8 Rules Substrates and Cognitive Modeling

In this chapter, you learn that you can build important capabilities on top of the rule-chaining problem-solving method. In particular, simple ideas enable rule-based systems *to explain the steps* that have led to a conclusion, *to reason in a variety of styles*, *to reason under time constraints*, *to determine a conclusion's probability*, and *to check the consistency of a newly proposed rule.*

You learn about *knowledge engineering*, and about two key heuristics that knowledge engineers use to acquire rules from human experts.

You also learn that rule-based systems have limits, when viewed from an engineering perspective, that render them too weak to be a universal tool for implementers who desire to capture all the characteristics of human experts. Accordingly, you will know that you should avoid the misnomer **expert system**, which commonly is used for rule-based systems that really perform at a level more like that of a novice.

Nevertheless, some cognitive scientists believe rules and rule chaining constitute part of a larger explanation of human information processing. By way of illustration, you learn about the SOAR *problem-solving architecture.*

RULE-BASED SYSTEMS VIEWED AS SUBSTRATE

In this section, you learn about the benefits and limitations that follow from using rules to represent knowledge and using rule chaining to solve problems. You also learn how to extract knowledge in general and rules in particular from human experts.

Explanation Modules Explain Reasoning

One way to show what a set of antecedent–consequent rules can do is to draw an **inference net** like those introduced in Chapter 7: one showed how forward chaining led to the conclusion that a particular animal is a giraffe; and another led to the verification of an hypothesis that a particular animal is a cheetah.

Sometimes, it is useful to look at part of an inference net to answer questions about *why* an assertion was used or about *how* an assertion was established. Much of this ability stands on the simple, highly constrained nature of rules. To decide how a given assertion was concluded, a rule-based deduction system needs to reflect only on the antecedent–consequent rules it has used, looking for those that contain the given assertion as a consequent. The required answer is just an enumeration of those antecedent–consequent rules, perhaps accompanied by information about their triggering assertions.

Consider, for example the cheetah example from Chapter 7. In that example, rule Z6 was used as shown in figure 8.1. Accordingly, if you ask ZOOKEEPER "How did you show that Swifty is a carnivore?" then ZOOKEEPER can determine the answer by moving to the left, saying, "By using rule Z6 and by knowing that Swifty is a mammal, has pointed teeth, claws, and forward-pointing eyes." If you ask "Why did you show that Swifty is a mammal?" then ZOOKEEPER can determine the answer by moving to the right, saying, "Because I wanted to use rule Z6 to show that the animal is a carnivore." Here, by way of summary, is the procedure ZOOKEEPER uses:

To answer a question about the reasoning done by a rule-based deduction system,

- ▷ If the question is a *how* question, report the assertions connected to the *if* side of the rule that established the assertion referenced in the question.
- ▷ If the question is a *why* question, report the assertions connected to the *then* sides of all rules that used the assertion referenced in the question.

Other questions, such as "Did you use rule Z6?" and "When did you use rule Z6?" are also easy to answer.

Of course, this ability to answer questions is not a surprise: you learned in Chapter 3 that goal trees always enable you to answer simple *how* and *why* questions, and an inference net always contains an implied goal tree. Figure 8.2 shows the goal tree that reaches from assertions at the bottom, through antecedent–consequent rules, to the cheetah conclusion at the top.

