MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.034 Artificial Intelligence
Fall, 2011

Recitation 1, Thursday September 15

Rule-based systems: Forward and Backward Chaining Notes   Prof. Bob Berwick, 32D-728

Forward Chaining

You can think of the forward chaining process as that of filtering a (finite) set of *rules* to find the one that is applicable, then firing the rule, i.e., carrying out that rule's consequent, to change the state of the world. The state of the world is represented as a set of database *assertions* (DB), which are statements about what is true in the world. You might want to think about how 'deeply' the rules actually encode the state of knowledge about a particular situation, for example, the grocery bagging rules. How robust or fragile are they? How could you improve them?

General Forward Chaining Pseudo-code

1. For all rules and assertions, find all *matches*, that is, Rule+Assertion combinations. (To do this, you check a a rule's variables against all assertions in the DB to find bindings that make all the antecedents of a rule match one or more assertions in the database, thereby making the rule true).

2. Check if any of the matches are *defunct*.
A *defunct* match is one where the consequents from the match are already in the assertion database.

3. Fire the first non-defunct match & add its consequents to the assertion database. (Note that if there is more than one non-defunct rule, one *must* have some *conflict resolution* method for picking a single rule to fire. Here we have used the order in which rules have matched, following the way they are listed. But one might pick the *last* rule, or the *most recently used* rule, or… what else?)

4. Go to Step 1 and repeat until no more matches fire.

Note 1: If you always choose the first matched rule instance, it's terribly inefficient to compute the entire set of matched rule instances. (Suppose the system has many thousands of rules and assertions. Most of the time, only a few of these will be in play.) If you use a different conflict resolution technique, however, you might very well need to look at the entire list. Here we assume that all the matched rule instances are computed.

Note 2: If rules contain DELETE actions, then assertions may be removed from the DB, so matches in Step 2 may become 'un'-defuncted. This adds greatly to the power of such a system, as we remark below.

Deduction Rule Systems vs. Production Rule Systems

Note 3: A deduction system only adds assertions; a production system can both add and delete assertions. (So in class Winston called production systems 'general programming languages' – meaning that they can do anything that a general-purpose computer can do. Why do you think this is?)

A deduction rule system will always converge to the same result except in the case of STOP assertions and infinite loop situations. (STOP assertions generally are not considered part of deduction systems.) What this means is that in a deduction system, you could fire any number of matched rules before re-matching rules with assertions, as long as you have a mechanism for making sure that the same assertion isn't added to the database.

A production rule system will not always converge to the same result if the conflict resolution technique introduces randomness into the order of rule execution. Production rule systems can add explicit STOP assertions, which stop execution of the chainer, or DELETE assertions that would cause other rules to be matched.

A rule with a consequent that stops the chaining is an example of representing knowledge about the problem-solving process itself, a kind of 'meta-knowledge.' You might contrast this sort of rule with ones that represents knowledge about the state of the world. How would you add *learning* to such systems?

Backward Chaining

For this class we will always assume that our backward chainers are trying to prove the truth of a conclusion, also called a goal or hypothesis. In the process, they construct a so-called AND-OR *goal-tree*.

General Backward Chaining Pseudo-code

1. First try to find an assertion in the DB that matches the current goal; if one exists, exit with *true*.

2. Otherwise, we want to see what rules can produce the goal, by matching the consequents of those rules against the goal. All the consequents that match are possible options, so we collect the results together in an OR node. If there are no matches, the current goal is a leaf that is *false*. (We might have the system ask the user as to the truth or falsity of the goal in this case, but we don't do that here.)

3. For each matching consequent, keep track of the variables that are bound. Look up the antecedent of that rule, and instantiate those same variables in the antecedent, resulting in a new goal. The antecedent may have AND or OR expressions, and so form an AND-OR node subgoal tree as appropriate.

4. Recursively backward chain on each subtree. (Note that we can short-circuit the evaluation of AND or OR nodes: if any AND subgoal is false, the entire tree of subgoals it is a part of must be false, and the other subgoals do not need to be evaluated; we can return false immediately from this subgoal tree; conversely, if any OR subgoal is true, the entire tree of subgoals it is a part of must be true, and the other subgoals do not need to be evaluated; we can return true immediately from this subgoal tree.)

Note that:

- The backward chainer tries rules in the order they appear in the database of rules.
- Antecedents are tried in the order the antecedents appear in each rule.
- When backward chaining, a NOT clause matches if and only if there is NO matching assertion in the list of assertions, and rules that connect assertions in the list of assertions to the NOT clause. Question: what about rule conflict resolution in this situation?

Forward vs. Backward Chaining

Many rule systems can chain either forward or backward, but which direction is better? There are several rules-of-thumb.

Most importantly, you want to think about how the rules relate facts to conclusions. Whenever the rules are such that a typical set of facts can lead to many conclusions, your rule system exhibits a high degree of "fan-out," and argues for backward chaining. Whenever the rules are such that a typical hypothesis (i.e., a goal to be proven) can lead to many questions, your rule system exhibits a high degree of "fan-in" and argues for forward chaining. However, there are other considerations:

1. If the facts that you have or may establish may lead to a large number of conclusions, but the number of ways to reach the particular conclusion in which you are interested is small, then there is more fan-out than fan-in, and you should typically use backward chaining.

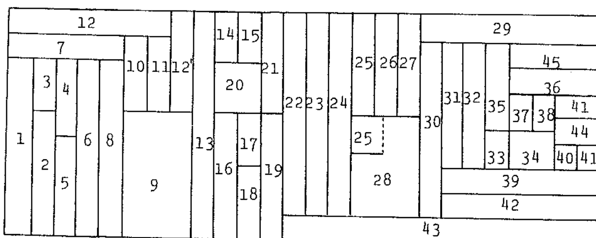
2. If the number of ways to reach the particular conclusion in which you are interested in is large, but the number of conclusions that you are likely to reach is small, then there is more fan-in than fan-out, and you should typically use forward chaining.

These are just rules-of-thumb. In many situations, however, neither fan-out nor fan-in dominates, leading to other considerations:

3. If you have not yet gathered any facts, and you are interested in only whether one of many possible conclusions is true, use backward chaining.

4. If you already have in hand all the facts you are ever going to get, and you want to know everything that can be concluded from those facts, use forward chaining.

Archeology: the first modern production rule system (circa 1949):



$$\text{S7.}\quad \begin{Bmatrix} \underline{La}_{\underline{k}} \\ Lb_{\underline{k}} \end{Bmatrix} LC_i(V_1+LC_{\underline{i}}) \longrightarrow \begin{Bmatrix} \emptyset \\ \underline{La}_{\underline{k}}+\underline{Lp}+LC_{\underline{i}} \end{Bmatrix} (V_1+\underline{La}_{\underline{k}}+LC_{\underline{i}})$$