

**AIM:**

To learn PROLOG terminologies and write basic programs.

**TERMINOLOGIES**

**1. Atomic Terms: -**

Atomic terms are usually strings made up of lower- and

uppercase letters, digits, and the underscore, starting with a

lowercase letter.

Ex:

dog

ab\_c\_321

**2. Variables: -**

Variables are strings of letters, digits, and the underscore,

starting with a capital letter or an underscore.

Ex:

Dog

Apple\_420

**3. Compound Terms: -**

Compound terms are made up of a PROLOG atom and a

number of arguments (PROLOG terms, i.e., atoms, numbers,

variables, or other compound terms) enclosed in parentheses

and separated by commas.

Ex:

**4. Facts: -**

is\_bigger(elephant,X)

f(g(X,\_),7)

A fact is a predicate followed by a dot.

Ex:

bigger\_animal(whale).

life\_is\_beautiful.

**5. Rules: -**

A rule consists of a head (a predicate) and a body (a sequence

of predicates separated by commas).

Ex:

is\_smaller(X,Y):-is\_bigger(Y,X).

aunt(Aunt,Child):-sister(Aunt,Parent),parent(Parent,Child).

**PROGRAM:**

**SOURCE CODING:**

**KB1:**

woman(mia).

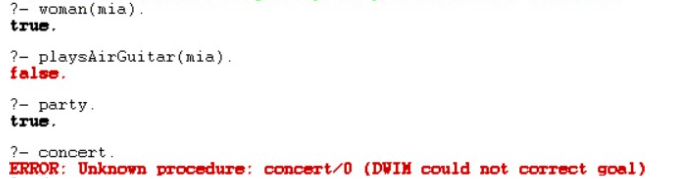
woman(jody).

woman(yolanda).

playsAirGuitar(jody).

party.

# OUTPUT:



**KB2:**

happy(yolanda).

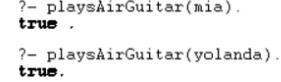
listens2music(mia).

Listens2music(yolanda):-happy(yolanda).

playsAirGuitar(mia):-listens2music(mia).

playsAirGuitar(Yolanda):-listens2music(yolanda).

# OUTPUT:



**KB3:**

likes(dan,sally).

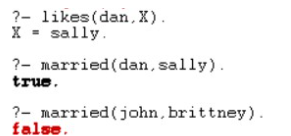
likes(sally,dan).

likes(john,brittney).

married(X,Y) :- likes(X,Y) , likes(Y,X).

friends(X,Y) :- likes(X,Y) ; likes(Y,X).

# OUTPUT:



**KB4:**

food(burger).

food(sandwich).

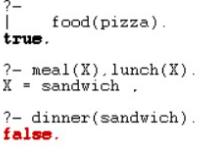
food(pizza).

lunch(sandwich).

dinner(pizza).

meal(X):-food(X).

# OUTPUT:



**KB5:**

owns(jack,car(bmw)).

owns(john,car(chevy)).

owns(olivia,car(civic)).

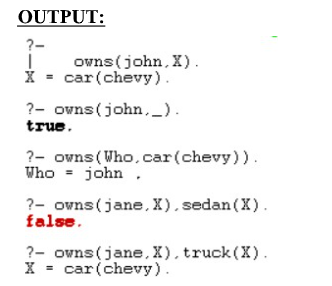
owns(jane,car(chevy)).

sedan(car(bmw)).

sedan(car(civic)).

truck(car(chevy)).

# OUTPUT:



**KB6:**

Find minimum maximum of two numbers

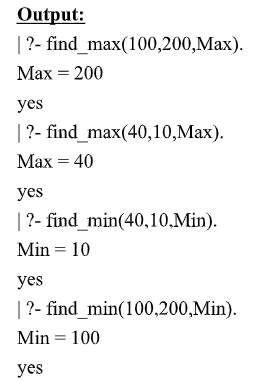
find\_max(X,Y,X):-X&gt;=Y,!.

find\_max(X,Y,Y):-X&lt;Y.

find\_min(X,Y,X):-X=&lt;Y,!.

find\_min(X,Y,Y):-X&gt;Y.