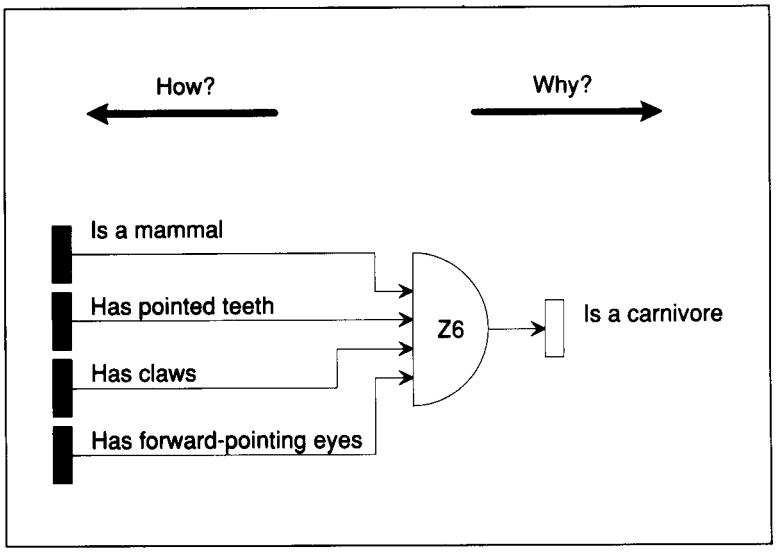

Figure 8.1 Rule-based systems retain a history of how they have tied assertions together, enabling a kind of introspection. The system answers *How did you show ... ?* by moving one step backward, mentioning rules and antecedents; it answers *Why did you show ... ?* questions by moving one step forward.

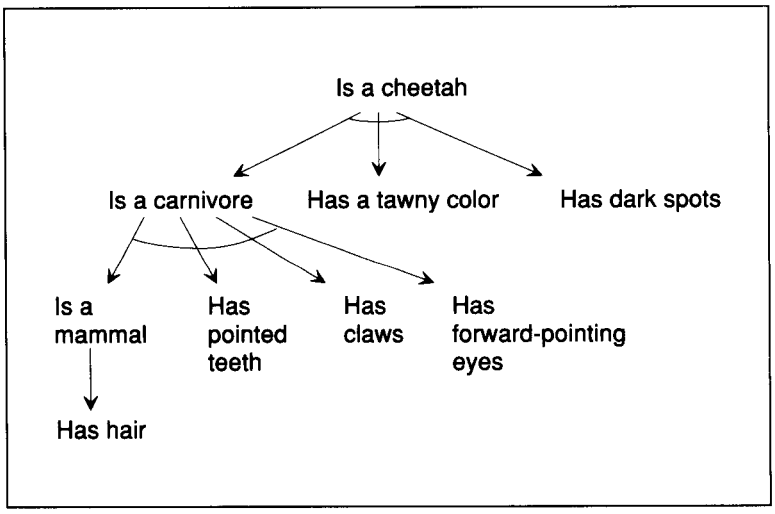

Figure 8.2 An inference net can be used to produce a goal tree. Each node in this goal tree corresponds to the application of an antecedent–consequent rule.

Reasoning Systems Can Exhibit Variable Reasoning Styles

Rule-based systems are often used in domains in which some assertions are almost certainly true and others are almost certainly false.

In the horse-evaluation system, taken up in Chapter 7, the principal rule says that a horse is valuable if it has parented something fast, which is a way of saying that a horse qualifies as a good stud or brood mare.

Nevertheless, the conclusion breaks down if a horse happens to become sterile, so a more reliable rule should say something about fertility:

Fertile-Parent Rule

If	*?x* is-a horse
	?x is-a-parent-of *?y*
	?y is fast
	?x is fertile
then	*?x* is valuable

If you try to use a system containing this rule at an auction, however, you will not buy many horses, because it is time consuming and expensive to determine whether a horse is fertile. Accordingly, you might well elect to assume that all horses are fertile, banishing the *?x is fertile* antecedent to a new part of the rule where **providing assumptions** are collected:

Modified Fertile-Parent Rule

If	*?x* is-a horse
	?x is-a-parent-of *?y*
	?y is fast
then	*?x* is valuable
providing	*?x* is fertile

Now you can arrange, easily, for your system to run in two modes:

- **Show-me mode**: Treat the providing assumptions as though they were ordinary antecedents; thus, refuse to accept anything that is not shown.
- **Ask-questions-later mode**: Ignore all providing assumptions. This mode is good in situations in which rapid response is more important than careful analysis.

You can even incorporate **unless assumptions** into your rules to complement the providing assumptions. Here is fanciful example:

Live-Horse Rule

If	*?x* is-a horse
	?x is-a-parent-of *?y*
	?y is fast
	?x is alive
then	*?x* is valuable

Taking the likelihood of thinking about the value of a dead horse to be small, you can rewrite the Live-Horse Rule as follows, moving the *?x is alive* assertion to a new part of the rule where unless assumptions are collected:

Modified Live-Horse Rule

If	*?x* is-a horse
	?x is-a-parent-of *?y*
	?y is fast
then	*?x* is valuable
unless	*?x* is dead

