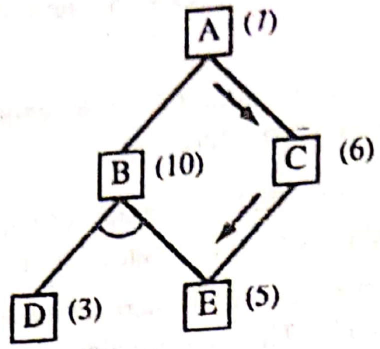

Figure 3.11: An Unnecessary Backward Propagation

revised and that change is not propagated up through B as well as through C, B may appear to be better. For example, if, as a result of expanding node E, we update its cost to 10, then the cost of C will be updated to 11. If this is all that is done, then when A is examined, the path through B will have a cost of only 11 compared to 12 for the path through C, and it will be labeled erroneously as the most promising path. In this example, the mistake might be detected at the next step, during which D will be expanded. If its cost changes and is propagated back to B, B's cost will be recomputed and the new cost of E will be used. Then the new cost of B will propagate back to A. At that point, the path through C will again be better. All that happened was that some time was wasted in expanding D. But if the node whose cost has changed is farther down in the search graph, the error may never be detected. An example of this is shown in Figure 3.12(*a*). If the cost of G is revised as shown in Figure 3.12(*b*) and if it is not immediately propagated back to E, then the change will never be recorded and a nonoptimal solution through B may be discovered.

A second point concerns the termination of the backward cost propagation of step 2(*c*). Because *GRAPH* may contain cycles, there is no guarantee that this process will terminate simply because it reaches the "top" of the graph. It turns out that the process can be guaranteed to terminate for a different reason, though. One of the exercises at the end of this chapter explores why.

3.5 Constraint Satisfaction

Many problems in AI can be viewed as problems of *constraint satisfaction* in which the goal is to discover some problem state that satisfies a given set of constraints. Examples of this sort of problem include cryptarithmetic puzzles (as described in Section 2.6) and many real-world perceptual labeling problems. Design tasks can also be viewed as constraint-satisfaction problems in which a design must be created within fixed limits on time, cost, and materials.

By viewing a problem as one of constraint satisfaction, it is often possible to reduce substantially the amount of search that is required as compared with a method that attempts to form partial solutions directly by choosing specific values for components of the eventual solution. For example, a straightforward search procedure to solve a cryptarithmetic problem might operate in a state space of partial solutions in which letters

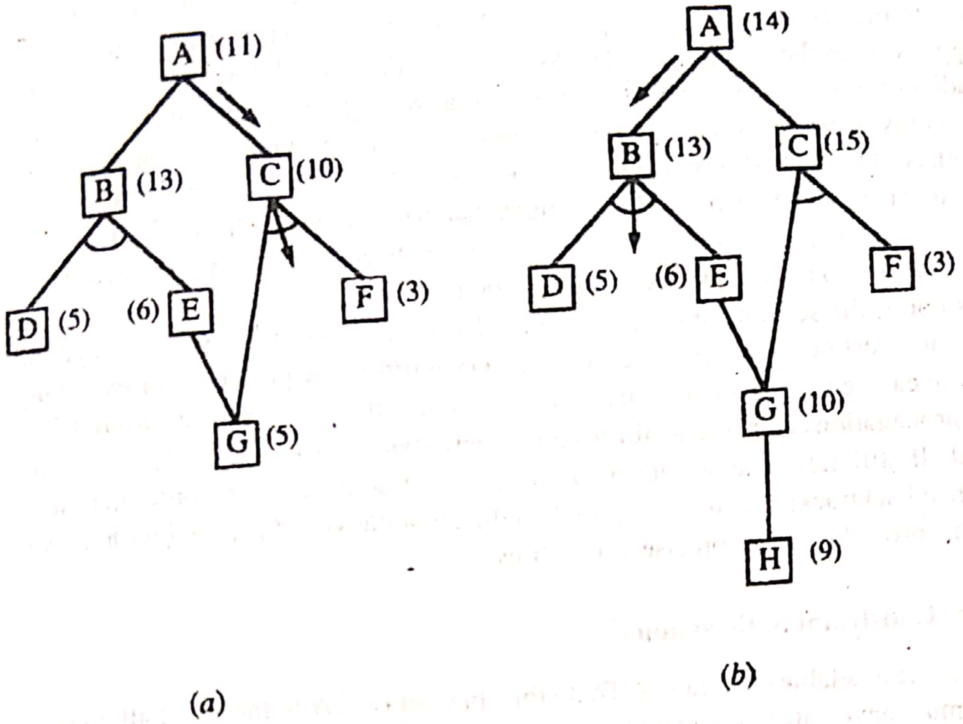

Figure 3.12: A Necessary Backward Propagation

are assigned particular numbers as their values. A depth-first control scheme could then follow a path of assignments until either a solution or an inconsistency is discovered. In contrast to this, a constraint satisfaction approach to solving this problem avoids making guesses on particular assignments of numbers to letters until it has to. Instead, the initial set of constraints, which says that each number may correspond to only one letter and that the sums of the digits must be as they are given in the problem, is first augmented to include restrictions that can be inferred from the rules of arithmetic. Then, although guessing may still be required, the number of allowable guesses is reduced and so the degree of search is curtailed.

