Procedural versus Declarative Knowledge
Forward versus Backward Reasoning
Backward-Chaining Rule Systems
Forward-Chaining Rule Systems
Matching
Use of Non Back Track

Unit 5

Rule Based Systems

Learning Objectives

After reading this unit you should appreciate the following:

- **Procedural Versus Declarative Knowledge**
- **Forward Reasoning**
- **Backward Reasoning**
- **Conflict Resolution**
- **Use of Non Backtrack**

Top

Procedural versus Declarative Knowledge

Preliminaries of Rule-based systems may be viewed as use of logical assertions within the knowledge representation.

A *declarative representation* is one in which knowledge is specified, but the use to which that knowledge is to be put is not given. A declarative representation, we must augment it with a program that specifies what is to be done to the knowledge and how. For example, a set of logical assertions can be combined with a resolution theorem prover to give a complete program for solving problems. There is a different way, though, in which logical assertions can be viewed, namely as a *program,* rather than as *data* to a program. In this view, the implication statements define the legitimate reasoning paths and the atomic assertions provide the starting points (or, if we reason backward, the ending points) of those paths.

A *procedural representation* is one in which the control information that is necessary to use the knowledge is considered to be embedded in the knowledge itself. To use a procedural representation, we need to augment it with an interpreter that follows the instructions given in the knowledge.

Screening logical assertions as code is not a very essential idea, given that all programs are really data to other programs that interpret (or compile) and execute them. The real difference between the declarative and the procedural views of knowledge lies in where control information resides. For example, consider the knowledge base:

man (Marcus)

man (Caesar)

person(Cleopatra)

$\forall x$: man (x)→ person(x)

Now consider trying to extract from this knowledge base the answer to the question

$\exists y$: person(y)

We want to bind *y* to a particular value for which *person* is true. Our knowledge base justifies any of the following answers:

y = Marcus

y = Caesar

y = Cleopatra

For the reason that there is more than one value that satisfies the predicate, but only one value is needed, the answer to the question will depend on the order in which the assertions are examined during the search for a response.

Of course, nondeterministic programs are possible. So, we could view these assertions as a nondeterministic program whose output is simply not defined. If we do this, then we have a "procedural" representation that actually contains no more information than does the "declarative" form. But most systems that view knowledge as procedural do not do this. The reason for this is that, at least if the procedure is to execute on any sequential or on most existing parallel machines, some decision must be made about the order in which the assertions will be examined.

There is no hardware support for randomness. So if the interpreter must have a way of-deciding, there is no real reason not to specify it as part of the definition of the language and thus to define the meaning of any particular program in the language. For example, we might specify that assertions will be examined in the order in which they appear in the program and that search will proceed depth-first, by which we mean that if a new subgoal is established then it will be pursued immediately and other paths will only be examined if the new one fails. If we do that, then the assertions we gave above describe a program that will answer our question with

y = Cleopatra

To see clearly the difference between declarative and procedural representations, consider the following assertions:

man(Marcus)

man(Caesar)

∀x : man(x) → person(x)

person(Cleopatra)

Viewed declaratively, this is the same knowledge base that we had before. All the same answers are supported by the system and no one of them is explicitly selected. But viewed procedurally, and using the control model we used to get *Cleopatra* as our answer before, this is a different knowledge base since now the answer to our question is *Marcus.* This happens because the first statement that can achieve the *person* goal is the inference rule

$\forall x: man(x) \rightarrow person(x)$

This rule sets up a subgoal to find a man. Again the statements are examined from the beginning, and now *Marcus* is found to satisfy the subgoal and thus also the goal. So *Marcus* is reported as the answer.

It is important to keep in mind that although we have said that a procedural representation encodes control information in the knowledge base, it does so only to the extent that the interpreter for the knowledge base recognizes that control information. So we could have gotten a different answer to the *person* question by leaving our original knowledge base intact and changing the interpreter so that it examines statements from last to first (but still pursuing depth-first search). Following this control regime, we report *Caesar* as our answer.

There has been a great deal of disagreement in AI over whether declarative or procedural knowledge representation frameworks are better. There is no clear-cut answer to the question. As you can see from this discussion, the distinction between the two forms is often very fuzzy. Rather than try to answer the question of which approach is better, what we do in the rest of this chapter is to describe ways in which rule formalisms and interpreters can be combined to solve problems. We begin with a mechanism called *logic programming,* and then we consider more flexible structures for rule-based systems.