In ask-questions-later mode, unless assumptions are ignored, but in show-me mode, unless patterns are treated as ordinary antecedents, except that they appear in negated form. Thus, *?x is dead* becomes *?x is alive.*

Of course, there are other ways to treat providing assumptions and unless assumptions, in addition to show-me mode and ask-questions-later mode. Essentially, each way specifies whether you work on providing assumptions and unless assumptions using assertions or rules or neither or both. Here are representative examples:

- **Decision-maker mode**: Assume that providing assumptions are true. Assume that unless assumptions are false, unless they match existing assertions.
- **Trusting-skeptic mode**: Assume that providing assumptions are true and that unless assumptions are false, unless you can show otherwise with a single rule. This mode is called the trusting-skeptic mode because you assume that the likelihood of overturning providing and unless assumptions is small and therefore trying to overturn them is worthy of but little effort.
- **Progressive-reliability mode**: Produce an answer in ask-questions-later mode, and then, as time permits, explore more and more of the providing and unless assumptions, producing a more and more reliable answer. This mode is reminiscent of the progressive deepening idea, introduced in Chapter 6, in connection with adversarial search.

Probability Modules Help You to Determine Answer Reliability

Rule-based deduction systems used for identification usually work in domains where conclusions are rarely certain, even when you are careful to incorporate everything you can think of into rule antecedents. Thus, rule-based deduction system developers often build some sort of certainty-computing procedure on top of the basic antecedent–consequent apparatus. Generally, certainty-computing procedures associate a probability between 0 and 1 with each assertion. Each probability reflects how certain an assertion is, with 0 indicating that an assertion is definitely false and 1 indicating that an assertion is definitely true.

Two Key Heuristics Enable Knowledge Engineers to Acquire Knowledge

A **domain expert** is a person who has accumulated a great deal of skill in handling problems in a particular area called the **domain of expertise**. **Knowledge engineering** is the extraction of useful knowledge from domain experts for use by computers. Often, albeit far from always, knowledge engineers expect to cast the acquired knowledge in the form of rules for rule-based systems.

To some extent, knowledge engineering is an art, and some people become more skilled at it than do others. Nevertheless, there are two key heuristics that enable any knowledge engineer to do the job well.

The first of the key knowledge-engineering heuristics is the *heuristic of specific situations.* According to this heuristic, it is dangerous to limit inquiry to office interviews, asking only the general question, "How do you do your job?" Instead, a knowledge engineer should go into the field to watch domain experts proceed about their normal business, asking copious questions about specific situations as those situations are witnessed.

Suppose, for example, that you are a knowledge engineer and that you are working on new rules to be used by the BAGGER system introduced in Chapter 7. If you ignore the heuristic of specific situations, you would have just asked a few real grocery-store baggers to describe what they do. But no matter how cooperative your domain experts are, they are unlikely to be able to help you much unless you provide more evocative stimulation. You might well learn nothing about what to do with, for example, eggs. On the other hand, if you adhere to the heuristic of specific situations, you would get yourself into a grocery store so as to watch baggers handle specific situations, like the one shown in figure 8.3, asking questions like, "Do you always put the eggs on top?"

The second of the key knowledge-engineering heuristics is the *heuristic of situation comparison.* The idea is to ask a domain expert for clarification whenever the domain expert's behavior varies in situations that look identical to the knowledge engineer. The purpose is to help the knowledge engineer to acquire a vocabulary that is sufficiently rich to support the necessary acquisition of knowledge.

Again, if you were the knowledge engineer charged with acquiring the knowledge needed to support BAGGER, you would need to understand the important characteristics of groceries from the bagging perspective. Noting, for example, that a real bagger handles the two items in figure 8.4 differently, you would ask why, whereupon the bagger would say something about frozen food, thus signaling that *frozen* is a word that should be in BAGGER's vocabulary.

Note that ideas for extracting knowledge for computers also apply when your motive is to extract knowledge for your own use. Accordingly, the two key heuristics for knowledge engineering deserve powerful-idea status:

To learn from an expert,

- ▷ Ask about specific situations to learn the expert's general knowledge.
- ▷ Ask about situation pairs that look identical, but that are handled differently, so that you can learn the expert's vocabulary.

