32002: AI (Rule chaining) **Spring 2015**

Lecture 7: January 29, 2015

Lecturer: K.R. Chowdhary *: Professor of CS (VF)*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

7.1 Introduction

Symbolic rules are popular for knowledge representation and reasoning. Their popularity stems mainly from their naturalness, which facilitates comprehension of the represented knowledge. The basic form of a rule is,

$$if < conditions > then < conclusion >$$

where ¡conditions¿ represent the *conditions* or *premises* of a rule, and ¡conclusion¿ represent its conclusion or consequence. The conditions of a rule are connected between each other with logical connectives such as *AND/OR* thus forming a logical function. When sufficient conditions of a rule are satisfied, the conclusion is derived and the rule is said to *fire* (or *trigger*). Rules represent general knowledge regarding a domain.

7.2 RBSs

A Rule Based System consists of a knowledge base and an inference engine (figure 7.1). The knowledge base contains rules and facts. The interpretation of a rule means, "if the antecedent can be satisfied the consequent can too." When the consequent defines an action, the effect of satisfying the antecedent is to schedule the action for execution. When the consequent defines a conclusion, the effect is to infer the conclusion.

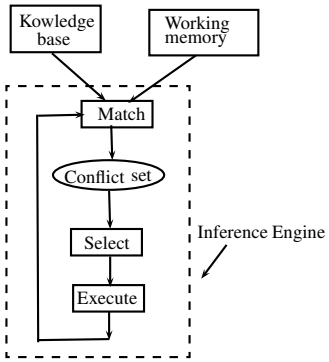

Figure 7.1: Inference Cycle.

Several key techniques for organizing RBSs have emerged. Rules can be used to express *deductive knowledge*, such as logical relationships, and thereby to support inference, verification, or evaluation tasks. Conversely, rules can be used to express goal-oriented knowledge that an RBS can apply in seeking problem solutions

and cite in justifying its own goal-seeking behavior. Finally, rules can be used to express *causal* relationships, which an RBS can use to answer "what-if" questions, or to determine possible causes for specified events.

In a rule based system, each *if* pattern may match to one or more of assertions in a collections of assertions. The collections of assertions is called *working-memory* (figure 7.1). The *then* patterns are assertions to be placed into the working memory. A system like this is called *deduction system*, as it deduces new inferences from the rules and assertions. The sets of rules may be like,

```
Ri:   if   if1
           if2
            ...
      then  then1
           then2
           ...
```

where R_i is name of the rule, or rule number. The assertions, i.e., working memory, matches with one or more 'if'patterns, like *if1*, *if2*, etc, and concludes the assertions, like *then1*, *then2*, etc. A more realistic example of a rule is:

```
R1: if 1. stiff neck,
       2. high temperature,
       3. impairment of conciousness occur togetehr,
    then
          meningitis is suspected.
```

In addition to its static memory for facts and rules, an RBS uses a *working-memory* to store temporary assertions. These assertions record earlier rule-based inferences. We can describe the contents of working memory as problem-solving state information. Ordinarily, the data in working memory adhere to the syntactic conventions of facts. Temporary assertions thus correspond to dynamic facts.

The basic function of an RBS is to produce results. The primary output may be - a problem solution, an answer to a question, or result of an analysis of some data. Whatever the case, an RBS employs several key processing determining its overall activity. A *world* manager maintains information in working memory, and a built-in control procedure defines the basic high-level loop; if the built-in control provides for programmable specialized control, an additional process manages branches to and returns from special control blocks.

Some times, the patterns specify the *actions* rather than *assertions*, e.g., "to put them on the table". In such case the rule based system is called *reaction system*.

In both the deduction systems and reaction systems, forward chaining is the process of moving from the *if* patterns to *then* patterns, where *if* patterns identifies the appropriate situation for deductions of new assertion, and performance of an action in the case of *reaction system*.

There are two broad kinds of rule system: *forward chaining* systems, and *backward chaining* systems, presented in the following sections.

7.3 Forward Chaining

In a forward chaining system, you start with the initial facts, and keep using the rules to draw new conclusions (or take certain actions), given those facts. Whenever an if pattern is observed to match an assertion, the

antecedent is *satisfied*. When all the *if* patterns of a rule are satisfied, the rule is *triggered*. When a triggered rule establishes a new assertion or performs an action, it is *fired*. This procedure is repeated until no more rules are applicable (figure 7.1).

