RAJALAKSHMI ENGINEERING COLLEGE

An AUTONOMOUS Institution

Affiliated to ANNA UNIVERSITY, Chennai

25 years OF ACADEMIC EXCELLENCE

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

LABORATORY MANUAL

REGULATION - 2023

AI23231 – PRINCIPLES OF ARTIFICIAL INTELLIGENCE

RAJALAKSHMI ENGINEERING COLLEGE

An Autonomous Institution, Affiliated to Anna University Rajalakshmi Nagar, Thandalam – 602 105

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

AI23231 – PRINCIPLES OF ARTIFICIAL INTELLIGENCE (***Regulation 2023***)
LAB RECORD

Name	M.HARISH
Register No.	2116-231501058
Year / Branch / Section:	1st Year / AIML / A
Semester	2nd Semester
Academic Year	2023-2024

INDEX

1.PROGRAMS ON PROBLEM SOLVING

EX.N0:1	**WRITE A PROGRAM TO SOLVE 8 QUEENS PROBLEM**
DATE:	

PROBLEM STATEMENT:

In 8x8 chess board find a way to place 8 Queens such that no queens can attack any other queens on chessboard. A queen can only be attacked if it lies on the same row or same column or the same diagonal of any other queen all the possible configurations.

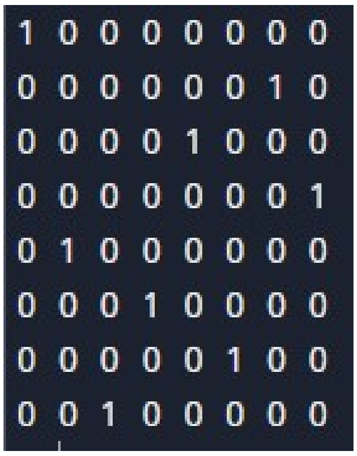
To solve this problem, we will make use of the Backtracking algorithm. The backtracking algorithm, in general checks all possible configurations and test whether the required result is obtained or not. For the given problem, we will explore all possible positions the queens can be relatively placed at. The solution will be correct when the number of placed queens = 8.

PROGRAM:

N = 8 # size of the

```
N = 8  # size of the chessboard

def solveNQueens(board, col):
    if col == N:
        print(board)
        return True
    for i in range(N):
        if isSafe(board, i, col):
            board[i][col] = 1
            if solveNQueens(board, col + 1):
                return True
            board[i][col] = 0
    return False

def isSafe(board, row, col):
    for x in range(col):
        if board[row][x] == 1:
            return False
    for x, y in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[x][y] == 1:
            return False
    for x, y in zip(range(row, N, 1), range(col, -1, -1)):
            if board[x][y] == 1:
                return False
        return True

    board = [[0 for x in range(N)] for y in range(N)]
    if not solveNQueens(board, 0):
        print("No solution found")
```

OUTPUT:



```
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
```

EX.N0 : 2.a)	**INTRODUCTION TO DEPTH FIRST SEARCH**
DATE :	

DEPTHFIRSTSEARCH

- Depth first search (DFS) algorithm or searching technique starts with the root node of the graph G, and then travel to deeper and deeper until we find the goal node or the node which has no children by visiting different node of the tree.
- The algorithm, then backtracks or return back from the dead end or last node towards the most recent node that is yet to be completely unexplored.
- The data structure(DS) which is being used in DFS Depth first search is stack. The process is quite similar to BFS algorithm.
- In DFS, the edges that goes to an unvisited node are called discovery edges while the edges that goes to an already visited node are called block edges.