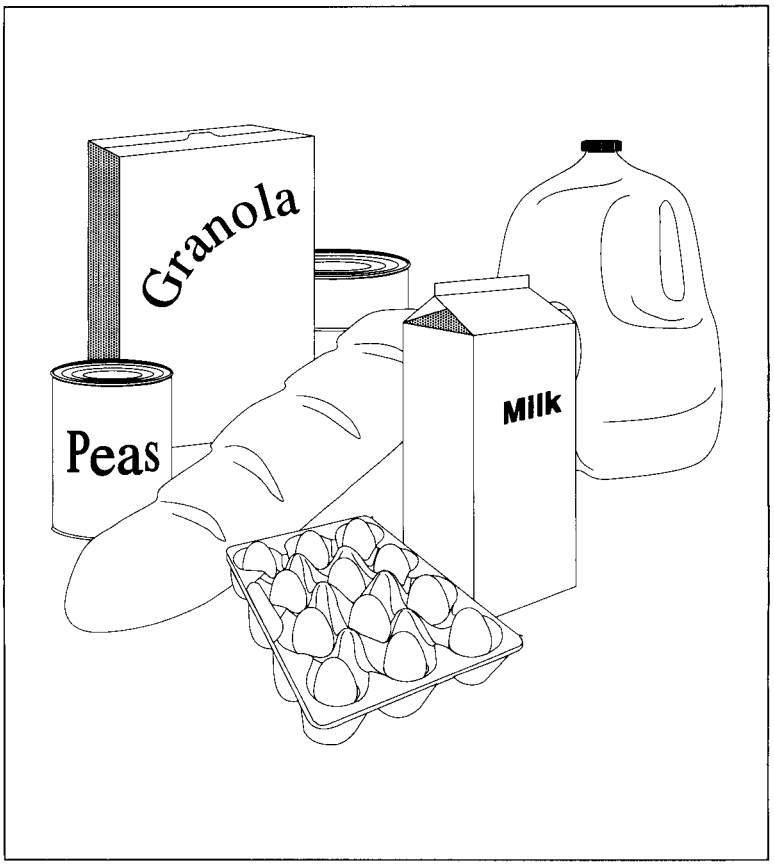

Figure 8.3 Knowledge engineers generally start their work by asking domain experts a few general questions about what experts do. Eventually, most knowledge-engineering jobs require knowledge engineers to use the heuristic of specific situations, watching domain experts at work on concrete examples.

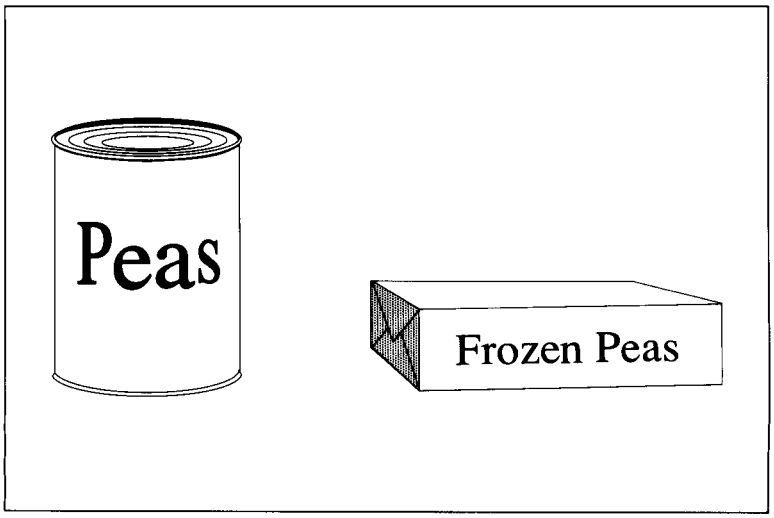

Figure 8.4 Knowledge engineers often ask why similar-looking situations are different, thus building essential vocabulary via the heuristic of situation comparison. Here, the domain expert will note that the essential difference is that one item is frozen.

