY

- Generate-and-test systems often do identification. Good generators are complete, nonredundant, and informed.
- The key idea in means–ends analysis is to reduce differences. Means–ends analysis is often mediated via difference-procedure tables.
- The key idea in problem reduction is to explore a goal tree. A goal tree consists of And goals, all of which must be satisfied, and Or goals, one of which must be satisfied.
- Problem reduction is ubiquitous in programming because subprocedure call is a form of problem reduction.
- The MOVER program uses problem reduction to plan motion sequences. While MOVER is at work, it constructs a goal tree that enables it to answer *how* and *why* questions.