ARTIFICIAL INTELLIGENCE

– Elaine Rich, Kevin Knight (2nd Edn.)

Chapter : 4 KNOWLEDGE REPRESENTATION ISSUES

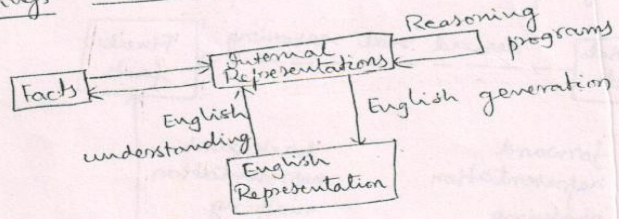
Representations and Mappings

To solve complex problems in A.I., we require (1) large amount of knowledge and (2) some mechanism to manipulate that knowledge to create solutions.

Usually, we deal with two types of entities.

(1) Facts : truths (things we want to represent)
(2) Representation of facts in some chosen formalism: (things we will actually manipulate)

Mappings between Facts and Representations



We have two levels:

(1) Knowledge level, at which facts are described.
(2) Symbol level, at which representations of objects at the knowledge level are defined in terms of symbols that can be manipulated by programs.

Ex: Fact : Spot is a dog

Internal representation : dog(Spot)

Fact : All dogs have tails.

Int. Rep. : $\forall x : dog(x) \rightarrow hastail(x)$

Int. Rep : hastail (Spot) (using deduction)

Fact : Spot has a tail.

Mapping functions are usually many-to-many.

Ex: ① All dogs have tails.
② Every dog has a tail.

①&② represent the same fact — 'every dog has at least one tail'

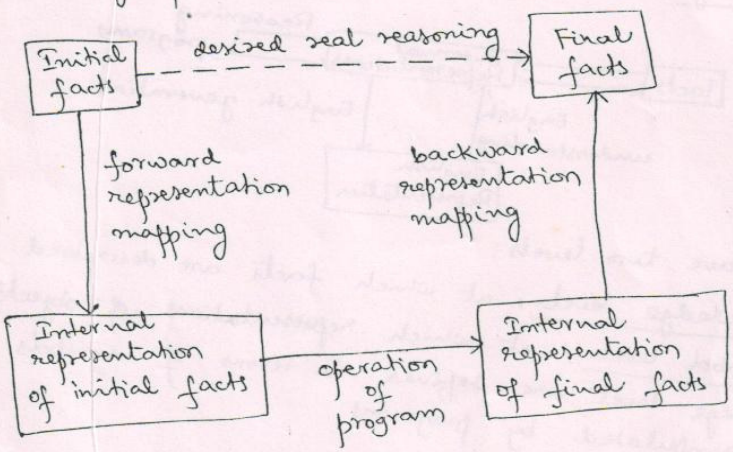
But ① could represent → Every dog has at least one tail
→ or Each dog has several tails.

② could represent → Every dog has at least one tail.
or → There is a tail that every dog has.

When we try to convert English sentences into some other representation (such as logical propositions), we must first decide what facts the sentences represent and then convert those facts into the new representation.

AI program manipulates the internal representation of the facts it is given. This manipulation should result in new structures that can also be interpreted as internal representation of facts. That is, these structures should be the internal representations of facts that correspond to the answer to the problem described by the starting set of facts.

Sometimes, a good representation makes the operation of a reasoning program not only correct but trivial.



Representation of facts

Approaches to knowledge representation:

There are four properties.

1) Representational Adequacy — the ability to represent all of the kinds of knowledge that are needed in that domain.
2) Inferential Adequacy — the ability to manipulate the representational structures in such a way as to derive new structures corresponding to new knowledge inferred from old.

Chapter 5 : USING PREDICATE LOGIC

Representing simple facts in Logic :

In propositional logic,

It is raining. It is sunny. It is windy.
RAINING. SUNNY. WINDY.

If it is raining, then it is not sunny
RAINING → ¬ SUNNY

Here the logical propositions are written as well-formed formulas (wff's).

Propositional logic has limitations.