Constraint satisfaction is a search procedure that operates in a space of constraint sets. The initial state contains the constraints that are originally given in the problem description. A goal state is any state that has been constrained "enough," where "enough" must be defined for each problem. For example, for cryptarithmetic, enough means that each letter has been assigned a unique numeric value.

Constraint satisfaction is a two-step process. First, constraints are discovered and propagated as far as possible throughout the system. Then, if there is still not a solution, search begins. A guess about something is made and added as a new constraint. Propagation can then occur with this new constraint, and so forth.

The first step, propagation, arises from the fact that there are usually dependencies among the constraints. These dependencies occur because many constraints involve more than one object and many objects participate in more than one constraint. So, for example, assume we start with one constraint, $N = E + 1$. Then, if we added the constraint $N = 3$, we could propagate that to get a stronger constraint on E, namely that $E = 2$. Constraint propagation also arises from the presence of inference rules that allow additional constraints to be inferred from the ones that are given. Constraint propagation terminates for one of two reasons. First, a contradiction may be detected. If this happens, then there is no solution consistent with all the known constraints. If the contradiction involves only those constraints that were given as part of the problem specification (as opposed to ones that were guessed during problem solving), then no solution exists. The second possible reason for termination is that the propagation has run out of steam and there are no further changes that can be made on the basis of current knowledge. If this happens and a solution has not yet been adequately specified, then search is necessary to get the process moving again.

At this point, the second step begins. Some hypothesis about a way to strengthen the constraints must be made. In the case of the cryptarithmetic problem, for example, this usually means guessing a particular value for some letter. Once this has been done, constraint propagation can begin again from this new state. If a solution is found, it can be reported. If still more guesses are required, they can be made. If a contradiction is detected, then backtracking can be used to try a different guess and proceed with it. We can state this procedure more precisely as follows:

Algorithm: Constraint Satisfaction

1. Propagate available constraints. To do this, first set *OPEN* to the set of all objects that must have values assigned to them in a complete solution. Then do until an inconsistency is detected or until *OPEN* is empty:

   (a) Select an object *OB* from *OPEN*. Strengthen as much as possible the set of constraints that apply to *OB*.

   (b) If this set is different from the set that was assigned the last time *OB* was examined or if this is the first time *OB* has been examined, then add to *OPEN* all objects that share any constraints with *OB*.

   (c) Remove *OB* from *OPEN*.

2. If the union of the constraints discovered above defines a solution, then quit and report the solution.

3. If the union of the constraints discovered above defines a contradiction, then return failure.

4. If neither of the above occurs, then it is necessary to make a guess at something in order to proceed. To do this, loop until a solution is found or all possible solutions have been eliminated:

   (a) Select an object whose value is not yet determined and select a way of strengthening the constraints on that object.

   (b) Recursively invoke constraint satisfaction with the current set of constraints augmented by the strengthening constraint just selected.

This algorithm has been stated as generally as possible. To apply it in a particular problem domain requires the use of two kinds of rules: rules that define the way constraints may validly be propagated and rules that suggest guesses when guesses are

Problem:

```
 SEND
+MORE
-----
MONEY
```

Initial State:

No two letters have the same value.

The sums of the digits must be as shown in the problem.

Figure 3.13: A Cryptarithmetic Problem

necessary. It is worth noting, though, that in some problem domains guessing may not be required. For example, the Waltz algorithm for propagating line labels in a picture, which is described in Chapter 14, is a version of this constraint satisfaction algorithm with the guessing step eliminated. In general, the more powerful the rules for propagating constraints, the less need there is for guessing.

To see how this algorithm works, consider the cryptarithmetic problem shown in Figure 3.13. The goal state is a problem state in which all letters have been assigned a digit in such a way that all the initial constraints are satisfied.

The solution process proceeds in cycles. At each cycle, two significant things are done (corresponding to steps 1 and 4 of this algorithm):

1. Constraints are propagated by using rules that correspond to the properties of arithmetic.
2. A value is guessed for some letter whose value is not yet determined.

