Depending on how the operators are chosen, the 8-Puzzle and the blocks world problem can also be considered partially commutative.

Both types of partially commutative production systems are significant from an implementation point of view because they tend to lead to many duplications of individual states during the search process. This is discussed further in Section 2.5.

Production systems that are not partially commutative are useful for many problems in which irreversible changes occur. For example, consider the problem of determining a process to produce a desired chemical compound. The operators available include such things as "Add chemical x to the pot" or "Change the temperature to t degrees." These operators may cause irreversible changes to the potion being brewed. The order in which they are performed can be very important in determining the final output. It is possible that if x is added to y, a stable compound will be formed, so later addition of z will have no effect; if z is added to y, however, a different stable compound may be formed, so later addition of x will have no effect. Nonpartially commutative production systems are less likely to produce the same node many times in the search process. When dealing with ones that describe irreversible processes, it is particularly important to make correct decisions the first time, although if the universe is predictable, planning can be used to make that less important.

2.5 ISSUES IN THE DESIGN OF SEARCH PROGRAMS

Every search process can be viewed as a traversal of a tree structure in which each node represents a problem state and each arc represents a relationship between the states represented by the nodes it connects. For example, Fig. 2.18 shows part of a search tree for a water jug problem. The arcs have not been labeled in the Fig., but they correspond to particular water-pouring operations. The search process must find a path or paths through the tree that connect an initial state with one or more final states. The tree that must be searched could, in principle, be constructed in its entirety from the rules that define allowable moves in the problem space. But, in practice, most of it never is. It is too large and most of it need never be explored. Instead of first building the tree *explicitly* and then searching it, most search programs represent the tree *implicitly* in the rules and generate explicitly only those parts that they decide to explore. Throughout our discussion of search methods, it is important to keep in mind this distinction between implicit search trees and the explicit partial search trees that are actually constructed by the search program.

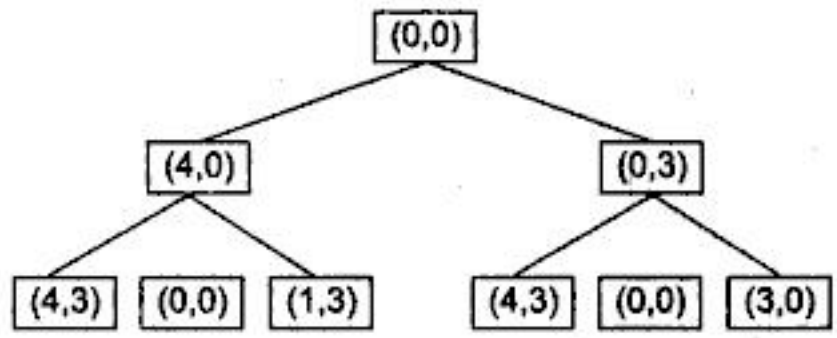

Fig. 2.18 *A Search Tree for the Water Jug Problem*

In the next chapter, we present a family of general-purpose search techniques. But before doing so we need to mention some important issues that arise in all of them:

- The direction in which to conduct the search (*forward* versus *backward* reasoning). We can search forward through the state space from the start state to a goal state, or we can search backward from the goal.
- How to select applicable rules *(matching).* Production systems typically spend most of their time looking for rules to apply, so it is critical to have efficient procedures for matching rules against states.
- How to represent each node of the search process (the *knowledge representation problem* and the *frame problem).* For problems like chess, a node can be fully represented by a simple array. In more complex problem solving, however, it is inefficient and/or impossible to represent all of the facts in the world and to determine all of the side effects an action may have.

We discuss the knowledge representation and frame problems further in Chapter 4. We investigate matching and forward versus backward reasoning when we return to production systems in Chapter 6.

One other issue we should consider at this point is that of search trees versus search graphs. As mentioned above, we can think of production rules as generating nodes in a search tree. Each node can be expanded in turn, generating a set of successors. This process continues until a node representing a solution is found. Implementing such a procedure requires little bookkeeping. However, this process often results in the same node being generated as part of several paths and so being processed more than once. This happens because the search space may really be an arbitrary directed graph rather than a tree.

For example, in the tree shown in Fig. 2.18, the node (4,3), representing 4-gallons of water in one jug and 3 gallons in the other, can be generated either by first filling the 4-gallon jug and then the 3-gallon one or by filling them in the opposite order. Since the order does not matter, continuing to process both these nodes would be redundant. This example also illustrates another problem that often arises when the search process operates as a tree walk. On the third level, the node (0, 0) appears. (In fact, it appears twice.) But this is the same as the top node of the tree, which has already been expanded. Those two paths have not gotten Us anywhere. So we would like to eliminate them and continue only along the other branches.

The waste of effort that arises when the same node is generated more than once can be avoided at the price of additional bookkeeping. Instead of traversing a search tree, we traverse a directed graph. This graph differs from a tree in that several paths may come together at a node. The graph corresponding to the tree of Fig. 2.18 is shown in Fig. 2.19.