Ex: Socrates is a man. Plato is a man.
we could write SOCRATESMAN. PLATOMAN.
But we can't draw any conclusion about the similarities here (between Socrates and Plato)

The better way : MAN(SOCRATES)
MAN(PLATO)

Ex: All men are mortal.
If we write MORTALMAN, it fails to capture the relationship between any individual being a man and that individual being a mortal. For this, we need variables and quantification.

So first-order predicate logic (simply, predicate logic) is used. Here, we can represent real-world facts as statements written as wff's.

Using Predicate logic to represent knowledge :

Ex: 1. Marcus was a man.
2. Marcus was a Pompeian.
3. All Pompeians were Romans.
4. Caesar was a ruler.
5. All Romans were either loyal to Caesar or hated him.
6. Everyone is loyal to someone.
7. People only try to assassinate rulers they are not-loyal to.
8. Marcus tried to assassinate Caesar.

In Predicate logic,

1. Marcus was a man.
man(Marcus)

2. Marcus was a Pompeian.
   Pompeian(Marcus)

3. All Pompeians were Romans.
   $\forall x$: Pompeian$(x) \rightarrow$ Roman(x)

4. Caesar was a ruler.
   ruler(Caesar).

5. All Romans were either loyal to Caesar or hated him.
   $\forall x$: Roman$(x) \rightarrow$ loyalto$(x,$ Caesar$) \vee$ hate$(x,$ Caesar$)$

6. Everyone is loyal to someone.
   $\forall x : \exists y :$ Loyalto(x,y)
   (for each person there exists someone to whom he or she is loyal, possibly a different someone for everyone?
   (Or) there exists someone to whom everyone is loyal?)

7. People only try to assassinate rulers they are not loyal to.
   $\forall x : \forall y :$ person$(x) \wedge$ ruler$(y) \wedge$ tryassassinate(x,y)
   $\rightarrow \neg$ Loyalto(x,y)

8. Marcus tried to assassinate Caesar.
   tryassassinate(Marcus, Caesar)

Answering the question: Was Marcus loyal to Caesar?

we start with $\neg$loyalto(Marcus, Caesar) [reasoning backward from the desired goal]

$\neg$loyalto(Marcus, Caesar)
↑ (7, substitution)
person(Marcus) ∧
ruler(Caesar) ∧
tryassassinate(Marcus, Caesar)
↑ (4)
person(Marcus) ∧
tryassassinate(Marcus, Caesar)
↑ (8)
person(Marcus)

This attempt fails since there is no way to satisfy the goal person(Marcus).

If we add, 9. All men are people
$\forall x$: man$(x) \rightarrow$ person(x)

we can conclude Marcus was a man. (satisfying the goal)

person(Marcus)
↑ (9, substitution)
man(Marcus)

Computable Functions and Predicates:

Ex: gt(1,0) lt(0,1)
gt(2,1) lt(1,2)
gt(3,2) lt(2,3)
⋮ ⋮

} To represent these facts, we use 'computable predicates'.

gt(2+3, 1) — computable function

Consider the facts:

1. Marcus was a man.
   man(Marcus)
2. Marcus was a Pompeian.
   Pompeian(Marcus)
3. Marcus was born in 40 A.D.
   born(Marcus, 40)
4. All men are mortal.
   $\forall x$: man(x) $\longrightarrow$ mortal(x)
5. All Pompeians died when the volcano erupted in 79 A.D.
   erupted(volcano, 79) $\wedge$ $\forall x$: [Pompeian(x) $\longrightarrow$ died(x, 79)]
6. No mortal lives longer than 150 years.
   $\forall x: \forall t_1: \forall t_2:$ mortal(x) $\wedge$ born(x, t_1) $\wedge$ gt($t_2 - t_1$, 150) $\longrightarrow$ dead(x, t_2)
7. It is now 1997.
   now = 1997

We want to answer the question: "Is Marcus alive?"
We can show that Marcus is dead by deducing
① He was killed by the volcano (OR)
② He would be more than 150 years old, which is not possible.
But, we need additional knowledge (facts).