Student Activity 5.1

Before reading next section, answer the following questions.

1. What is the difference between procedural knowledge and declarative knowledge?
2. Represent the following facts in predicate logic:

   a. All men are people.

   b. She eats everything Bill eats.

   c. If there is a road from city B to C then there must be a road from city C to B.

If your answers are correct, then proceed to the next section.

Top

Forward versus Backward Reasoning

Problems in AI can be handled in two of the available ways:

- Forward, from the start states
- Backward, from the goal states, which is used in PROLOG as well.

Taking into account the problem of solving a particular instance of the 8-puzzle. The rules to be used for solving the puzzle can be written as shown in Figure 5.1.

Reason forward from the initial states. Begin building a tree of move sequences that might be solutions by starting with the initial configuration(s) at the root of the tree. Generate the next

Assume the areas of the tray are numbered:

1	2	3
4	5	6
7	8	9

Square 1 empty and Square 2 contains tile n →
Square 2 empty and Square 1 contains tile n
Square 1 empty and Square 4 contains tile n →
Square 4 empty and Square 1 contains tile n
Square 2 empty and Square 1 contains tile n →
Square 1 empty and Square 2 contains tile n

Figure 5.1: A Sample of the Rules for Solving the 8-Puzzle

level of the tree by finding all the rules whose *left* sides match the root node and using their right sides to create the new configurations. Generate the next level by taking each node generated at the previous level and applying to it all of the rules whose left sides match it. Continue until a configuration that matches the goal state is generated.

Reason backward from the goal states. Begin building a tree of move sequences that might be solutions by starting with the goal configuration(s) at the root of the tree. Generate the next level of the tree by finding all the rules whose *right* sides match the root node. These are all the rules that, if only we could apply them, would generate the state we want. Use the left sides of the rules to generate the nodes at this second level of the tree. Generate the next level of the tree by taking each node at the previous level and finding all the rules whose right sides match it. Then use the corresponding left sides to generate the new nodes. Continue until a node that matches the initial state is generated. This method of reasoning backward from the desired final state is often called *goal-directed reasoning.*

To reason forward, the left sides are matched against the current state and the right sides (the results) are used to generate new nodes until the goal is reached. To reason backward, the right sides are matched against the current node and the left sides are used to generate new nodes representing new goal states to be achieved. This continues until one of these goal states is matched by an initial state.

In the case of the 8-puzzle, it does not make much difference whether we reason , forward or backward; about the same number of paths will be explored in either case. But this is not always true. Depending on the topology of the problem space, it may be significantly more efficient to search in one direction rather than the other. Four factors influence the question of whether it is better to reason forward or backward:

- Are there more possible start states or goal states? We would like to move from the smaller set of states to the larger (and thus easier to find) set of states.
- In which direction is the branching factor (i.e., the average number of nodes that can be reached directly from a single node)? We would like to proceed in the direction with the lower branching factor.
- Will the program be asked to justify its reasoning process to a user? If so, it is important to proceed in the direction that corresponds more closely with the way the user will think.
- What kind of event is going to trigger a problem-solving episode? If it is the arrival of a new fact, forward reasoning makes sense. If it is a query to which a response is desired, backward reasoning is more natural.

We may as well consider a few practical examples that make these issues clearer. Have you ever noticed that it seems easier to drive from an unfamiliar place home than from home to an unfamiliar place. The branching factor is roughly the same in both directions. But for the purpose of finding our way around, there are many more locations that count as being home than there are locations that count as the unfamiliar target place. Any place from which we know how to get home can be considered as equivalent to home. If we can get to any such place, we can get home easily. But in order to find a route from where we are to an unfamiliar place, we pretty much have to be already at the unfamiliar place. So in going toward the unfamiliar place, we are aiming at a much smaller target than in going home. This suggests that if our starting position is home and our goal position is the unfamiliar place, we should plan our route by reasoning backward from the unfamiliar place.

