Chapter -6 (of book)

Representing knowledge using rules

Saathi

Date ___/___/_____

→ Now, we shall discuss the use of rules to encode knowledge. This is a particularly imp. issue since rule based reasoning systems have played a very imp. role in evolution of AI.

→ Now, we consider a set of rules to represent both knowledge about relationship in the world, as well as knowledge about how to solve problems using content of rules.

6.1 ★ Procedural vs. Declarative knowledge

→ A "declarative repre" " is one in which knowledge is specified, but the use of knowledge to be put is not given.

→ A "procedural repre" " is one in which the control info. that is necessary to use the knowledge is considered to be "embedded" in knowledge itself.

→ In "declarative repre" ", we must argue with a program that specifies what is to be done to the knowledge & how? For ex., a set of logical assertions can be combined with a resolution theorem prover to give a complete program for solving problems. Here, logical assertions can be viewed as a "program" rather than as "data" to a program. These reasoning paths define the possible execution paths in the same way like traditional control structure like if---then---else.

→ The real difference between the "declarative" & "procedural" views of knowledge lies in where control info. resides. For ex., consider the knowledge base:-

Page No.

Saathi

Date

1. man (Marcus)
2. man (Caesar)
3. person (Cleopatra)
4. $\forall x : man(x) \rightarrow person(x)$

Knowledge base

→ Now consider trying to extract from this knowledge base to answer 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 foll. answers :-

y = Marcus
y = Caesar
y = Cleopatra

→ Bcoz there is more than one value satisfies this predicate, but only one value is needed. So, the ans. to the question will depend on the order in which assertions are examined.

→ If we view these assertions as declarative, then they don't say anything about how they will be examined. If we view them as procedural, then they do.

Control info.

for ex, we might specify that assertions will be examined in order in which they appear in program then that search will proceed by depth - first.

→ for above sequence of assertions will answer our question with

y = clepatra.

To see difference bet^n declarative and procedural representation clearly consider the following assertions:

Man (Marcus)
Man (Caesar)
$\forall x$: man (x) → Person (x)
Person (Cleoputra)

new knowledge base

→ If we view declaratively then this is the same knowledge base that we had before because order does not matter.

→ But if we view above knowledge procedurally then our answer is "Marcus". This happens because $\forall x$: man (x) → Person (x) . this rule of inference achieves "person" goal first.

→ This rule sets up a goal to find "person" and subgoal to find "man". Again the statements are examined from beginning and now "Marcus" satisfies the subgoal to be "Man". So Marcus is reported as an answer.

Saathi

Date __/__/____

6.2 ☆ Logic programming

→ Logic programming is a programming language approach in which logical assertions are viewed as programs. There are several prog. languages used today and the most popular of which is PROLOG.

→ A "PROLOG" program is described as a series of logical assertions, each of which is a "Horn clause".

→ A "Horn clause" is a clause that has at most one positive literal. For ex., f, ¬p ∨ q and p→q are all "Horn clause", Since (p) is a positive literal in (f). (q) is a positive literal in (¬p ∨ q) and, as we know that (p→q) = (¬p ∨ q), so it also contain (q) as a positive literal. Therefore, (p→q) is also a "Horn clause."

p→q
¬p∨q

→ The fact that "PROLOG" programs are composed only of "Horn" clause and not of arbitrary logical expression due to two imp. reasons:
(1) Due to its uniform repre".
(2) The logic of Horn clause systems is decidable.

→ Fig. shows an example of a simple knowledge base represented in standard logical notation & then in PROLOG. Both of these repre" contain two types of statements i.e. "facts" which contain only constants & rules to contain variables.

→ "Facts" represent statements about specific objects, and

Page No.

Date __/__/____

"Rules" represent statements about classes of objects.

Fig

$\forall x: pet(x) \wedge small(x) \rightarrow apartment\,pet(x)$
$\forall x: cat(x) \vee dog(x) \rightarrow pet(x)$
$\forall x: poodle(x) \rightarrow dog(x) \wedge small(x)$
poodle(fluffy) — [illegible]

Repreⁿ in logic

```
apartment pet (x) :- pet(x), small(x)
pet(x) :- cat(x)
pet(x) :- dog(x)
small(x) :-
dog(x) :- poodle(x)
small(x) :- poodle(x)
poodle(fluffy).
```

Repreⁿ in PROLOG

→ In PROLOG "Horn" clauses are started on the right of the implication sign. That means, the literal after "→" will be written first.

→ There are several superficial ([illegible]), synatctic difference betⁿ the logic & the PROLOG repreⁿ :-

(1) In logic, variables are quantified. In PROLOG, quantification is provided implicitly by the way the variables are interpreted.

(2) In logic, there are explicit symbols for and(∧) and OR(∨). In PROLOG, there is an explicit symbol for ~~and~~ AND (,), but there is none for "OR". Since symbol for comma OR ના symbol નથી.