---

8. Alive means not dead.
   $\forall x: \forall t:$ [alive(x, t) $\longrightarrow$ $\neg$ dead(x, t)] $\wedge$ [$\neg$ dead(x, t) $\longrightarrow$ alive(x, t)]
9. If someone dies, then he is dead at all later times.
   $\forall x: \forall t_1: \forall t_2:$ died(x, t_1) $\wedge$ gt(t_2, t_1) $\longrightarrow$ dead(x, t_2)

The above facts (numbered differently) :-

1. man(Marcus)
2. Pompeian(Marcus)
3. born(Marcus, 40)
4. $\forall x$: man(x) $\longrightarrow$ mortal(x)
5. $\forall x$: Pompeian(x) $\longrightarrow$ died(x, 79)

6. erupted(volcano, 79)
7. $\forall x : \forall t_1 : \forall t_2 :$ mortal(x) $\wedge$ born(x, t_1) $\wedge$ gt$(t_2 - t_1, 150)$ $\rightarrow$ dead(x, t_2)
8. now = 1997
9. $\forall x : \forall t :$ [alive$(x,t) \rightarrow \neg$dead(x,t)] $\wedge$ [$\neg$dead$(x,t) \rightarrow$ alive(x,t)]
10. $\forall x : \forall t_1 : \forall t_2 :$ died(x, t_1) $\wedge$ gt(t_2, t_1) $\rightarrow$ dead(x, t_2)

To answer : "Is Marcus alive?"

To prove Marcus is dead :

$\neg$alive(Marcus, now)
↑ (9, substitution)
dead(Marcus, now)
↑ (10, substitution)
died(Marcus, t_1) $\wedge$ gt(now, t_1)
↑ (5, substitution)
Pompeian(Marcus) $\wedge$ gt(now, 79)
↑ (2)
gt(now, 79)
↑ (8, substitute equals)
gt(1997, 79)
↑ (compute gt)
nil

Another way to prove Marcus is dead :

$\neg$alive(Marcus, now)
↑ (9, substitution)
dead(Marcus, now)
↑ (7, substitution)
mortal(Marcus) $\wedge$
born(Marcus, t_1) $\wedge$
gt(now $- t_1$, 150)
↑ (4, substitution)
man(Marcus) $\wedge$
born(Marcus, t_1) $\wedge$
gt(now $- t_1$, 150)
↑ (1)
born(Marcus, t_1) $\wedge$
gt(now $- t_1$, 150)
↑ (3)
gt(now $-$ 40, 150)
↑ (8)
gt(1997 $-$ 40, 150)
↑ (compute minus)
gt(1957, 150)
↑ (compute gt)
nil

Conversion to CNF (Conjunctive Normal Form) (Clause Form)

Consider:
"All romans who know Marcus either hate Caesar or think that anyone who hates anyone is crazy".

As a wff,

$\forall x: [\text{Roman}(x) \wedge \text{know}(x, \text{Marcus})] \rightarrow$
$[\text{hate}(x, \text{Caesar}) \vee (\forall y: \exists z: \text{hate}(y,z) \rightarrow \text{thinkcrazy}(x,y))]$

To use this formula in a proof requires complex matching process. The formula would be easier to work with if,

- It were flatter, ie, there was less embedding of components.
- The quantifiers were separated from the rest of the formula, so that they did not need to be considered.

CNF has both of these properties.

The above in CNF:

$\neg\text{Roman}(x) \vee \neg\text{know}(x, \text{Marcus}) \vee$
$\text{hate}(x, \text{Caesar}) \vee \neg\text{hate}(y,z) \vee \text{thinkcrazy}(x,y)$

Clause :- A wff in CNF but with no instances of '$\wedge$'.

Algorithm :- Conversion to CNF.

(1) Eliminate $\rightarrow$. (using $a \rightarrow b$ is equivalent to $\neg a \vee b$).

So, we get

$\forall x: \neg[\text{Roman}(x) \wedge \text{know}(x, \text{Marcus})] \vee$
$[\text{hate}(x, \text{caesar}) \vee (\forall y: \neg(\exists z \cdot \text{hate}(y,z)) \vee \text{thinkcrazy}(x,y))]$

(2) Reduce the scope of each $\neg$ to a single term.
(using $\neg(\neg p) = p$, deMorgan's laws ($\neg(a \wedge b) = \neg a \vee \neg b$, $\neg(a \vee b) = \neg a \wedge \neg b$), and standard correspondence between quantifiers ($\neg\forall x: P(x) = \exists x: \neg P(x)$ and $\neg\exists x: P(x) = \forall x: \neg P(x)$)

So, (1) becomes,

$\forall x: [\neg\text{Roman}(x) \vee \neg\text{know}(x, \text{Marcus})] \vee$
$[\text{hate}(x, \text{caesar}) \vee (\forall y: \forall z: \neg\text{hate}(y,z) \vee \text{thinkcrazy}(x,y))]$

(3) Standardize variables so that each quantifier binds a unique variable.