On the other hand, consider the problem of symbolic integration. The problem space is the set of formulas, some of which contain integral expressions. The start state is a particular formula containing some integral expression. The desired goal state is a formula that is equivalent to the initial one and that does not contain any integral expressions. So we begin with a single easily identified start state and a huge number of possible goal states. Thus to solve this problem, it is

better to reason forward using the rules for integration to try to generate an integral-free expression than to start with arbitrary integral-free expressions, use the rules for differentiation, and try to generate the particular integral we are trying to solve. Again we want to head toward the largest target; this time that means chaining forward. These two examples have illustrated the importance of the relative number of start states to goal states in determining the optimal direction in which to search when the branching factor is approximately the same in both directions. When the branching factor is not the same, however, it must also be taken into account.

Consider again the problem of proving theorems in some particular domain of mathematics. Our goal state is the particular theorem to be proved. Our initial states are normally a small set of axioms. Neither of these sets is significantly bigger than the other. But consider the branching factor in each of the two directions, from a small set of axioms we can derive a very large number of theorems. On the other hand, this large number of theorems must go back to the small set of axioms. So the branching factor is significantly greater going forward from the axioms to the theorems than it is going backward from theorems to axioms. This suggests that it would be much better to reason backward when trying to prove theorems. Mathematicians have long realized this, as have the designers of theorem-proving programs.

The third factor that determines the direction in which search should proceed is the need to generate coherent justifications of the reasoning process as it proceeds. This is often crucial for the acceptance of programs for the performance of very important tasks. For example, doctors are unwilling to accept the advice of a diagnostic program that cannot explain its reasoning to the doctors' satisfaction. This issue was of concern to the designers of MYCIN, a program that diagnoses infectious diseases. It reasons backward from its goal of determining the cause of a patient's illness. To do that, it uses rules that tell it such things as "If the organism has the following set of characteristics as determined by the lab results, then it is likely that it is organism x. By reasoning backward using such rules, the program can answer questions like "Why should I perform that test you just asked for?" with such answers as "Because it would help to determine whether organism x is present." By describing the search process as the application of a set of production rules, it is easy to describe the specific search algorithms without reference to the direction of the search.

We can also search both forward from the start state and backward from the goal simultaneously until two paths meet somewhere in between. This strategy is called *bidirectional search*. It seems appealing if the number of nodes at each step grows exponentially with the number of steps that

have been taken. Empirical results suggest that for blind search, this divide-and-conquer strategy is indeed effective. Unfortunately, other results, de Champeau and Sint suggest that for informed, heuristic search it is much less likely to be so. Figure 5.2 shows why bidirectional search may be ineffective. The two searches may pass each other, resulting in more work than it would have taken for one of them, on Its own, to have finished.

However, if individual forward and backward steps are performed as specified by a program that has been carefully constructed to exploit each in exactly those situations where it can be the most profitable, the results can be more encouraging. In fact, many successful AI applications have been written using a combination of forward and backward reasoning, and most AI programming environments provide explicit support for such hybrid reasoning.

Although in principle the same set of rules can be used for both forward and backward reasoning, in practice it has proved useful to define two classes of rules, each of which encodes a particular kind of knowledge.

- Forward rules, which encode knowledge about how to respond to certain input configurations.
- Backward rules, which encode knowledge about how to achieve particular goals.
- By separating rules into these two classes, we essentially add to each rule an additional piece of information, namely how it should be used in problem solving.

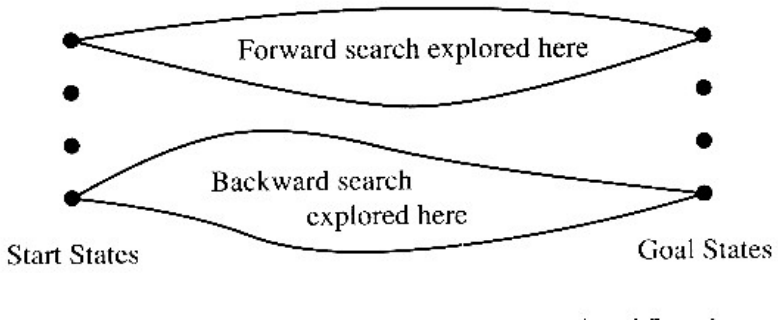

Figure 5.2: A Bad Use of Heuristic Bi-directional Search

Top

Backward-Chaining Rule Systems

Backward-chaining rule systems, of which PROLOG is an example, are good for goal- directed problem solving. For example, a query system would probably use backward chaining to reason about and answer user questions.