The selection process is carried out in two steps:

1. *Pre-selection:* Determining the set of all executable rules, also called the *conflict-set*.

2. *Selection:* Selection of a rule from the conflict set by means of a conflict resolving strategy.

7.3.1 Forward chaining Algorithm

A simple forward chaining algorithm (1) starts from the known facts in the knowledge-base, and triggers all the rules whose premises are the known facts, then adds the consequent for each into the knowledge-base. This process is repeated until the query is answered or until there is no conclusion generated to be added into the knowledge-base. We will use symbols θ, λ, γ to represent unifiers, *unify* is a function which unifies the assertion q' and the query α, and returns a unifier λ if they are unified, else returns null.

The forward chaining algorithm (1) pucks-up every sentence $s \in \Gamma$, and checks all possible unifiers θ for s. It returns the goal when it is reached.

Algorithm 1 Forward-chaining(Input: Γ, α) // α is a query, Γ is knowledge-base

```
1: while True do
2:    new = {}
3:    for each sentence s ∈ Γ do
4:       convert s into the format p1 ∧ p2 ∧ ··· ∧ pn → q
5:       for each unifier θ such that (p1 ∧ p2 ∧ ..., ∧pn)θ = (p'1 ∧ p'2 ∧ ..., ∧p'n) for some p'i s ∈ Γ do
6:          q' = qθ
7:          new = new ∪ {q'}
8:          λ = unify(q', α)
9:          if λ is not null then
10:            return λ
11:         end if
12:      end for
13:      Γ = Γ ∪ new
14:   end for
15: end while
16: Return Fail
```

The following example demonstrates the working of forward chaining algorithm.

Example 7.1 *Produce the inference, given the knowledge-base* $\Gamma = \{man(x) \rightarrow mortal(x), man(socrates)\}$ *and query* $\alpha = mortal(w)$*, i.e., "who is mortal?"*

Solution. We apply the algorithm (1) manually, and note that $p_1 \wedge \cdots \wedge p_n = p_1 = man(x)$, $p'_1 = man(socrates)$, and $\theta = \{socrates/x\}$. Also,

$$\begin{aligned} q' &= q\theta \\ &= mortal(x)\{socrates/x\} \\ &= mortal(socrates) \end{aligned}$$

On unification of α (i.e, $mortal(w)$), and q', the unifier λ obtained is,

$$\begin{aligned} \lambda &= unify(q', \alpha) \\ &= unify(mortal(socrates), mortal(w)) \\ &= \{socrates/w\} \end{aligned}$$

Hence, the answer obtained is $w = socrates$. □

7.3.2 Conflict Resolution

When we are doing data-directed reasoning, we may not like to fire all of the rules in case more than one rule is applicable. In cases where we want to eliminate some applicable rules, some kind of conflict resolution is necessary for arriving at the most appropriate rule(s) to fire. In a deduction system all rules generally fire, but in a reaction system, when more than one rule are triggered at the same time, only one of the possible actions is desired.

Here are some other common approaches:

1. *Selection by order*
   - *Order:* Pick the first applicable rule in the order they have been presented.
   - *Recency:* Select an applicable rule based on how recently it has been used. There are different versions of this strategy, ranging from firing the rule that matches on the most recently created (or modified) wff to firing the rule that has been least recently used. The former could be used to make sure that a problem solver stays focused on what it was just doing (typical of depth-first search); the latter would ensure that every rule gets a fair chance to influence the outcome (typical of breadth-first search).
2. *Selection according to syntactic structure of the rule*
   - *Specificity:* Select the applicable rule whose conditions are most specific.
     i) *if* $bird(x)$ *then* $add(canfly(x))$
     ii) *if* $bird(x) \wedge weight(y, x) \wedge \text{gt}(y, 5kg)$ *then* $add(cannotfly(x))$
     iii) *if* $bird(x) \wedge penguin(x)$ *then* $add(cannotfly(x))$

     Here, the second and third rules are both more specific than the first.
   - Apply the syntactically largest rule, i.e., the rule which contains the most propositions.
3. *Selection by means of supplementary knowledge*

- Apply the rule with the highest priority. For this purpose each rule must be given a priority, which may be represented, e.g., by a number.