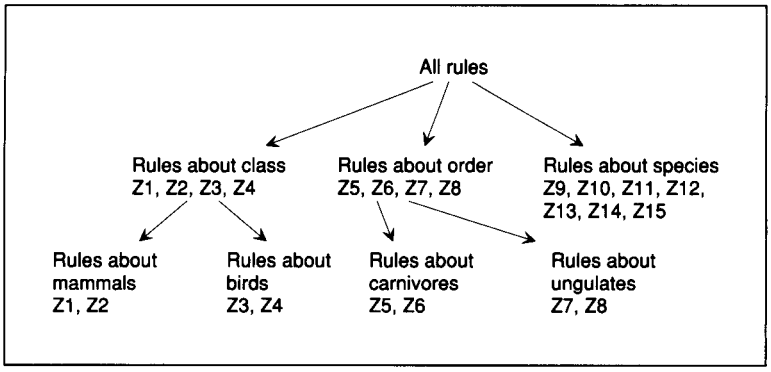

Figure 8.5 The rules in a rule-based deduction system form natural groups according to the conclusions expressed in their consequents.

Acquisition Modules Assist Knowledge Transfer

When you represent knowledge using if–then rules, you lose flexibility and power, but you gain the opportunity to add interesting capabilities to your basic problem-solving apparatus. To add a question-answering superprocedure, for example, you need to deal with only rules and rule histories.

Another relatively easy capability to add is a rule-transfer superprocedure that helps knowledge engineers to make new rules.

Suppose, for example, that you are a knowledge engineer and that you are working on new rules to be used by the ZOOKEEPER system introduced in Chapter 7. Further suppose that you have determined that another rule is needed, one that captures the idea that an animal is a carnivore if it is seen stalking another animal.

The knowledge engineer therefore proposes the following new rule:

Z16a If ?x stalks a different kind of animal
then ?x is a carnivore

There is evidence an antecedent is missing, however. When compared with the other rules that conclude that an animal is a carnivore, this proposed rule lacks an antecedent requiring that the animal is a mammal. Noting this lack, it makes sense to ask whether the omission is intended. The answer would lead to a refined rule:

Z16b If ?x is a mammal
?x stalks a different kind of animal
then ?x is a carnivore

Suppose Robbie creates PROCRUSTES, a knowledge-acquisition assistant, to automate the search for omissions.

To encourage knowledge engineers to construct new rules that look like old ones, PROCRUSTES makes heavy use of natural rule groups. It forms these natural rule groups by filtering all existing rules down through a tree like the one shown in figure 8.5.

For each rule group, PROCRUSTES forms a typical member by combining all antecedents and all consequents that occur in, say, 30 percent of the group's rules. For the group of those rules that conclude that an animal is a carnivore, there is one such antecedent (that the animal is a mammal), and one such consequent (that the animal is a carnivore).

Typical-carnivore
If the animal is a mammal
then it is a carnivore

All PROCRUSTES needs to do, then, is to compare proposed rules with the typical rules of the applicable rule groups, asking the knowledge engineer for a decision whenever something in a typical rule is not in a proposed rule. In the example, the carnivore rule group is applicable. Because the only antecedent of the typical carnivore rule is missing from the proposed rule, Z16a, PROCRUSTES would suggest that antecedent, leading to the more reliable rule, Z16b.

Thus, procedures such as PROCRUSTES can do a lot by *helping you to transfer* rulelike knowledge from knowledge engineers to rule-based systems. In Chapter 17, you learn about another procedure that can do even more by *directly producing* rulelike knowledge from precedents and exercises.

Rule Interactions Can Be Troublesome

It would seem that rule-based systems allow knowledge to be tossed into systems homogeneously and incrementally without concern for relating new knowledge to old. Evidently, rule-based systems should permit knowledge engineers to focus their attention on the rules, letting the rule chainer control rule interactions.

There are problems about which to worry, however. One particular problem is that the *advantage* of bequeathing control becomes the *disadvantage* of losing control, as King Lear failed to foresee.

Rule-Based Systems Can Behave Like Idiot Savants

Rule-based systems do some things so well, they can be said to be **savants** with respect to those things:

- Rule-based systems solve many problems.
- Rule-based systems answer simple questions about how they reach their conclusions.

Still, basic rule-based systems lack many of the characteristics of domain experts, qualifying them to be *idiot savants*:

- They do not reason on multiple levels.
- They do not use constraint-exposing models.
- They do not look at problems from different perspectives.
- They do not know how and when to break their own rules.

- They do not have access to the reasoning behind their rules.