In PROLOG, rules are restricted to Horn clauses. This allows for rapid indexing because all of the rules for deducing a given fact share the same rule head. Rules are matched with the unification procedure. Unification tries to find a set of bindings for variables to equate a (sub) goal with the head of some rule. Rules in a PROLOG program are matched in the order in which they appear.

Other backward-chaining systems allow for more complex rules. In MYCIN, for example, rules can be augmented with probabilistic certainty factors to reflect the fact that some rules are more reliable than others.

Top

Forward-Chaining Rule Systems

In forward-chaining systems, left sides of rules are matched against the state description. Rules that match dump their right-hand side assertions into the state, and the process repeats.

Matching is typically more complex for forward-chaining systems than backward ones. For example, consider a rule that checks for some condition in the state description and then adds an assertion. After the rule fires, its conditions are probably still valid, so it could fire again immediately. However, we will need some mechanism to prevent repeated firings, especially if the state remains unchanged.

While simple matching and control strategies are possible, most forward-chaining systems implement highly efficient matchers and supply several mechanisms for preferring one rule over another.

Combining Forward and Backward Reasoning

Though many problems are solved using either of these techniques yet there are a few that are best handled via forward chaining and other aspects by backward chaining. Consider a forward-chaining medical diagnosis program. It might accept twenty or so facts about a patient's condition, then forward chain on those facts to try to deduce the nature and/or cause of the disease. Now suppose that at some point, the left side of a rule was *nearly* satisfied-say, nine out of ten of its preconditions were met. It might be efficient to apply backward reasoning to satisfy the tenth precondition in a directed manner, rather than wait for forward chaining to supply the fact by

accident. Or perhaps the tenth condition requires further medical tests. In, that case, backward chaining can be used to query the user.

Whether it is possible to use the same rules for both forward and backward reasoning also depends on the form of the rules themselves. If both left sides and right sides contain pure assertions, then forward chaining can match assertions on the left side of a rule and add to the state description the assertions on the right side. But if arbitrary procedures are allowed as the right sides of rules, then the rules will not be reversible. Some production languages allow only reversible rules; others do not.

When irreversible rules are used, then a commitment to the direction of the search must be made at the time the rules are written. But, as we suggested above, this is often a useful thing to do anyway because it allows the rule writer to add control knowledge to the rules themselves.

Top

Matching

Till now we have discussed little about how we extract from the entire collection of rules those that can be applied at a given point. To do so requires some kind of matching between the current state and the preconditions of the rules. How should this be done? The answer to this question can be critical to the success of a rule-based system. We discuss a few proposals below.

Indexing

One way to select applicable rules is to do a simple search through all the rules, comparing each one's preconditions to the current state and extracting all the ones that match. But there are two problems with this simple solution:

- In order to solve very interesting problems, it will be necessary to use a large number of rules. Scanning through all of them at every step of the search would be hopelessly inefficient.
- It is not always immediately obvious whether a rule's preconditions are satisfied by a particular state.

Sometimes there are easy ways to deal with the first of these problems. Instead of searching through the rules, use the current state as an index into the rules and select the matching ones immediately. For example, consider the legal-move generation rule for chess shown in Figure 5.3. To be able to access the appropriate rules immediately, all we need do is assign an index to each board position. Simply treating the board description as a large number can do this. Any reasonable hashing function can then be used to treat that number as an index into the rules. All the rules that describe a given board position will be stored under the same key and so will be

found together. Unfortunately, this simple indexing scheme only works because preconditions of rules match exact board configurations. Thus the matching process is easy but at the price of complete lack of generality in the statement of the rules. It is often better to write rules in a more general form, such as that shown in Figure 5.4. When this is done, such simple indexing is not possible. In fact, there is often a trade-off between the ease of writing rules (which is increased by the use of high-level descriptions) and the simplicity of the matching process (which is decreased by such descriptions).