Date ___/___/_____

disjunction (∨) is not there, so it must be represented as a list of alternative statements, and anyone of which may provide the basis for a conclusion.

(3) In logic, implications of the form "p→q" "p implies q" are written as "p→q". In PROLOG, the same implication is written "backward" as, q :- p. This form of is natural in PROLOG, bcoz the interpreter always works backwards from a goal, and this form causes every rule to begin with the component that must be matched first. This first component is called the (head) of the rule.

→ The first two of these differences arise naturally from the fact that PROLOG-programs are actually sets of Horn clauses that are transformed as follows:-

(1) If the Horn clause contains a single literal which is positive, then leave it as it is.

(2) Otherwise, rewrite the Horn clause as an implication by combining all the negative literals of the implication and leaving the single positive literal as the consequent.

→ The PROLOG clause,

P(x) :- Q(x, y)

is equivalent to the logical expression,

∀x : ∃y : Q(x, y) → P(x)

→ Now, we know that P(x) :- Q(x, y) is written

Saathi

Date __/__/____

like $(Q(x,y) \rightarrow P(x))$. But, what about quantifiers like ∀ and ∃. This expression is equal to $\neg Q(x,y) \vee P(x)$. આપેલા expression માં જે negative literal હોય, તેમાં જે variable પહેલાથી આવે, જોઈએ કે 'y' તો existentially quantified (∃) થશે, જ્યારે જે variable પછીના આવે, અને head તરીકે હોય અને જોઈએ કે 'x' તો universally quantified (∀) થશે.

$$\therefore \boxed{\forall x : \exists y : Q(x,y) \rightarrow P(x)}$$

X → The basic PROLOG control strategy described above is simple.

(1) Begin with a problem statement, which is viewed as a goal to be proved.

(2) Look for the assertions that can prove the goal.

(3) Consider facts, which prove goal directly, and also consider any rule whose head matches the goal.

(4) To decide whether a fact or rule can be applied to the current problem, invoke a standard unification procedure.

(5) Reason backward from that goal until a path is found that terminates with assertion in the program.

(6) Consider paths using DFS strategy.

Example

Suppose we want to find a value of X that satisfies the predicate "apartment(X)". We state this goal to PROLOG as, ? - apartment(X).

Soln → Here 'apartment' predicate there as its head.

```
apartment(X) :- pet(X), small(X)
pet(X) :- cat(X)
pet(X) :- dog(X)
dog(X) :- poodle(X)
small(X) :- poodle(X)
poodle(fluffy)
```

Page No.

Date __/__/____

→ The "apartment" rule will succeed if ~~one of the~~ both of the clauses on its right hand side can be satisfied.

→ So, we have to try to probe each of them, that is "pet(x) & small(x).

→ They will be tried in the order in which they appear. There are rules for "pet" with it on the right hand side.

→ There are two rules, i.e "pet(x):-cat(x)" and "pet(x):- dog(x)" and anyone of these must be satisfied.

→ The first "pet(x):-cat(x)" will fail, bcoz there are no assertions about predicate "cat" in the program.

→ But, there's one assertion for "dog", that is "dog(x) :- poodle(x)". This succeeds using the rule about "dog" & "poodles" & using the fact "poodle(fluffy)".

→ This results in variable 'X' to be bound with "fluffy".

→ Now, second clause "small(x) must be checked, since "pet" & "small" were connected by "conjuction (AND)" operation.

Saathi

Date __/__/____

→ Now, again there's one assertion for "small", that is "small(x) :- poodle(x)". This also succeeds using rule about "small" & "poodle" & "poodle(fluffy)".

→ Therefore, small(fluffy) is also proved.

→ Now,

AND

appartment(x) :- pet(x) ; small(x)

Connected by AND (Conjuction) and both are proved for "fluffy". So, "appartment(fluffy)" is also proved.

Example completed

Example over here

Continue theory

→ Logical negation (¬) can't be represented explicitly in pure PROLOG. So, it is not possible to encode directly the logical assertion,

$$\forall x : dog(x) \rightarrow \neg cat(x)$$

→ Instead, negation is represented implicitly by the "lack" of an assertion. This leads to the problem-solving strategy called as "negation as failure".

Ex. ?- cat(fluffy)

This would return false, broz it is unable to prove that "Fluffy" is a cat.

Saathi

Date __/__/____

63 ☆ Forward vs. backward reasoning Q-12

→ The objective of search procedure is to discover a path through a problem space from initial configuration to a goal state.

→ While PROLOG only searches from a goal state, there are actually two directions in which a search could proceed:-

- Forward, from start states
- Backward, from goal states

→ Production system model of search process provides an easy way of viewing forward & backward reasoning as symmetric process.