# OUTPUT:



**KB7:**

Here are some simple clauses.

likes(mary,food).

likes(mary,wine).

likes(john,wine).

likes(john,mary).

**How do you add the following facts?**

1. John likes anything that Mary likes

2. John likes anyone who likes wine

3. John likes anyone who likes themselves

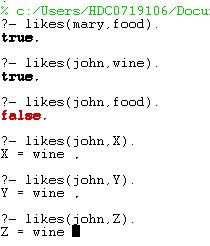
**% New facts and rules**

likes(john, X) :- likes(mary, X).

likes(john, Y) :- likes(Y, wine).

likes(john, Z) :- likes(Z, Z).

# OUTPUT:



**RESULT:**

Thus to learn PROLOG terminologies has been executed successfully.

|  |  |
| --- | --- |
| **EX.NO :** | **PRINCIPLES OF ARTIFICIAL INTELLIGENCE**  **UNIFICATION AND RESOLUTION** |
| **DATE :** |

**AIM:**

To execute programs based on Unification and Resolution.Deduction in prolog is based

on the Unification and Instantiation. Matching terms are unified and variables get instantiated.

Example 1: In the below prolog program , unification and instantiation take place after querying.

Facts :

likes(john, jane).

likes(jane, john).

Query :

?- likes(john, X).

Answer : X = jane.

Here upon asking the query first prolog start to search matching terms in predicate with two

arguments and it can match likes(john, ...) i.e.Unification. Then it looks for the value of X asked

in query and it returns answer X = jane i.e.Instantiation - X is instantiated to jane.

Example 2 : At the prolog query prompt, when you write below query,

?- owns(X, car(bmw)) = owns(Y, car(C)).

You will get Answer : X = Y, C = bmw.

Here owns(X, car(bmw)) and owns(Y, car(C)) unifies – because

(i) predicate names are same on both side

(ii) number of arguments for that predicate, i.e. 2, are

equal both side.

(iii) 2nd argument with predicate inside the brackets are same both side and even in that

predicate again number of arguments are same. So, here terms unify in which X=Y. So,

Y is substituted with X -- i.e. written as {X | Y} and C is instantiated to bmw, -- written

as {bmw | C} and this is called Unification with Instantiation.

But when you write ?- owns(X, car(bmw)) = likes(Y, car(C)). then prolog will return False, since

it can not match the ;owns; and ;likes; predicates.

Resolution is one kind of proof technique that works this way –

(i) select two clauses that

contain conflicting terms

(ii) combine those two clauses and

(iii) cancel out the conflicting terms.

For example we have following statements,

(1) If it is a pleasant day you will do strawberry picking

(2) If you are doing strawberry picking you are happy.

Above statements can be written in propositional logic like this -

(1) strawberry\_picking ← pleasant

(2) happy ← strawberry\_picking

And again these statements can be written in CNF like this -

(1) (strawberry\_picking ∨~pleasant) ∧

(2) (happy ∨~strawberry\_picking)

By resolving these two clauses and cancelling out the conflicting terms

;strawberry\_picking; and ;~strawberry\_picking;, we can have one new

clause,

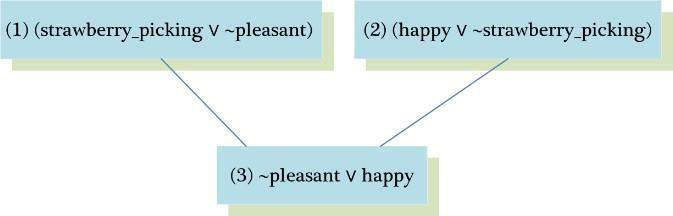
(3) ~pleasant ∨ happy

How ? See the figure on right.

When we write above new clause in infer or implies form, we have

;pleasant → happy; or ;happy ← pleasant;

i.e. If it is a pleasant day you are happy.