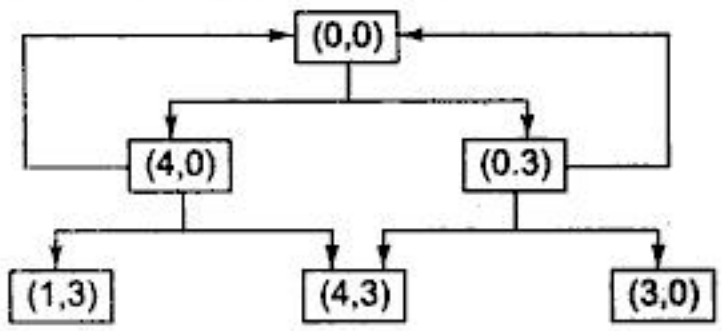

Fig. 2.19 *A Search Graph for the Water Jug Problem*

Any tree search procedure that keeps track of all the nodes that have been generated so far can be converted to a graph search procedure by modifying the action performed each time a node is generated. Notice that of the two systematic search procedures we have discussed so far, this requirement that nodes be kept track of is met by breadth-first search but not by depth-first search. But, of course, depth-first search could be modified, at the expense of additional storage, to retain in memory nodes that have been expanded and then backed-up over. Since all nodes are saved in the search graph, we must use the following algorithm instead of simply adding a new node to the graph.

Algorithm: Check Duplicate Nodes

1. Examine the set of nodes that have been created so far to see if the new node already exists.
2. If it does not-simply add it to the graph just as for a tree.
3. If it does already exist, then do the following:
   (a) Set the node that is being expanded to point to the already existing-node corresponding to its successor rather than to the new one. The new one can simply be thrown away.
   (b) If you are keeping track of the best (shortest or otherwise least-cost) path to each node, then check to see if the new path is better or worse than the old one. If worse, do nothing. If better, record the new path as the correct path to use to get to the node and propagate the corresponding change in cost down through successor nodes as necessary.

One problem that may arise here is that cycles may be introduced into the search graph. A *cycle* is a path through the graph in which a given node appears more than once. For example, the graph of Fig. 2.19 contains two cycles of length two. One includes the nodes (0, 0) and (4, 0); the other includes the nodes (0, 0) and (0, 3). Whenever there is a cycle, there can be paths of arbitrary length. Thus it may become more difficult to show that a graph traversal algorithm is guaranteed to terminate.

Treating the search process as a graph search rather than as a tree search reduces the amount of effort that is spent exploring essentially the same path several times. But it requires additional effort each time a node is

generated to see if it has been generated before. Whether this effort is justified depends on the particular problem. If it is very likely that the same node will be generated in several different ways, then it is more worthwhile to use a graph procedure than if such duplication will happen only rarely.

Graph search procedures are especially useful for dealing with partially commutative production systems in which a given set of operations will produce the same result regardless of the order in which the operations are applied. A systematic search procedure will try many of the permutations of these operators and so will generate the same node many times. This is exactly what happened in the water jug example shown above.

2.6 ADDITIONAL PROBLEMS

Several specific problems have been discussed throughout this chapter. Other problems have not yet been mentioned, but are common throughout the AI literature. Some have become such classics that no AI book could be complete without them, so we present them in this section. A useful exercise, at this point, would be to evaluate each of them in light of the seven problem characteristics we have just discussed.

A brief justification is perhaps required before this parade of toy problems is presented. Artificial intelligence is not merely a science of toy problems and microworlds (such as the blocks world). Many of the techniques that have been developed for these problems have become the core of systems that solve very nontoy problems. So think about these problems not as defining the scope of AI but rather as providing a core from which much more has developed.

The Missionaries and Cannibals Problem

Three missionaries and three cannibals find themselves on one side of a river. They have agreed that they would all like to get to the other side. But the missionaries are not sure what else the cannibals have agreed to. So the missionaries want to manage the trip across the river in such a way that the number of missionaries on either side of the river is never less than the number of cannibals who are on the same side. The only boat available holds only two people at a time. How can everyone get across the river without the missionaries risking being eaten?

The Tower of Hanoi

Somewhere near Hanoi there is a monastery whose monks devote their lives to a very important task. In their courtyard are three tall posts. On these posts is a set of sixty-four disks, each with a hole in the center and each of a different radius. When the monastery was established, all of the disks were on one of the posts, each disk resting on the one just larger than it. The monks' task is to move all of the disks to one of the other pegs. Only one disk may be moved at a time, and all the other disks must be on one of the pegs. In addition, at no time during the process may a disk be placed on top of a smaller disk. The third peg can, of course, be used as a temporary resting place for the disks. What is the quickest way for the monks to accomplish their mission?

Even the best solution to this problem will take the monks a very long time. This is fortunate, since legend has it that the world will end when they have finished.

The Monkey and Bananas Problem

A hungry monkey finds himself in a room in which a bunch of bananas is hanging from the ceiling. The monkey, unfortunately, cannot reach the bananas. However, in the room there are also a chair and a stick. The ceiling is just the right height so that a monkey standing on a chair could knock the bananas down with the stick. The monkey knows how to move around, carry other things around, reach for the bananas, and wave a stick in the air. What is the best sequence of actions for the monkey to take to acquire lunch?