For ex:, $\forall x: P(x) \vee \forall x: Q(x)$ could be converted to
$\forall x: P(x) \vee \forall y: Q(y)$

(∵ variable are just dummy names, this process cannot affect the truth value of the wff.)

④ Move all quantifiers to the left of the formula, without changing their relative order.

$\forall x : \forall y : \forall z : [\neg \text{Roman}(x) \vee \neg \text{know}(x, \text{Marcus})] \vee$
$[\text{hate}(x, \text{Caesar}) \vee (\neg \text{hate}(y,z) \vee \text{thinkcrazy}(x,y))]$

Now the wff is in "Prenex Normal Form". It consists of a prefix of quantifiers followed by a matrix, which is quantifier-free.

⑤ Eliminate existential quantifiers.

$\exists y : \text{President}(y)$ can be transformed into $\text{President}(S1)$
(S1 is a function with no arguments that somehow produces a value that satisfies President. Variable y is replaced by S1.)

$\forall x : \exists y : \text{father-of}(y, x)$ can be written as
$\forall x : \text{father-of}(S2(x), x)$
(since the value of y that satisfies father-of depends on the particular value of x.)
(These generated functions are called "Skolem functions". Skolem functions with no arguments are called "Skolem Constants".)

⑥ Drop the prefix.

So, ④ is,
$[\neg \text{Roman}(x) \vee \neg \text{know}(x, \text{Marcus})] \vee$
$[\text{hate}(x, \text{Caesar}) \vee (\neg \text{hate}(y,z) \vee \text{thinkcrazy}(x,y))]$

⑦ Convert the matrix into a conjunction of disjuncts.

Here, since there are no and's, apply the associative property
$a \vee (b \vee c) = (a \vee b) \vee c$.

$\neg \text{Roman}(x) \vee \neg \text{know}(x, \text{Marcus}) \vee$
$\text{hate}(x, \text{Caesar}) \vee \neg \text{hate}(y,z) \vee \text{thinkcrazy}(x,y)$

(distributive property: $(a \wedge b) \vee c = (a \vee c) \wedge (b \vee c)$

Ex: $[\text{winter} \vee (\text{summer} \wedge \text{wearingsandals})]$
$\wedge [\text{wearingboots} \vee (\text{summer} \wedge \text{wearingsandals})]$

↳ $(\text{winter} \vee \text{summer}) \wedge$
$(\text{winter} \vee \text{wearingsandals}) \wedge$
$(\text{wearingboots} \vee \text{summer}) \wedge$
$(\text{wearingboots} \vee \text{wearingsandals})$)

(8) Create a separate clause corresponding to each conjunct. In order for a wff to be true, all the clauses that are generated from it must be true.

(9) Standardize apart the variables in the set of clauses generated in step 8. (ie, Rename the variables so that no two clauses make reference to the same variable. For this, use —

$(\forall x : P(x) \wedge Q(x)) = \forall x : P(x) \wedge \forall x : Q(x)$)

After this procedure, we'll have a set of clauses, each of which is a disjunction of literals.

- These clauses can be used in 'Resolution' to generate proofs.

The Basis of Resolution:

The Resolution procedure is a simple iterative process:
— at each step, two clauses ('parent clauses') are compared (resolved), yielding a new clause that has been inferred from them. The new clause represents ways that the two parent clauses interact with each other.

Ex: winter ∨ summer → (1)
¬winter ∨ cold → (2)

(This means both the clauses must be true — although they look independent, are really conjoined.)

Now, one of 'winter' and '¬winter' will be true at any point.

If winter is true, then 'cold' must be true (to guarantee the truth of second clause).