In principle, there is nothing to prevent building more humanlike systems using rules, because rules can be used as a sort of programming language. When used as a programming language, however, rules have to be measured against alternative programming languages; when so measured, rules have little to offer.

RULE-BASED SYSTEMS VIEWED AS MODELS FOR HUMAN PROBLEM SOLVING

Do computational theories of problem solving have promise as psychological models of human reasoning? The answer is yes, at least to many computationally-oriented psychologists who try to understand ordinary human activity using metaphors shared with researchers who concentrate on making computers smarter.

Rule-Based Systems Can Model Some Human Problem Solving

In the human-modeling world, if–then rules are called **productions** and rule-based systems are called **production systems**. Hard-core rule-based-system enthusiasts believe that human thinking involves productions that are triggered by items in **short-term memory**. They also believe that short-term memory is inhabited by only a few simple symbolic **chunks**, whereas **long-term memory** holds all the productions. Specific combinations of the short-term memory items trigger the long-term memory's productions.

Protocol Analysis Produces Production-System Conjectures

To learn how people solve problems, information-processing psychologists pore over transcripts of subjects talking their way through problems. These transcripts are called **protocols**. The psychologists who study protocols generally think in terms of two important concepts:

- The **state of knowledge** is what the subject knows. Each time the subject acquires knowledge through his senses, makes a deduction, or forgets something, the state of knowledge changes.
- The **problem-behavior graph** is a trace of a subject moving through states of knowledge as he solves a problem.

The problem-behavior graph is important because it helps you to unravel characteristics of the subject who produces it. By analyzing the way one state of knowledge becomes another, you can draw inferences about the productions that cause those state changes. Consider, for example, the following protocol fragment:

A Protocol for Robbie

Let's see, I want to identify this animal, I wonder if it's a cheetah—I have nothing to go on yet, so I think I'll start by looking into the possibility that it is a carnivore—I had better check to see if it is a mammal first—yes, that is OK—hair—it doesn't seem to be eating anything so I can't tell if it is a carnivore—but oh, yes, it has pointed teeth, claws, and forward-pointing eyes, so it is a carnivore all right—now where was I—it's a carnivore, and I also see that it has a tawny color and dark spots—surely it must be a cheetah!

It would seem that assertions accumulate in the following order: hair, mammal, pointed teeth, claws, forward-pointing eyes, carnivore, tawny color, dark spots, and cheetah. From observations such as these, the information-processing psychologists would probably deduce that Robbie's production system is a backward-chaining one with rules much like those used in the ZOOKEEPER examples.

SOAR Models Human Problem Solving, Maybe

In artificial intelligence, an **architecture** is an integrated collection of representations and methods that is said to handle a specified class of problems or to model insightfully a form of natural intelligence.

SOAR[†] is an elaborate, production-centered architecture that was developed both to explain human problem solving and to support the implementation of applied systems. SOAR features a long-term memory for productions and a short-term memory for items that trigger the productions.

In the rest of this section, you learn about SOAR's key architectural features, but you learn only a bit, as a full treatment would require a book, not a section of a chapter.

SOAR Searches Problem Spaces

First, SOAR developers are committed to the idea that all problems can be formulated in terms of a search through a **problem space**, a net consisting of situation-describing **problem states**. SOAR starts from an initial situation, the **current state**, in the expectation that it will arrive at an identifiable **goal state** eventually. One such net is shown in figure 8.6.

SOAR moves from problem state to problem state in three-step cycles: First, SOAR establishes a **preference net**, a net in which preference labels and preference links describe the merits of various choices. Second, SOAR translates the preference labels and preference links into dominance relations among the states. Third, SOAR uses the dominance relations to select

[†] "SOAR" is an acronym for state, operator, and result, which together constitute one basic search step in SOAR. "SOAR" is an obscure acronym, however; many SOAR researchers cannot tell you its derivation.