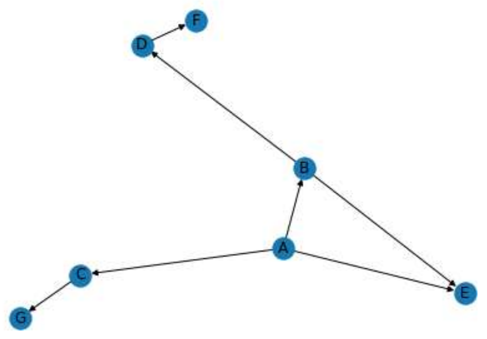
AIM:

SOURCECODE :

```
import networkx as nx

# FUNCTION TO SOLVE DFS
def solveDFS(graph, v, visited):
    visited.add(v)
    print(v, end=' ')
    for neighbour in graph[v]:
        if neighbour not in visited:
            solveDFS(graph, neighbour, visited)

# CREATE A GRAPH USING NETWORKX
g = nx.DiGraph()
g.add_edges_from([('A', 'B'), ('A', 'C'), ('C', 'G'), ('B', 'D'), ('B', 'E'), ('D',
'F'),    ('A', 'E')])

# Add edges for the graph
nx.draw(g, with_labels=True)  # Graph Visualization

# SOLVE DFS FOR THAT GRAPH
print("Following is DFS from (starting from vertex A):")
visited = set()
solveDFS(g, 'A', visited)
```

OUTPUT:

```
Following is DFS from (starting from vertex A)
A B D F E C G
```



RESULT:

1.WATER JUG PROGRAM USING BFS

EX.N0:2.b)	**WRITE PROGRAM TO WATER JUG PROGRAM USING BFS**
DATE:	

PROBLEMSTATEMENT:

DEPTHFIRSTSEARCH –WATERJUGPROBLEM

In the water jug problem in Artificial Intelligence, we are provided with two jugs: one having the capacity to hold 3 gallons of water and the other has the capacity to hold 4 gallons of water. There is no other measuring equipment available and the jugs also do not have any kind of marking on them. So, the agent’s task here is to fill the 4-gallon jug with 2 gallons of water by using only these two jugs and no other material. Initially, both our jugs are empty.

AIM:

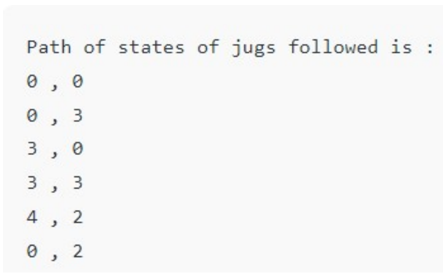
.

ALGORITHM:

PROGRAM:

```python
from collections import deque
def BFS(a, b, target):
    m = {}
    isSolvable = False
    path = []
    q = deque()
    q.append((0, 0))

    while len(q) > 0:
        u = q.popleft()

        # If this state is already visited
        if (u[0], u[1]) in m:
            continue

        # If any of the jugs has more water than its capacity or less than 0, skip this 
state
        if u[0] > a or u[1] > b or u[0] < 0 or u[1] < 0:
            continue

        # Add this state to the path
        path.append([u[0], u[1]])

        # Mark this state as visited
        m[(u[0], u[1])] = 1

        # If we reach the target, print the solution
        if u[0] == target or u[1] == target:
            isSolvable = True
            if u[0] == target and u[1] != 0:
                path.append([u[0], 0])
            elif u[1] == target and u[0] != 0:
                path.append([0, u[1]])
            sz = len(path)
            for i in range(sz):
                print("(", path[i][0], ",", path[i][1], ")")
            break

        # Add the next possible states
        q.append((u[0], b))  # Fill Jug2
        q.append((a, u[1]))  # Fill Jug1

        for ap in range(max(a, b) + 1):
            # Pour water from Jug2 to Jug1
            c = u[0] + ap
            d = u[1] - ap
            if c == a or (d == 0 and d >= 0):
                q.append((c, d))
            # Pour water from Jug1 to Jug2
            c = u[0] - ap
            d = u[1] + ap
            if (c == 0 and c >= 0) or d == b:
                q.append((c, d))

        q.append((a, 0))  # Empty Jug1
        q.append((0, b))  # Empty Jug2

    if not isSolvable:
        print("No solution")

if __name__ == "__main__":
    Jug1, Jug2, target = 4, 3, 2
    print("Path from initial state to solution state ::")
    BFS(Jug1, Jug2, target)
```