If ¬winter is true, then 'summer' must be true (to guarantee the truth of first clause)

From (1) & (2), we can deduce, summer ∨ cold.

Resolution operates by taking two clauses that each contain the same literal, (here, winter). The literal must occur in positive form in one clause and in negative form in the other. The 'resolvent' is obtained by combining all of the literals of the two parent clauses except the ones that cancel

If an 'empty clause' is produced, then a contradiction is found.

winter, ¬winter } will produce empty clause.

If a contradiction exists, then eventually it will be found. If no contradiction exists, it is possible that the procedure will never terminate.

Resolution in Propositional Logic.

The procedure for producing a proof by resolution of proposition P with respect to a set of axioms F is the following:

Algorithm:

(1) Convert all the propositions of F to clause form.

(2) Negate P and convert the result to clause form. Add it to the set of clauses obtained in step (1).

(3) Repeat until either a contradiction is found or no progress can be made.

(a) Select two clauses. (Parent clauses)

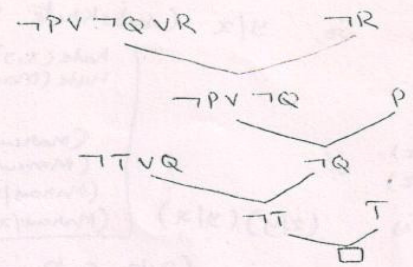
(b) Resolve them together. The 'resolvent' (resulting clause) will be the disjunction of all of the literals of both of the parent clauses with the following exception: If there are any pairs of literals L and ¬L such that one of the parent clauses contains L and the other contains ¬L, then select one such pair and eliminate both L and ¬L from the resolvent.

(c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

Given axioms	Converted to clause form	
P	P	(1)
$(P \wedge Q) \rightarrow R$	$\neg P \vee \neg Q \vee R$	(2)
$(S \vee T) \rightarrow Q$	$\neg S \vee Q$	(3)
	$\neg T \vee Q$	(4)
T	T	(5)

A few facts in Propositional logic

Resolution :



Explanation :

We want to prove R. For this one of ¬P, ¬Q or R must be true. But we're assuming that ¬R is true.

So, ¬P or ¬Q must be true for (2) to be true.

But from (1), P is true. So, ¬Q must be true.

(4) is true if either ¬T or Q is true.

Since ¬Q is true, ¬T must be true for (4) to be true.

But from (3), T is true. We get empty clause.

This indicates contradiction.

⇒ What we have assumed is wrong. So, R is true.

Refutation :- Resolution produces proofs by 'refutation'. It means, to prove a statement (ie, show that it is valid), resolution attempts to show that the negation of the statement produces a contradiction with the known statements (ie, that it is unsatisfiable).

The Unification Algorithm :

In Propositional Logic, it is easy to determine that two literals can not both be true at the same time. (Ex: L and ¬L)

In Predicate Logic, this matching process is more complex, since the ~~meaning~~ arguments of the predicates must be considered.

Ex: man(John) and ¬man(John) is a contradiction,
while man(John) and ¬man(Spot) is not.

To determine contradictions, we need a matching procedure that compares two literals and discovers whether there exists a set of substitutions that makes them identical. This is possible thro' 'Unification Algorithm'.

Ex: tryassassinate(Marcus, Caesar)
hate(Marcus, Caesar)
} cannot be unified.

39

Ex: P(x,x)
P(y,z)

P's are matched. So, y/x (substitute y for x).

Now P(y,y)
P(y,z)

z/y ⇒ P(z,z) ✓
P(z,z)

The composition is (z/y)(y/x)

hate(x,y) could be unified
hate(Marcus,z) with any of the
following substitutions:
(Marcus/x, z/y)
(Marcus/x, y/z)
(Marcus/x, Caesar/y, Caesar/z)
(Marcus/x, Polonius/y, Polonius/z)

In general, the substitution is $(a_1/a_2, a_3/a_4, \ldots)(b_1/b_2, b_3/b_4, \ldots)$.

Algorithm: Unify(L1,L2)

(returns as its value a list representing the composition of the substitutions that were performed during the match. The empty list, NIL, indicates that a match was found without any substitutions. The list consisting of the single value FAIL indicates that the unification procedure failed.)