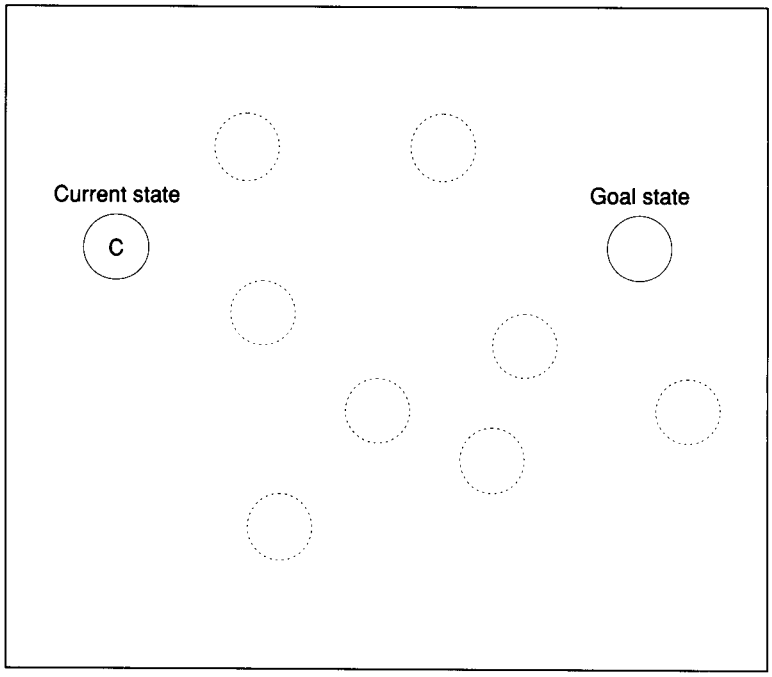

Figure 8.6 A problem space. At this point in the search, there are no links to indicate which problem states are considered neighbors.

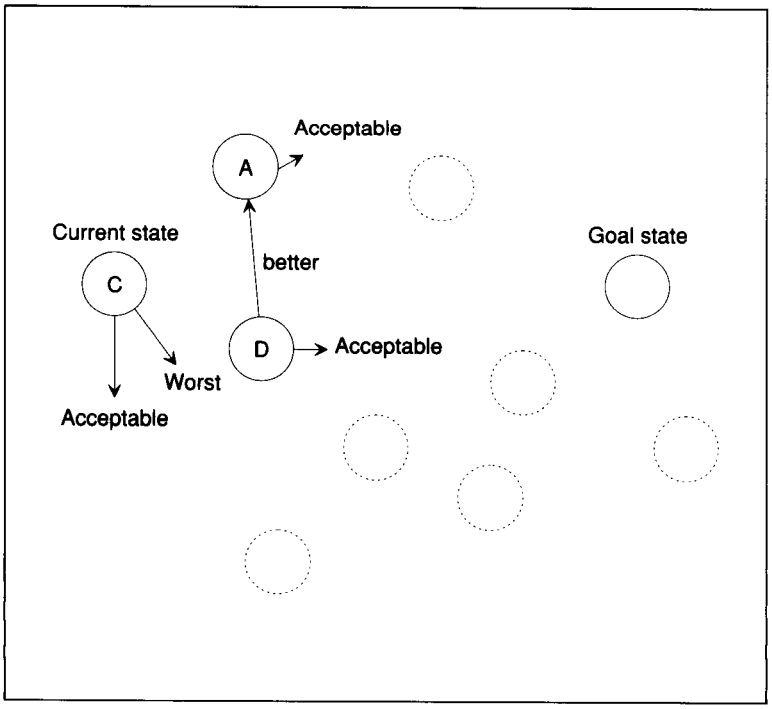

Figure 8.7 A preference net uses preference labels and preference links to describe the absolute and relative merits of a state and its neighbors.

the next current state. Then, SOAR repeats the cycle, placing new links, determining new dominance relations, and updating the current state, until SOAR reaches the goal state.

One example of a preference net is shown in figure 8.7. State C is the current state. The links labeled *acceptable*, *worst*, and *better* carry preference information. More generally, the links that can appear in a preference net are given in the following specification:

A **preference net** is a representation

That is a state space

In which

- ▷ Absolute preferences are identified by Is links that connect states to *acceptable*, *rejected*, *best*, and *worst* nodes.
- ▷ Relative preferences are identified by *better*, *worse*, and *indifferent* links that connect states to each other.

SOAR Uses an Automatic Preference Analyzer

To use preference labels and links, SOAR uses an **automatic preference analyzer**—one that resolves inconsistencies caused by rules with limited vision.