In the first step, it does not usually matter a great deal what order the propagation is done in, since all available propagations will be performed before the step ends. In the second step, though, the order in which guesses are tried may have a substantial impact on the degree of search that is necessary. A few useful heuristics can help to select the best guess to try first. For example, if there is a letter that has only two possible values and another with six possible values, there is a better chance of guessing right on the first than on the second. Another useful heuristic is that if there is a letter that participates in many constraints then it is a good idea to prefer it to a letter that participates in a few. A guess on such a highly constrained letter will usually lead quickly either to a contradiction (if it is wrong) or to the generation of many additional constraints (if it is right). A guess on a less constrained letter, on the other hand, provides less information.

The result of the first few cycles of processing this example is shown in Figure 3.14. Since constraints never disappear at lower levels, only the ones being added are shown for each level. It will not be much harder for the problem solver to access the constraints as a set of lists than as one long list, and this approach is efficient both in terms of storage space and the ease of backtracking. Another reasonable approach for this problem would be to store all the constraints in one central database and also to record at each node the changes that must be undone during backtracking. C1, C2, C3, and C4 indicate the carry bits out of the columns, numbering from the right.

Initially, rules for propagating constraints generate the following additional constraints:

- M = 1, since two single-digit numbers plus a carry cannot total more than 19.
- S = 8 or 9, since S + M + C3 > 9 (to generate the carry) and M = 1, S + 1 + C3 > 9, so S + C3 > 8 and C3 is at most 1.
- O = 0, since S + M(1) + C3 (<= 1) must be at least 10 to generate a carry and it can be at most 11. But M is already 1, so O must be 0.
- N = E or E + 1, depending on the value of C2. But N cannot have the same value as E. So N = E + 1 and C2 is 1.
- In order for C2 to be 1, the sum of N + R + C1 must be greater than 9, so N + R must be greater than 8.
- N + R cannot be greater than 18, even with a carry in, so E cannot be 9.

At this point, let us assume that no more constraints can be generated. Then, to make progress from here, we must guess. Suppose E is assigned the value 2. (We chose to guess a value for E because it occurs three times and thus interacts highly with the other letters.) Now the next cycle begins.

The constraint propagator now observes that:

- N = 3, since N = E + 1.
- R = 8 or 9, since R + N (3) + C1 (1 or 0) = 2 or 12. But since N is already 3, the sum of these nonnegative numbers cannot be less than 3. Thus R + 3 + (0 or 1) = 12 and R = 8 or 9.
- 2 + D = Y or 2 + D = 10 + Y, from the sum in the rightmost column.

Again, assuming no further constraints can be generated, a guess is required. Suppose C1 is chosen to guess a value for. If we try the value 1, then we eventually reach dead ends, as shown in the figure. When this happens, the process will backtrack and try C1 = 0.

A couple of observations are worth making on this process. Notice that all that is required of the constraint propagation rules is that they not infer spurious constraints. They do not have to infer all legal ones. For example, we could have reasoned through to the result that C1 equals 0. We could have done so by observing that for C1 to be 1, the following must hold: 2 + D = 10 + Y. For this to be the case, D would have to be 8 or 9. But both S and R must be either 8 or 9 and three letters cannot share two values. So C1 cannot be 1. If we had realized this initially, some search could have been avoided. But since the constraint propagation rules we used were not that sophisticated,