1. If L1 or L2 are both variables or constants, then:
   (a) If L1 and L2 are identical, then return NIL.
   (b) Else if L1 is a variable, then if L1 occurs in L2 then return {FAIL}, else return (L2/L1).
   (c) Else if L2 is a variable, then if L2 occurs in L1 then return {FAIL}, else return (L1/L2).
   (d) Else return {FAIL}.
2. If the initial predicate symbols in L1 and L2 are not identical, then return {FAIL}.
3. If L1 and L2 have a different number of arguements, then return {FAIL}.
4. Set SUBST to NIL. (At the end of this procedure, SUBST will contain all the substitutions used to unify L1 and L2).
5. For i ← 1 to number of arguments in L1:
   (a) Call Unify with the ith argument of L1 and the ith argument of L2, putting result in S.
   (b) If S contains FAIL, then return {FAIL}.
   (c) If S is not equal to NIL, then:
      (i) Apply S to the remainder of both L1 and L2.
      (ii) SUBST := APPEND(S, SUBST).
6. Return SUBST.

Explanation: for 1(b) and 1(c).

$f(x, x)$
$f(g(x), g(x))$

If we substitute $g(x)$ for x ($g(x)/x$), we would have to substitute it for x in the remainder of the expressions. This leads to infinite recursion since it will never be possible to eliminate x.

Resolution in Predicate Logic:

Algorithm:

1. Convert all the statements of F to clause form.
2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in 1.
3. Repeat until either a contradiction is found, no progress can be made, or a predetermined amount of effort has been expended.
   (a) Select two clauses. (Parent clauses)
   (b) Resolve them together. The resolvent will be the disjunction of all the literals of both parent clauses with appropriate substitutions performed and with the following exception:

If there is one pair of literals T1 and ¬T2 such that one of the parent clauses contains T1 and the other contains T2 and if T1 and T2 are unifiable, then neither T1 nor T2 should appear in the resolvent. We call T1 and T2 'complementary literals'. Use the substitution produced by the unification to create the resolvent. If there is more than one pair of complementary literals, only one pair should be omitted from the resolvent.

   (c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

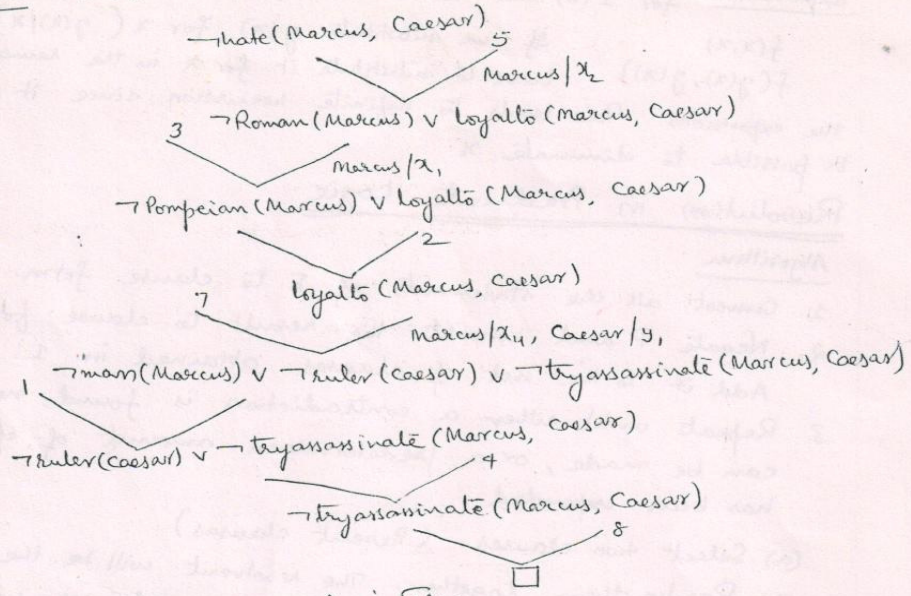
Ex:—

Axioms in clause form:

1. man(Marcus)
2. Pompeian(Marcus)
3. ¬Pompeian(x_1) ∨ Roman(x_1)
4. ruler(Caesar)
5. ¬Roman(x_2) ∨ loyalto(x_2, caesar) ∨ hate(x_2, Caesar)
6. loyalto(x_3, $f1(x_3)$)
7. ¬man(x_4) ∨ ¬ruler(y_1) ∨ ¬tryassassinate(x_4, y_1) ∨ loyalto(x_4, y_1)
8. tryassassinate(Marcus, Caesar)

40

Prove: hate(Marcus, Caesar)



So, hate(Marcus, Caesar) is True.