To see how SOAR's automatic preference analyzer works, consider the nodes and links shown in figure 8.7. SOAR's automatic preference analyzer first collects problem states C, A, and D, as those are labeled *acceptable*. Next, SOAR establishes that state D dominates state A because there is a *better* link from state D to state A. Then, SOAR establishes that state C is dominated by both of the other states because state C is labeled *worst* and it does not dominate any other state.

At this point, all dominated states are rejected, leaving only state D. Consequently, SOAR selects state D to be the next problem state, leaving SOAR ready to start another cycle of preference marking, preference analysis, and state selection.

While discovering new problem states and labeling known problem states with preference information, SOAR uses no conflict-resolution strategy to decide which triggered rule should fire. Instead, SOAR fires all triggered rules and decides what to do by looking at what they all do, rather than by using some rigidly prescribed, result-ignorant conflict-resolution strategy that suppresses all but one of the triggered rules.

On a higher level, in addition to determining which new problem state should replace the current state, SOAR's automatic preference analyzer looks for opportunities to reformulate the problem in terms of a better set of problem states in which to search for the goal, thereby replacing the current **problem space**. On a lower level, SOAR's automatic preference

analyzer looks for opportunities to replace the method, called the **current operator**, that discovers new problem states.

Finally, whenever SOAR gets stuck—finding no unambiguous way to replace any current problem space, problem state, or operator—SOAR's automatic preference analyzer announces an impasse, and SOAR's **universal subgoaling mechanism** creates a subgoal to resolve the impasse. SOAR users are expected to anticipate various sorts of problem-specific impasses so as to provide the appropriate productions for setting up subgoal-handling problem spaces, problem-space–dependent problem states, and problem-state–dependent operators.

In summary, the detailed behavior of SOAR's automatic preference analyzer is dictated by the following procedure:

To determine the preferred state using SOAR's automatic preference analyzer,

- ▷ Collect all the states that are labeled *acceptable.*
- ▷ Discard all the acceptable states that are also labeled *rejected.*
- ▷ Determine dominance relations as follows:
  - ▷ State A dominates state B if there is a *better* link from A to B but no *better* link from B to A.
  - ▷ State A dominates state B if there is a *worse* link from B to A but no *worse* link from A to B.
  - ▷ A state labeled *best*, and not dominated by another state, dominates all other states.
  - ▷ A state labeled *worst*, which does not dominate any other state, is dominated by all other states.
- ▷ Discard all dominated states.
- ▷ Select the next state from among those remaining as follows:
  - ▷ If only one state remains, select it.
  - ▷ Otherwise, if no states remain, select the current state, unless it is marked *rejected.*
  - ▷ Otherwise, if all the remaining states are connected by *indifferent* links,
    - ▷ If the current state is one of the remaining states, keep it.
    - ▷ Otherwise, select a state at random.
  - ▷ Otherwise, announce an impasse.

SUMMARY

- The simplicity of rule-based deduction systems enables you to build extremely useful modules on top of a basic chaining procedure.
- Explanation modules explain reasoning by using a goal tree to answer *how* and *why* questions. Probability modules help you to determine answer reliability. Acquisition modules assist knowledge engineers in knowledge transfer from a human expert to a collection of rules.
- Variable-precision reasoning systems exhibit variable reasoning styles, ranging from quick and superficial to slow and careful.
- Two key heuristics enable knowledge engineers to acquire knowledge from human experts. One is to work with specific situations; another is to ask about situation pairs that look identical, but are handled differently.
- Rule-based systems can behave like idiot savants. They do certain tasks well, but they do not reason on multiple levels, they do not use constraint-exposing models, they do not look at problems from different perspectives, they do not know how and when to break their own rules, and they do not have access to the reasoning behind their rules.
- Rule-based systems can model some human problem solving. SOAR, the most highly developed rule-based model of human problem solving, uses an automatic preference analyzer to determine what to do, instead of using a fixed conflict-resolution strategy.

BACKGROUND

The discussion of reasoning styles is based on the work of Ryszard S. Michalski and Patrick H. Winston [1986]. The discussion of PROCRUSTES is based on the work of Randall Davis [1976].

A system for propagating certainties is discussed in Chapter 11.

SOAR was developed by Allen Newell and his students [Laird et al. 1987; Newell 1990].