OUTPUT:



```
Path of states of jugs followed is :
0 , 0
0 , 3
3 , 0
3 , 3
4 , 2
0 , 2
```

RESULT:

WATER JUG PROGRAM USING DFS

EX.N0:3	**WRITE PROGRAM TO WATER JUG PROGRAM USING DFS**
DATE:	

PROBLEM STATEMENT:

In the water jug problem in Artificial Intelligence, we are provided with two jugs: one having the capacity to hold 3 gallons of water and the other has the capacity to hold 4 gallons of water. There is no other measuring equipment available and the jugs also do not have any kind of marking on them. So, the agent's task here is to fill the 4-gallon jug with 2 gallons of water by using only these two jugs and no other material. Initially, both our jugs are empty.

AIM:

.

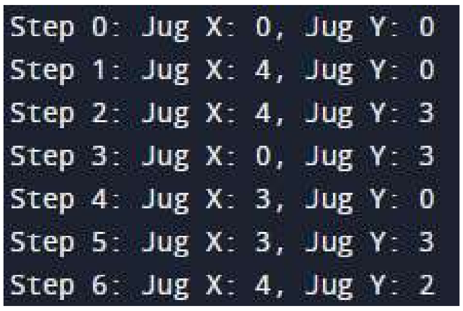
ALGORITHM:

PROGRAM:

```
def water_jug_dfs(capacity_x, capacity_y, target):
    def dfs(x, y, path):
        if x == target or y == target:
            path.append((x, y))
            return True
        if visited[x][y]:
            return False
        visited[x][y] = True

        if x < capacity_x:  # Fill Jug X
            if dfs(capacity_x, y, path):
                path.append((x, y))
                return True
        if y < capacity_y:  # Fill Jug Y
            if dfs(x, capacity_y, path):
                path.append((x, y))
                return True
        if x > 0:  # Empty Jug X
            if dfs(0, y, path):
                path.append((x, y))
                return True
        if y > 0:  # Empty Jug Y
            if dfs(x, 0, path):
                path.append((x, y))
                return True
        if x > 0 and y < capacity_y:  # Pour water from Jug X to Jug Y
            pour = min(x, capacity_y - y)
            if dfs(x - pour, y + pour, path):
                path.append((x, y))
                return True
        if y > 0 and x < capacity_x:  # Pour water from Jug Y to Jug X
            pour = min(y, capacity_x - x)
            if dfs(x + pour, y - pour, path):
                path.append((x, y))
                return True

        return False

    visited = [[False for _ in range(capacity_y + 1)] for _ in range(capacity_x + 1)]
    path = []

    if dfs(0, 0, path):
        path.reverse()
        return path
    else:
        return "No solution found."
```

```python
capacity_x = 4
capacity_y = 3
target = 2

solution_path = water_jug_dfs(capacity_x, capacity_y, target)
if solution_path != "No solution found.":
    for step, (x, y) in enumerate(solution_path):
        print(f"Step {step}: Jug X: {x}, Jug Y: {y}")
else:
    print("No solution found.")
```

OUTPUT:



```
Step 0: Jug X: 0, Jug Y: 0
Step 1: Jug X: 4, Jug Y: 0
Step 2: Jug X: 4, Jug Y: 3
Step 3: Jug X: 0, Jug Y: 3
Step 4: Jug X: 3, Jug Y: 0
Step 5: Jug X: 3, Jug Y: 3
Step 6: Jug X: 4, Jug Y: 2
```

RESULT:

EX.N0:4	**A* SEARCH ALGORITHM**
DATE:	