K. SRINIVASA RAO, MCA
Asst. Prof. (GITAM)
VSP.

Chapter 6: Representing Knowledge using Rules

Declarative vs Procedural representation:

Ex: man(Marcus)
man(Caesar)
person(cleopatra)
$\forall x$: man(x) → person(x)

Declarative — just declaring the facts (and/or rules)
— order is not important
— extracting the desired information is not possible.

For ex: $\exists y$: person(y) (Question or Goal)

y = Marcus (or) Caesar (or) cleopatra
(but none of them is selected)

Procedural — in which the control information that is necessary to use the knowledge is considered to be embedded in the knowledge itself.

Control information — how to move through the given facts and rules
✓ — part of the interpreter

Ex: 1. man(Marcus)
2. man(Caesar)
3. person(cleopatra)
4. $\forall x$: man(x) → person(x)
(person(X) :- man(X).)

Order from first to last.

Qn: $\exists y$: person(y).
Fact 3 is matched. So, y = cleopatra

Now consider:
1. man(Marcus)
2. man(Caesar)
3. $\forall x$: man(x) → person(x)
(person(X) :- man(X).)
4. person(cleopatra)

Order: from first to last.

for the above question,
. Rule 3 is matched. The subgoal 'man(X)' is established.
. Start from the first. Then X = Marcus.
. So, y = Marcus

(Ordering the facts makes difference)

14-12-1998

Ch: 6 · REPRESENTING KNOWLEDGE USING RULES

Logic Programming:

There are several logic programming systems today, one of which is 'PROLOG'.

A PROLOG program is a series of logical assertions, each of which is a 'Horn clause'. Programs written in pure PROLOG are composed only of Horn clauses.

Horn clause — a clause that has at most one positive literal.

Ex: p, $\neg p \vee q$, $p \rightarrow q$ ($p \rightarrow q$ is $\neg p \vee q$)

Advantages of Horn clauses:

1) Because of the uniform representation, a simple and efficient interpreter can be written.
2) The logic of Horn clause systems is decidable (unlike that of full first-order predicate logic)

A simple knowledge base:

A representation in logic:

$\forall x$: 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)

A representation in PROLOG

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

15-12-98

Both of these representations contain two types of statements,

(1) Facts — which contain only constants (ie, no variables)
(2) Rules — rules which do contain variables.

Facts — represent statements about specific objects.
Rules — represent statements about classes of objects.

(Ex: Fact : poodle(fluffy).
Rule : pet(X) :- dog(X).)

Syntactic differences between the logic and the PROLOG:

1) In PROLOG, variables begin with Upper case letters and constants begin with lower case letters or numbers.
2) In logic, there are explicit symbols for and ($\wedge$) and or ($\vee$). In PROLOG, there is an explicit symbol for and (,), but there is none for 'or'.
3) In logic, implications like "p implies q" are written as $p \rightarrow q$. In PROLOG, it is written "backward", as $q :- p$.

A problem in PROLOG:

Find a value of X that satisfies the predicate apartmentpet(X).

So, the goal is ? - apartmentpet(X).

16/12/98 Soln:- PROLOG interpreter looks for a fact with the predicate 'apartmentpet' or a rule with that predicate as its head.

- There are no facts with this predicate. So, select the rule : apartmentpet(X) :- pet(X), small(X).
- This rule will succeed if the clauses pet(X) and small(X) are true. The interpreter tries these in the order in which they appear.

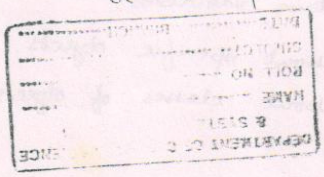
(1) There are no facts with predicate 'pet'. But there are two rules.

pet(X) :- cat(X).
pet(X) :- dog(X).