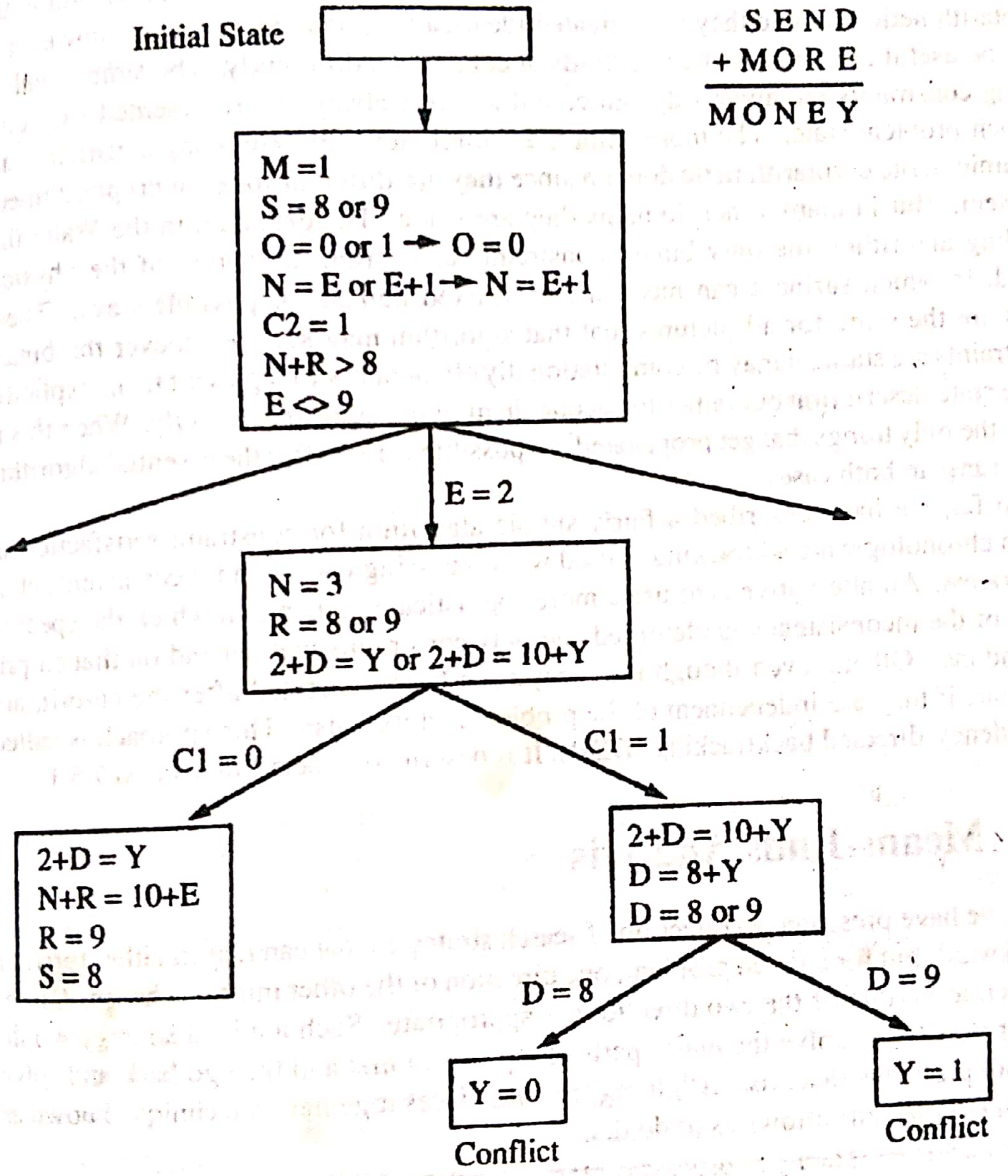

Figure 3.14: Solving a Cryptarithmetic Problem

it took some search. Whether the search route takes more or less actual time than does the constraint propagation route depends on how long it takes to perform the reasoning required for constraint propagation.

A second thing to notice is that there are often two kinds of constraints. The first kind are simple; they just list possible values for a single object. The second kind are more complex; they describe relationships between or among objects. Both kinds of constraints play the same role in the constraint satisfaction process, and in the cryptarithmetic example they were treated identically. For some problems, however, it may be useful to represent the two kinds of constraints differently. The simple, value-listing constraints are always dynamic, and so must always be represented explicitly in each problem state. The more complicated, relationship-expressing constraints are dynamic in the cryptarithmetic domain since they are different for each cryptarithmetic problem. But in many other domains they are static. For example, in the Waltz line labeling algorithm, the only binary constraints arise from the nature of the physical world, in which surfaces can meet in only a fixed number of possible ways. These ways are the same for all pictures that that algorithm may see. Whenever the binary constraints are static, it may be computationally efficient not to represent them explicitly in the state description but rather to encode them in the algorithm directly. When this is done, the only things that get propagated are possible values. But the essential algorithm is the same in both cases.

So far, we have described a fairly simple algorithm for constraint satisfaction in which chronological backtracking is used when guessing leads to an inconsistent set of constraints. An alternative is to use a more sophisticated scheme in which the specific cause of the inconsistency is identified and only constraints that depend on that culprit are undone. Others, even though they may have been generated after the culprit, are left alone if they are independent of the problem and its cause. This approach is called dependency-directed backtracking (DDB). It is described in detail in Section 7.5.1.

3.6 Means-Ends Analysis

So far, we have presented a collection of search strategies that can reason either forward or backward, but for a given problem, one direction or the other must be chosen. Often, however, a mixture of the two directions is appropriate. Such a mixed strategy would make it possible to solve the major parts of a problem first and then go back and solve the small problems that arise in "gluing" the big pieces together. A technique known as *means-ends analysis* allows us to do that.

The means-ends analysis process centers around the detection of differences between the current state and the goal state. Once such a difference is isolated, an operator that can reduce the difference must be found. But perhaps that operator cannot be applied to the current state. So we set up a subproblem of getting to a state in which it can be applied. The kind of backward chaining in which operators are selected and then subgoals are set up to establish the preconditions of the operators is called *operator subgoaling*. But maybe the operator does not produce exactly the goal state we want. Then we have a second subproblem of getting from the state it does produce to the goal. But if the difference was chosen correctly and if the operator is really effective at reducing the difference, then the two subproblems should be easier to solve than the