- Reason forward from initial states:- Start building a tree of move sequences that might be solu"s starting with the initial configuration at root of the tree. Generate the next level of tree by finding all the rules whose left sides match the root node & using their right sides to create new configuration

- Reason backward from goal state:- Start building a move sequence by starting with the goal state at root of the tree. Generate the next level of tree by finding all the rules whose right sides match the root node. Use the left side of rules to generate the nodes at this second level of the tree. Generate next level of tree by ~~taking~~ finding all rules whose right sides match it. Continue until a node that matches the initial state. This method of reasoning backward from desired final state

Page No.

is often called "goal-directed reasoning."

→ Notice that the same rules can be used both to reason forward from initial state & to reason backward from goal state.

→ To reason forward, the left sides are matched against the current state & to reason backward, right sides are matched against the current state.

→ In case of 8-puzzle, it doesn't make much difference whether we reason forward or backward, since same no. of paths will be explored in both the cases. But this is not always true. It depends on topology of problem space and it may be more efficient to search in one direction rather than the other.

→ Four factor can affect, whether it is better to reason forward or backward:-

1. Are there more possible start states or goal states? We would like to move from smaller set of states to larger set of states.

2. In which direction, the branching factor is greater? "Branching factor" is avg. no. of nodes that can be reached directly from single node. We would like to proceed in direction with lower "branching factor."

3. Will the program be asked to justify its reasoning process to user? If so, then it's imp to proceed in direction in which user will think.

Saathi

Date __/__/____

→ For example, it seems easier to drive from an unfamiliar place to home than from home to an unfamiliar place. "Why like this?" The branching factor is almost same in both directions. But in order to find a route from home to an unfamiliar place, then it's too difficult, bcoz all nearby locations are also unknown. In opposite to this, if we want to travel from unfamiliar locaⁿ to home, then it's really easy, bcoz all nearby locaⁿs from home are known. This suggests that if our starting posⁿ is home & our goal posⁿ is the unknown place, then we should plan our route by reasoning backward from the unfamiliar place.

Example

→ On the other hand, consider the problem of symbolic integration. The "start state" is containing some integral expression. The desired "goal state" is the state that doesn't contain any integral expressions. So, we begin with a single identified start state & a huge possible goal states. Thus to solve this problem, it's better to reason forward using the rules for integration to generate "integraⁿ-free expression", rather than to start with integral-free expression. So, here we should move forward.

Example

⊛ forward & Backward Reasoning

Example

Knowledge Base:

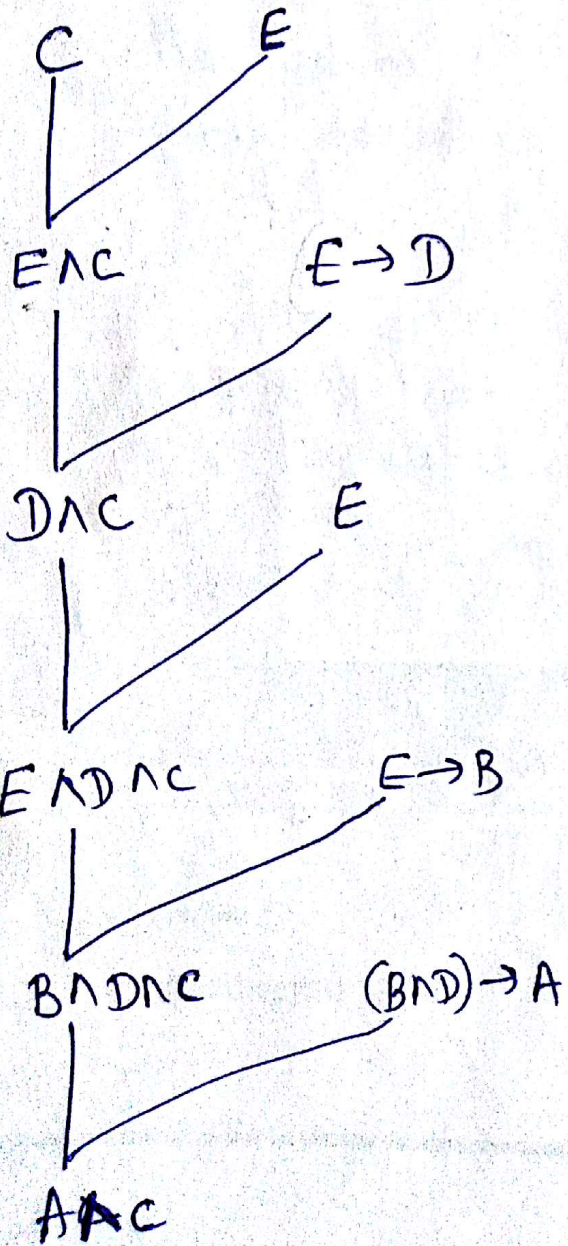
1) $E \rightarrow D$
2) $E \rightarrow B$
3) $(B \wedge D) \rightarrow A$
4) E
5) C

root for forward

root for Backward

Query = $A \wedge C$

forward chaining desivation



Backward Chaining desivation