A*SEARCHALGORITHM

- A heuristic algorithm sacrifices optimality, with precision and accuracy for speed, to solve problems faster and more efficiently.
- All graphs have different nodes or points which the algorithm has to take, to reach the final node. The paths between these nodes all have a numerical value, which is considered as the weight of the path. The total of all paths transverse gives you the cost of that route.
- Initially, the Algorithm calculates the cost to all its immediate neighboring nodes, n, and chooses the one incurring the least cost. This process repeats until no new nodes can be chosen and all paths have been traversed. Then, you should consider the best path among them. If f(n) represents the final cost, then it can be denoted as :
- f(n)=g(n)+h(n), where:
- g(n)= cost of traversing from one node to another. This will vary from node to node
- h(n) = heuristic approximation of the node's value. This is not a real value but an approximation cost.

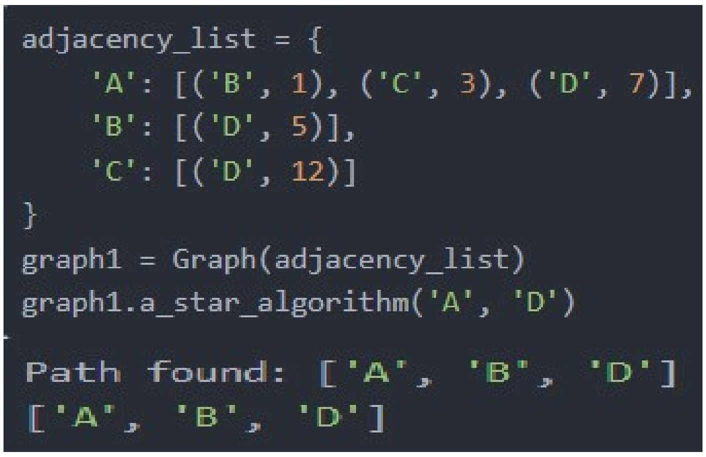
AIM:

ALGORITHM:

PROGRAM:

```
from collections import deque

class Graph:
    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        return self.adjacency_list[v]

    def h(self, n):
        H = {'A': 1, 'B': 1, 'C': 1, 'D': 1}
        return H[n]

    def a_star_algorithm(self, start_node, stop_node):
        open_list = set([start_node])
        closed_list = set([])

        g = {}
        g[start_node] = 0

        parents = {}
        parents[start_node] = start_node

        while len(open_list) > 0:
            n = None

            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n] + self.h(n):
                    n = v

            if n == None:
                print('Path does not exist!')
                return None

            if n == stop_node:
                reconst_path = []

                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]

                reconst_path.append(start_node)
                reconst_path.reverse()

                print('Path found: {}'.format(reconst_path))
                return reconst_path
for (m, weight) in self.get_neighbors(n):
                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n

                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)

            open_list.remove(n)
            closed_list.add(n)

        print('Path does not exist!')
        return None
adjacency_list = {
    'A': [('B', 1), ('C', 3)],  'B': [('D', 1)], 'C': [('D', 12)] }
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')
```

OUTPUT:



```
adjacency_list = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')

Path found: ['A', 'B', 'D']
['A', 'B', 'D']
```

RESULT:

AO* SEARCH ALGORITHM

EX.N0:5	**AO* SEARCH ALGORITHM**
DATE:	

- AO* algorithm is a best first search algorithm. AO* algorithm uses the concept of AND-OR graphs to decompose any complex problem given into smaller set of problems which are further solved.
- AND-OR graphs are specialized graphs that are used in problems that can be broken down into sub problems where AND side of the graph represent a set of task that need to be done to achieve the main goal , whereas the or side of the graph represent the different ways of performing task to achieve the same main goal

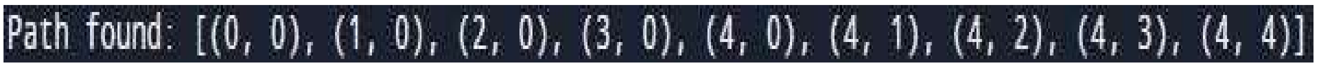
AIM:

ALGORITHM:

PROGRAM:

```python
class Node:
    def __init__(self, state, g_value, h_value, parent=None):
        self.state = state
        self.g_value = g_value
        self.h_value = h_value
        self.parent = parent

    def f_value(self):
        return self.g_value + self.h_value

def a_star_search(initial_state, is_goal, successors, heuristic):
    open_list = [Node(initial_state, 0, heuristic(initial_state), None)]
    closed_set = set()
    while open_list:
        open_list.sort(key=lambda node: node.f_value())
        current_node = open_list.pop(0)
        if is_goal(current_node.state):
            path = []
            while current_node:
                path.append(current_node.state)
                current_node = current_node.parent
            return list(reversed(path))
        closed_set.add(current_node.state)
        for child_state in successors(current_node.state):
            if child_state in closed_set:
                continue
            g_value = current_node.g_value + 1
            h_value = heuristic(child_state)
            child_node = Node(child_state, g_value, h_value, current_node)
            for i, node in enumerate(open_list):
                if node.state == child_state:
                    if node.g_value > g_value:
                        open_list.pop(i)
                        break
                elif node.g_value > g_value:
                    open_list.insert(i, child_node)
                    break
            else:
                open_list.append(child_node)
    return None

if __name__ == "__main__":
    def is_goal(state):
        return state == (4, 4)

    def successors(state):
        x, y = state
        return [(x+1, y), (x, y+1)]


def heuristic(state):
    x, y = state
    return abs(4 - x) + abs(4 - y)

initial_state = (0, 0)
path = a_star_search(initial_state, is_goal, successors, heuristic)
if path:
    print("Path found:", path)
else:
    print("No path found")
```

OUTPUT:



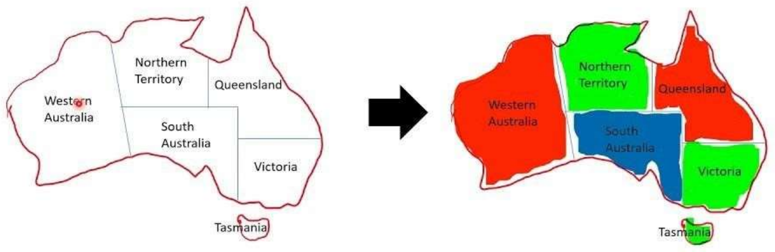
```
Path found: [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)]
```

RESULT:

EX.N0: 6	**CSP-MAP COLOURING**
DATE:	

Map Colouring

- Two adjacent regions cannot have the same color no matter whatever color we choose.
- The goal is to assign colors to each region so that no neighboring regions have the same color.



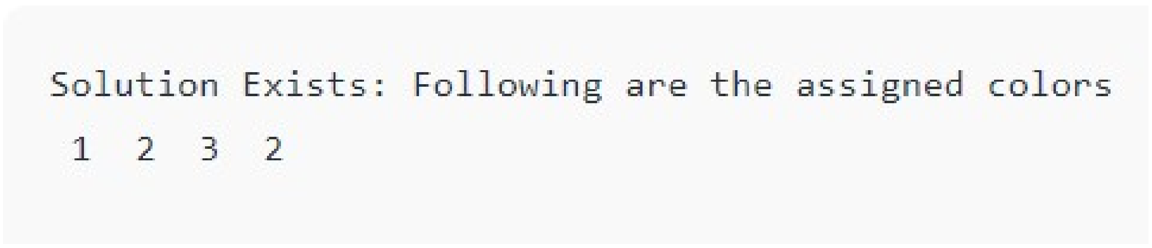
AIM:

ALGORITHM:

PROGRAM:

```
class Graph():
def  init (self, vertices): self.V = vertices
self.graph = [[0 for column in range(vertices)] for row in range(vertices)]

def isSafe(self, v, colour, c): for i in range(self.V):
if self.graph[v][i] == 1 and colour[i] == c: return False
return True

def graphColourUtil(self, m, colour, v): if v == self.V:
return True

for c in range(1, m+1):
if self.isSafe(v, colour, c): colour[v] = c
if self.graphColourUtil(m, colour, v+1): return True
colour[v] = 0 return False
def graphColouring(self, m): colour = [0] * self.V
if not self.graphColourUtil(m, colour, 0): return False
print("Solution exists and the following are the assigned colours:") for c in colour:
print(c, end=' ') return True

if  name    == ' main ': g = Graph(4)

g.graph = [ [0, 1, 1, 1],
[1, 0, 1, 0],
[1, 1, 0, 1],
[1, 0, 1, 0]
]
m = 3 g.graphColouring(m)
```

OUTPUT:



```
Solution Exists: Following are the assigned colors
 1  2  3  2
```

RESULT:

EX.N0: 7	MINMAX ALGORITHM
DATE:	

AIM:

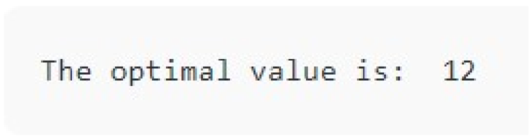
ALGORITHM:

PROGRAM:

```
import math
def minimax(curDepth, nodeIndex, maxTurn, scores, targetDepth): if curDepth == targetDepth:
return scores[nodeIndex]


if maxTurn:
return max(minimax(curDepth + 1, nodeIndex * 2, False, scores, targetDepth),
minimax(curDepth + 1, nodeIndex * 2 + 1, False, scores, targetDepth))

else:
return min(minimax(curDepth + 1, nodeIndex * 2, True, scores, targetDepth), minimax(curDepth
+ 1, nodeIndex * 2 + 1, True, scores, targetDepth))

scores = [3, 5, 2, 9, 12, 5, 23, 23]
```

```
treeDepth = int(math.log2(len(scores)))


print("The optimal value is:", end=" ") print(minimax(0, 0, True, scores, treeDepth))
```

<u>OUTPUT:</u>



```
The optimal value is:  12
```

<u>RESULT</u>:

EX.N0: 8	**ALPHA-BETA PRUNING**
DATE:	

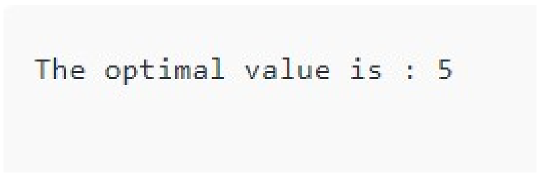
AIM:

ALGORITHM:

PROGRAM:

```
MAX, MIN = 1000, -1000

def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):
    if depth == 3:
        return values[nodeIndex]

    if maximizingPlayer:
        best = MIN
        for i in range(0, 2):
            val = minimax(depth + 1, nodeIndex * 2 + i, False, values, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                break
        return best
    else:
        best = MAX
        for i in range(0, 2):
            val = minimax(depth + 1, nodeIndex * 2 + i, True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                break
        return best

if __name__ == "__main__":
    values = [3, 5, 6, 9, 1, 2, 0, -1]
    print("The optimal value is:", minimax(0, 0, True, values, MIN, MAX))
```

OUTPUT:



```
The optimal value is : 5
```

RESULT:

EX.NO:9	**<u>INTRODUCTION TO PROLOG</u>**
DATE :	