But sometimes from the collection of the statements we have, we want to know the answer

of this question - &amp;quot;Is it possible to prove some other statements from what we actually

know?&amp;quot; In order to prove this we need to make some inferences and those other statements

can be shown true using Refutation proof method i.e. proof by contradiction using

Resolution. So for the asked goal we will negate the goal and will add it to the given

statements to prove the contradiction.

Let;s see an example to understand how Resolution and Refutation work. In below

example, Part(I) represents the English meanings for the clauses, Part(II) represents the

propositional logic statements for given english sentences, Part(III) represents the

Conjunctive Normal Form (CNF) of Part(II) and Part(IV) shows some other statements we

want to prove using Refutation proof method.

Part(I) : English Sentences

(1) If it is sunny and warm day you will enjoy.

(2) If it is warm and pleasant day you will do strawberry picking

(3) If it is raining then no strawberry picking.

(4) If it is raining you will get wet.

(5) It is warm day

(6) It is raining

(7) It is sunny

Part(II) : Propositional Statements

(1) enjoy ← sunny ∧ warm

(2) strawberry\_picking ← warm ∧ pleasant

(3) ~strawberry\_picking ← raining

(4) wet ← raining

(5) warm

(6) raining

(7) sunny

Part(III) : CNF of Part(II)

(1) (enjoy ∨~sunny∨~warm) ∧

(2) (strawberry\_picking ∨~warm∨~pleasant) ∧

(3) (~strawberry\_picking ∨~raining) ∧

(4) (wet ∨~raining) ∧

(5) (warm) ∧

(6) (raining) ∧

(7) (sunny)

Part(IV) : Other statements we want to prove by Refutation

(Goal 1) You are not doing strawberry picking.

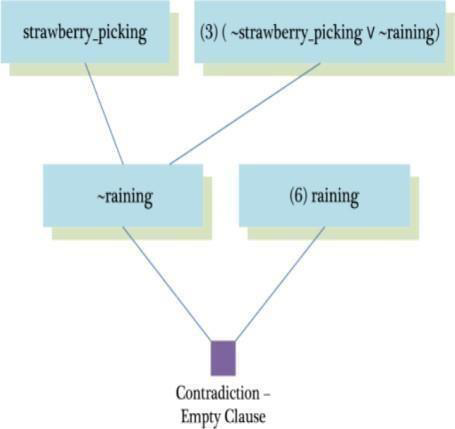
(Goal 2) You will enjoy.

(Goal 3) Try it yourself : You will get wet.

Goal 1 : You are not doing strawberry picking.

Prove : ~strawberry\_picking

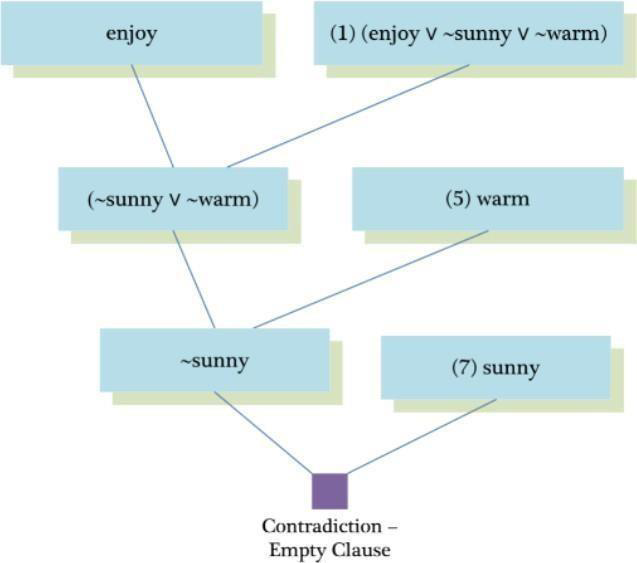
Assume : strawberry\_picking (negate the goal and add it to given clauses).



Goal 2 : You will enjoy.

Prove : enjoy

Assume : ~enjoy (negate the goal and add it to given clauses)



**SOURCE CODE:**

enjoy:-sunny,warm.

strawberrry\_picking:-warm,plesant.

notstrawberry\_picking:-raining.