Try these in the order in which they occur cat(X) will fail because there are no assertions about the predicate 'cat' in the program.

- No facts with the predicate 'dog'. But we have a rule : dog(X) :- poodle(X).

Now there is a fact with predicate 'poodle'. It is poodle(fluffy). So X is 'fluffy'.



(2) Now small(X) is small(fluffy).
There is no fact to prove small(fluffy) is true.
But there is a rule: small(X) :- poodle(X).
From poodle(fluffy), X is fluffy.
So, apartmentpet(fluffy) is true. (X is fluffy).

Forward vs Backward Reasoning :-

Search directions — Forward, from the start states
Backward, from the goal states

(PROLOG only searches from a goal state)

Ex: 8-puzzle problem.

Start-

2	8	3
1	6	4
7		5

Goal

1	2	3
8		4
7	6	5

Assume the areas of the tray are numbered as:

1	2	3
4	5	6
7	8	9

Define Rules:

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

⋮

Forward Reasoning :- Consider the initial state as the root-of the tree. Generate the next level of the tree by finding all the rules whose left sides match the root node and using their right sides to create 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.

4.9

Backward Reasoning :-

Consider the goal state as the root of the tree.

Generate the next level of the tree by finding all the rules whose right sides match the root node. 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 is also called 'goal-directed reasoning')

Note :- Depending on the topology of the problem space, one kind of search may be better than the other.

To reason forward or backward, consider the following points:

(1) Are there more possible start states or goal states?
(we will move from smaller set of states to larger set of states)

(2) Branching factor :- is the average no. of nodes that can be reached directly from a single node.
(we will choose the direction with the lower branching factor)

(3) we will proceed in the direction that corresponds more closely with the way the user will think.

(4) If it is the arrival of a new fact, forward reasoning is better. If it is a query to which a response is desired, backward reasoning is more natural.

Bidirectional 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'.

K. SRINIVASA RAO, MCA
Asst. Prof. (GITAM)
VSP.

Two classes of rules:

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 — encode knowledge about how to respond to certain input configurations
- Backward rules — encode knowledge about how to achieve particular goals

Hence, we essentially add to each rule an additional piece of information, namely how it should be used in problem solving. The following are the three types of rule systems.

(1) Backward-Chaining Rule systems:

These are good for goal-directed problem solving.

Ex: 1. PROLOG (a query system uses backward chaining to reason about and answer user questions)

In PROLOG — 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.

Ex: 2. MYCIN

MYCIN has more complex rules. In MYCIN, rules can be augmented with probabilistic certainty factors to reflect the fact that some rules are more reliable than others.

(2) Forward-chaining rule systems:

Sometimes we want to be directed by incoming data.

Ex: when you sense searing heat near your hand, then you are likely to jerk your hand away.

- This can be better modelled by forward-chaining rule system (as recognize-act cycle).
- Forward chaining means: left sides of rules are matched against the state descriptions and rules that match dump their right-hand sides into the states.

- Matching is more complex for forward-chaining systems than backward ones.

After the rule fires, its conditions are probably still valid, so it could fire again immediately. We will need some mechanism to prevent repeated firings, especially if the state remains unchanged.

- Most forward-chaining systems implement highly efficient matchers and supply several mechanisms for preferring one rule over another.

(3) Combining Forward and Backward reasoning:

Sometimes certain aspects of a problem are best handled via forward chaining and other aspects by backward chaining.

Ex: medical diagnosis program

Patient's health condition:

10 conditions (on the LHS) ⟶ Disease (on the RHS)

Suppose that at some point, the left side of a rule was nearly satisfied — say 9 preconditions were met.

Then, it is better to go for backward reasoning here to satisfy the 10th precondition, rather than wait for forward chaining to supply the fact by accident.

or, perhaps the 10th condition requires further medical tests. In that case, backward chaining can be used to query the user.

Same rules can be used for — forward reasoning, and backward reasoning.

- If both left sides and right sides contain pure assertions, then forward chaining can match assertions on the left-side of a rule and 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.

✓ When irreversible rules are used, then a commitment to the direction of search must be made at the time the rules are written.