Considering the following rule, for which many conjunct ordering may be possible,

$$\forall x \forall y [p(x) q(x, y) \rightarrow r(x, a).]$$

Here, all the $p(x)$ may be ordered then find $q(x, y)$ for each x and y. Alternatively, for each x find $p(x)$, then find all $q(x, y)$ for different y. Which approach is better? Finding this is solution of a *conjective-ordering problem*, which is $NP - hard$. A heuristic can use most constrained, the one having fewest values. This shows a close relationship between *CSP* (constraint satisfaction problem), discussed in next chapters, is a pattern matching problem. Due to these complexity issues the forward-chaining algorithm is *NP-hard* in its inner loop.

However, since the size and arities are constant in the real-world, the complexity of expression (**??**) is a polynomial in nature.

One of the drawback of forward chaining is that it makes all allowable inferencing based on the facts and rules available, irrespective of whether they are relevant to deriving goals or not. This makes it very inefficient. To avoid this, we work for selected subclass of rules. Yet another approach is backward chaining, which arrives to only those premises which are leading to goal, as the system is goal driven.

7.4 Backward Chaining

While forward chaining allows conclusions to be drawn only from a given knowledge base, a backward-chained rule interpreter is suitable for requesting still unknown facts. In a backward chaining system, you start with some hypothesis (goal) you are trying to prove, and keep looking for rules that would allow you to conclude that hypothesis, perhaps setting new sub-goals to prove as you go.

For this, the goal expression is initially placed in the working memory. Followed with, this the system performs two operations:

1. matches the rule 'consequent'with the goal,
2. selects the matched rule and places its premises in the working memory.

The second in above corresponds to the transformation of the problem into subgoals. The process continues in the next iteration, with these premises becoming the new goals to match against the consequent of other rules. Thus, the backward chaining system, in the human sense are *hypothesis testing*.

Backward chaining uses *stack*. First, the goal is put in the stack, then all the rules which results to this goal are searched and put on the stack. These becomes the sub-goals, and the process goes on till all the facts are proved or are available as literals (ground clauses). Every time the goal and premises are decided, the unification is performed.

7.4.1 Backward Chaining Algorithm

The algorithm for backward chaining returns the set of substitutions (*unifier*) which makes the goal true. These are initially empty. The input to the algorithm is knowledge-base Γ, $goals\alpha$, and current substitution θ

(initially empty). The algorithm returns the substitution set λ for which the goal is inferred. The algorithm 2 shows the backward-chaining algorithm.

Algorithm 2 Backward-chaining(Input: Γ, α, θ) // α is a query, Γ is knowledge-base, θ current substitution (initially empty), λ represent substitution set, i.e., query result (initially empty).

1: $q' \leftarrow \alpha\theta$
2: **for** each sentence $s \in \Gamma$, where $s = p_1 \wedge p_2 \wedge \ldots, \wedge p_n \rightarrow q$ and $\gamma \leftarrow unify(q, q')$ and $\gamma \neq null$ **do**
3: $\quad \alpha_{new} \leftarrow (p_1 \wedge p_2 \wedge \cdots \wedge p_n)$
4: $\quad \theta \leftarrow \theta\gamma$
5: $\quad \lambda \leftarrow$backward-chaining$(\Gamma, \alpha_{new}, \theta) \cup \lambda$
6: **end for**
7: return λ

Example 7.2 *Apply the backward-chaining algorithm for inferencing from a given knowledge-base.*

Solution. Let $\Gamma = \{man(x) \rightarrow mortal(x), man(socrates)\}$. Assume that it is required to infer "Who is mortal?" That is, goal $\alpha = mortal(w)$. The loop iterations for the backward-chaining algorithm 2 are as follows:

1st Iteration: Initially, θ is empty, hence, $q^{'} = \alpha\theta = mortal(w)$. From algorithm and knowledge-base Γ, $s = man(x) \rightarrow mortal(x)$. Thus, $\gamma = unify(man(x), man(w)) = \{w/x\}$. Also, the new goal, $\alpha_{new} \leftarrow man(x)$. The new value of current substitution is, $\theta \leftarrow \theta\gamma = \{w/x\}$. Next, compute $\lambda = backward\text{-}chaining(\Gamma, man(x), \{w/x\}) \cup \lambda$.

2nd iteration: To compute the λ in second iteration, we apply the algorithm in a recursive mode, and get $q' = man(x)\{w/x\} = man(w)$. Next, $\gamma = unify(man(socrates), man(w))$. This results to $\gamma = \{socrates/w\}$, and is an inference. □