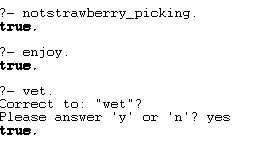
wet:-raining.

warm.

raining.

sunny.

# OUTPUT:



**RESULT:**

Thus, to execute programs based on Unification and Resolution has been executed successfully.

|  |  |
| --- | --- |
| **EX.NO:** | **PRINCIPLES OF ARTIFICIAL INTELLIGENCE**  **Recursive Best-First Search** |
| **DATE:** |

**AIM:**

**ALGORITHM:**

**PROGRAM:**

**CODING:**

class Node:

def \_\_init\_\_(self, state, parent=None, cost=0, heuristic=0):

self.state = state

self.parent = parent

self.cost = cost # g(n)

self.heuristic = heuristic # h(n)

self.f\_cost = cost + heuristic # f(n) = g(n) + h(n)

def \_\_lt\_\_(self, other):

return self.f\_cost < other.f\_cost

def \_\_repr\_\_(self):

return f"Node({self.state}, f\_cost={self.f\_cost})"

def rbfs(problem, node, f\_limit):

print(f"RBFS called with node: {node}, f\_limit: {f\_limit}")

if problem.is\_goal(node.state):

return node, 0 # Solution found

successors = []

for successor in problem.get\_successors(node.state):

s = Node(successor['state'], node, node.cost + successor['cost'], successor['heuristic'])

successors.append(s)

if not successors:

return None, float('inf') # Dead end

successors.sort()

while successors:

best = successors[0]

if best.f\_cost > f\_limit:

return None, best.f\_cost

alternative = successors[1].f\_cost if len(successors) > 1 else float('inf')

result, best.f\_cost = rbfs(problem, best, min(f\_limit, alternative))

if result is not None:

return result, best.f\_cost

successors.sort()

return None, float('inf')

class Problem:

def \_\_init\_\_(self, initial\_state, goal\_state, successors):

self.initial\_state = initial\_state

self.goal\_state = goal\_state

self.successors = successors

def is\_goal(self, state):

return state == self.goal\_state

def get\_successors(self, state):

return self.successors[state]

def recursive\_best\_first\_search(problem):

start\_node = Node(problem.initial\_state, None, 0, problem.get\_successors(problem.initial\_state)[0]['heuristic'])

result, \_ = rbfs(problem, start\_node, float('inf'))

return result

# Example usage:

initial\_state = 'A'

goal\_state = 'G'

successors = {

'A': [{'state': 'B', 'cost': 1, 'heuristic': 5}, {'state': 'C', 'cost': 4, 'heuristic': 2}],

'B': [{'state': 'D', 'cost': 2, 'heuristic': 2}, {'state': 'E', 'cost': 5, 'heuristic': 1}],

'C': [{'state': 'F', 'cost': 1, 'heuristic': 4}],

'D': [{'state': 'G', 'cost': 3, 'heuristic': 0}],

'E': [],

'F': [{'state': 'G', 'cost': 1, 'heuristic': 0}],

'G': []

}

problem = Problem(initial\_state, goal\_state, successors)

solution = recursive\_best\_first\_search(problem)

# Reconstruct path

path = []

node = solution

while node:

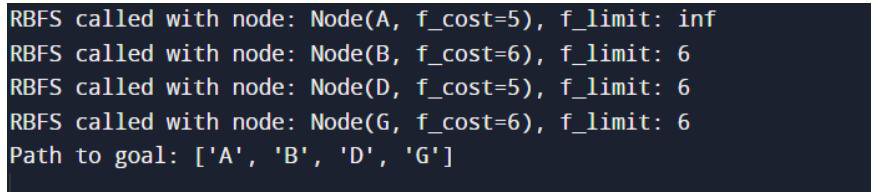
path.append(node.state)

node = node.parent

path.reverse()

print("Path to goal:", path)

**OUTPUT:**



**RESULT:**