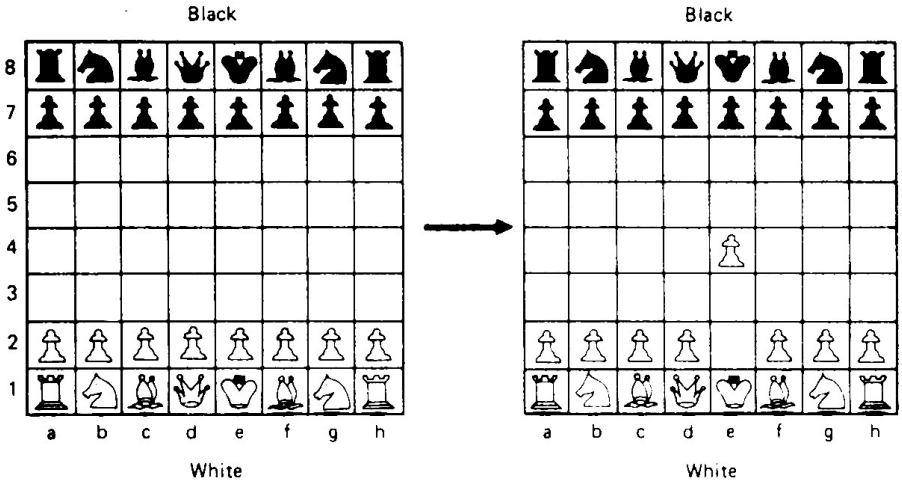

Figure 5.3: One Legal Chess Move

White pawn at
Square (file e, rank 2)
AND
Square (file e, rank 3)
is empty
AND
Square (file e, rank 4)
is empty

→ Move pawn from
Square (file e, rank 2)
to Square (file e, rank 4)

Figure 5.4: Another Way to Describe Chess Moves

All of this does not mean that indexing cannot be helpful even when the preconditions of rules are stated as fairly high-level predicates. In PROLOG and many theorem-proving systems, for example, rules are indexed by the predicates they contain, so all the rules that could be

applicable to proving a particular fact can be accessed fairly quickly. In the chess example, rules can be indexed by pieces and their positions. Despite some limitations of this approach, indexing in some form is very important in the efficient operation of rule-based systems.

Matching with Variables

The problem of selecting applicable rules is made more difficult when preconditions are not stated as exact descriptions of particular situations but rather describe properties (of varying complexity) that the situations must have. It often turns out that discovering whether there is a match between a particular situation and the preconditions of a given rule must itself involve a significant each process.

In many rule-based systems, we need to compute the whole set of rules that match the current state description. Backward-chaining systems usually use depth-first backtracking to select individual rules, but forward-chaining systems generally employ sophisticated *conflict resolution strategies* to choose among the applicable rules. While it is possible to apply unification repeatedly over the cross product of preconditions and state description elements, it is more efficient to consider the *many-many* match problems, in which many rules are matched against many elements in the state description simultaneously.

One efficient many-many match algorithms is RETE, which gains efficiency from three major sources:

The temporal nature of data. Rules usually do not alter the state description radically. Instead, a rule will typically add one or two elements, or perhaps delete one or two, but most of the state description remains the same. If a rule did not match in the previous cycle, it will most likely fail to apply in the current cycle. RETE maintains a network of rule conditions, and it uses changes in the state description to determine which new rules might apply (and which rules might no longer apply). Full matching is only pursued for candidates that could be affected by incoming or outgoing data.

Structural similarity in rules. Different rules may share a large number of pre-conditions. For example, consider rules for identifying wild animals. One rule *concludes jaguar [x] if mammal [x], feline[x], carnivorous [x],* and *has-spots[x].* Another rule concludes *tiger[x]* and is identical to the first rule except that it replaces *has-spots* with *has-stripes.* If we match the two rules independently, we will repeat a lot of work unnecessarily. RETE stores the rules so that they share structures in memory; sets of conditions that appear in several rules are matched (at most) once per cycle.

Persistence of variable binding consistency. While all the individual preconditions of a rule might be met, there may be variable binding conflicts that prevent the rule from firing. For example, suppose we know the facts son(*Mary, Joe*) and son(*Bill, Bob*). The individual preconditions of the rule

$$son(x, y) \wedge son(y, z) \rightarrow grandparent(x, z)$$

can be matched, but not in a manner that satisfied the constraint imposed by the variable y. Fortunately, it is not necessary to compute binding consistency from scratch every time a new condition is satisfied. RETE remembers its previous calculations and is able to merge new binding information efficiently.

For more details about the RETE match algorithm, see Forgy (1982). Other matching algorithms (e.g. Miranker (1987) and Oflazer (1987)) take different stands on how much time to spend on saving state information between cycles. They can be more or less efficient than RETE, depending on the types of rules written for the domain and on the degree of hardware parallelism available.

Complex and Approximate Matching

A more complex matching process is required when the preconditions of a rule specify required properties that are not stated explicitly in the description of the current state. In this case, a separate set of rules must be used to describe how some properties can be inferred from others.

An even more complex matching process is required if rules should be applied if their preconditions *approximately* match the current situation. This is often the case in situations involving physical descriptions of the world. For example, a speech-understanding program must contain rules that map from a description of a physical waveform to phones (instances of English phonemes, such as p or d). There is so much variability in the physical signal, as a result of background noise, differences in the way individuals speak, and so forth, that one can hope to find only an approximate match between the rule that describes an ideal sound and the input that describes an unideal world. Approximate matching is particularly difficult to deal with because as we increase the tolerance allowed in the match, we also increase the number of rules that will match, thus increasing the size of the main search process. But approximate matching is nevertheless superior to exact matching in situations such as speech understanding, where exact matching may often result in no rules being matched and the search process coming to a grinding halt. Although symbolic techniques for approximate matching exist, there is another, very different approach that can be used to solve this problem.

For some problems, almost all the action is in the matching of the rules to the problem state. Once that is done, so few rules apply that the remaining search is trivial. This was the case, for example, in ELIZA (Weizenbaum, 1966), an early AI program that simulated the behaviour of a Rogerian therapist. A fragment of a dialogue between ELIZA and a user is shown in Figure 5.5. ELIZA's knowledge about both English and psychology was coded in a set of simple rules. Figure 5.6 shows some ELIZA-like rules.

ELIZA is operated by matching the left sides of the rules against the user's last sentence and using the appropriate right side to generate a response. For example, if the user typed "My brother is mean to me," ELIZA might respond, "Who else in your family is mean to you?" or "Tell me more about your family." The rules were indexed by keywords so only a few had actually to be matched against a particular sentence. Some of the rules had no left side, so the rule could apply anywhere. These rules were used if no other rules matched and they generated replies such as "Tell me more about that." Notice that the rules themselves cause a form of approximate matching to occur. The patterns ask about specific words in the user's sentence. They do not need to match entire sentences. Thus a great variety of sentences can be matched by a single rule, and the grammatical complexity of English is pretty much ignored. This account both for ELIZA's major strength, its ability to say something fairly reasonable almost all of the time, and its major weakness, the superficiality of its understanding and its ability to be led completely astray. Approximate matching can easily lead to both these results.

Person	:	Men are all alike.
ELIZA	:	In what way?
Person	:	They're always bugging us about something or other.
ELIZA	:	Can you think of a specific example?
Person	:	Well, my boyfriend made me come here.
ELIZA	:	Your boyfriend made you come here.
Person	:	He says I'm depressed much of the time.
ELIZA	:	I am sorry to hear you are depressed.
Person	:	It's true. I am unhappy.
ELIZA	:	Do you think coming here will help you not to be unhappy?
Person	:	I need some help, that much seems certain.
ELIZA	:	What would it mean to you if you got some help?
Person	:	Perhaps I could learn to get along with my mother.
ELIZA	:	Tell me more about your family.

Figure 5.5: A Bit of a Dialogue with ELIZA

(X me Y)	→	(X you Y)
(I remember X)	→	(Why do remember X just now?)
(My (family-member) is Y)	→	(Who else in your family is Y?)
(X (family-member)Y)	→	(Tell me more about your family)

Figure 5.6: Some ELIZA-like Rules

One way of dealing with the frame problem is to avoid storing entire state descriptions at each node but instead to store only the changes from the previous node. If this is done, the matching process will have to be modified to scan backward from a node through its predecessors, looking for the required objects.

Conflict Resolution

The result of the matching process is a list of rules whose antecedents have matched the current state description along with whatever variable bindings were generated by the matching process. It is the job of the search method to decide on the order in which rules will be applied. But sometimes it is useful to incorporate some of that decision making into the matching process. This phase of the matching process is then called conflict resolution.

There are three basic approaches to the problem of conflict resolution in a production system:

- Assign a preference based on the rule that matched.
- Assign a preference based on the objects that matched.
- Assign a preference based on the action that the matched rule would perform.

Student Activity 5.2

Before reading next section, answer the following questions.

1. What factors determine whether it is better to reason forward or backward?
2. Give some examples where forward reasoning is better than backward reasoning.

If your answers are correct, then proceed to the next section.

Top

Use of Non Back Track

The real world is unpredictable, dynamic and uncertain. A robot cannot hope maintain a correct and complete description of the world. This mean that robot does not consider the trade-off between devising and executing plans. This trade-off has several aspects. For one thing, robot may not possess enough information about the world for it to do any useful planning. In this case, it mostly first engages in information gathering activity. Furthermore, once it begins executing a plan, the robot most continually monitor the results of its actions. If the result is unexpected, then re-planning may be necessary.

Since robots operate in the real world, so searching and backtracking is a costly affair. Consider an example of an AI-first search for moving furniture into a room, operating in a simulated world with full information. Preconditions of operators can be checked quickly, and if an operator fails to apply, another can be tried checking preconditions in the real world, however, can be time consuming if the robot does not have full information. These problems can be solved by the adopting the approach of non back tracking, it is cheaper approach then the searching and backtracking when robots work for real life problem.

Student Activity 5.3

Answer the following questions.

1. What is indexing process?
2. How conflict resolution is useful in matching?

Summary

- A *declarative representation* is one in which knowledge is specified, but the use to which that knowledge is to be put is not given.
- A *procedural representation* is one in which the control information that is necessary to use the knowledge is considered to be embedded in the knowledge itself.

- In PROLOG and many theorem-proving systems, rules are indexed by the predicates they contain, so all the rules that could be applicable to proving a particular fact can be accessed fairly quickly.
- The method of reasoning backward from the desired final state is called *goal-directed reasoning.*
- Backward-chaining systems usually use depth-first backtracking to select individual rules, but forward-chaining systems generally employ sophisticated *conflict resolution strategies* to choose among the applicable rules.

Self-assessment Questions

Fill in the blanks (Solved)

1. A _____ representation is one in which knowledge is specified but not its use.
2. Matching is typically more _____ for forward chaining systems than backward ones.

Answers

1. declarative
2. complex

True or False (Solved)

1. ELIZA was an early AI program that simulated the behaviour of a Georgian therapist.
2. PROLOG uses backward reasoning.

Answers

1. False
2. True

Fill in the blanks (Unsolved)

1. A _____ representation is one in which the control information that is necessary to use the knowledge is considered to be embedded in the knowledge itself.
2. _____ rules encode knowledge about how to respond to certain input configuration.
3. _____ rules encode knowledge about how to achieve particular goals.

True or False (Unsolved)

1. Approximate matching is superior to exact matching in situations such as speech understanding.
2. Assigning a preference based on the rule that matches, is not a basic approach to conflict resolution in a production system.
3. In forward-chaining systems, left sides of rules are matched against the state description.

Detailed Questions

1. A problem-solving search can proceed either forward or backward. Give some example of both types of problems.
2. If a problem-solving search program were to be written to solve each of the following types of problem, determine whether the search should proceed forward or backward:
    a. water jug problem
    b. blocks world
    c. natural language understanding.

1. A _____ representation is one in which the control information that is necessary to use the knowledge is considered to be embedded in the knowledge itself.
2. _____ rules encode knowledge about how to respond to certain input configuration.
3. _____ rules encode knowledge about how to achieve particular goals.

True or False (Unsolved)

1. Approximate matching is superior to exact matching in situations such as speech understanding.
2. Assigning a preference based on the rule that matches, is not a basic approach to conflict resolution in a production system.
3. In forward-chaining systems, left sides of rules are matched against the state description.

Detailed Questions

1. A problem-solving search can proceed either forward or backward. Give some example of both types of problems.
2. If a problem-solving search program were to be written to solve each of the following types of problem, determine whether the search should proceed forward or backward:
    a. water jug problem
    b. blocks world
    c. natural language understanding.