PROLOG

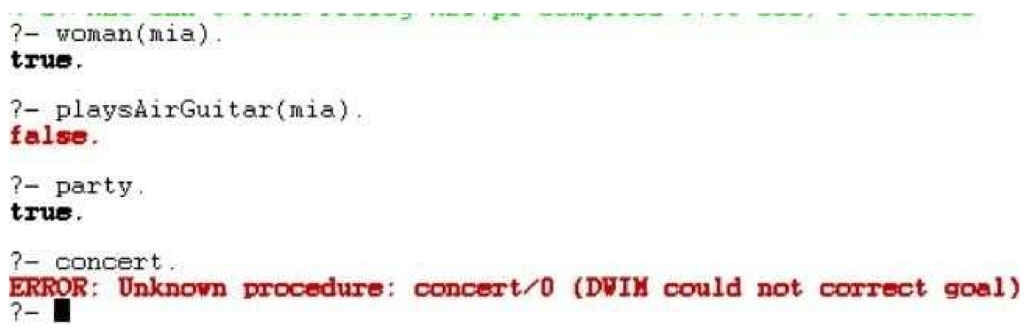
To learn PROLOG terminologies and write basic programs.

<u>TERMINOLOGIES</u>

SOURCECODE:

KB1:
woman(mia).
woman(jody).
woman(yolanda).
playsAirGuitar(jody).
party.
Query1:?-woman(mia). Query2:?-
playsAirGuitar(mia). Query3:?-
party.
Query4:?-concert.

OUTPUT:-

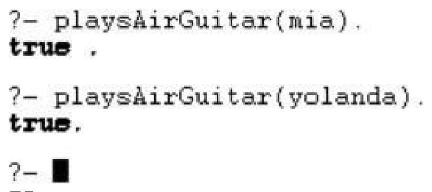


```
?- woman(mia).
true.

?- playsAirGuitar(mia).
false.

?- party.
true.

?- concert.
ERROR: Unknown procedure: concert/0 (DWIM could not correct goal)
?-
```

KB2:

happy(yolanda). listens2music(mia).
Listens2music(yolanda):-happy(yolanda).
playsAirGuitar(mia):-listens2music(mia).
playsAirGuitar(Yolanda):-listens2music(yolanda).

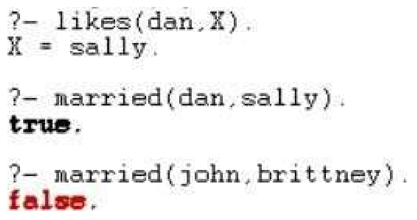
OUTPUT:-



```
?- playsAirGuitar(mia).
true .

?- playsAirGuitar(yolanda).
true.

?-
```

K
B likes(dan,sally).
3: 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:-

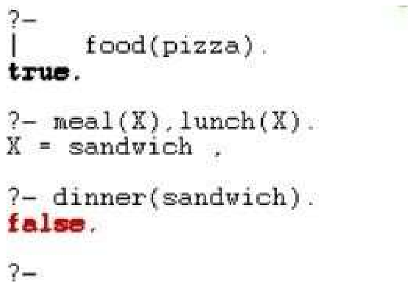


```
?- likes(dan,X).
X = sally.

?- married(dan,sally).
true.

?- married(john,brittney).
false.
```

<u>KB4:</u>

food(burger).
food(sandwich
). food(pizza).
lunch(sandwic
h).
dinner(pizza).
meal(X):-
food(X).

<u>OUTPUT:</u>

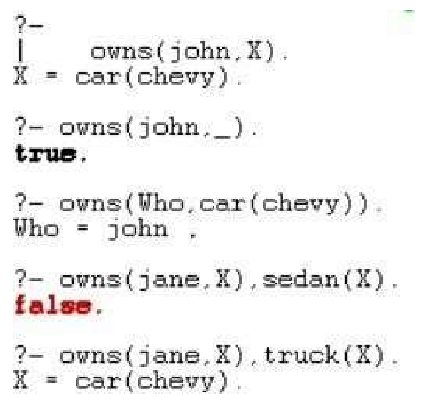


```
?-
|    food(pizza).
true.

?- meal(X),lunch(X).
X = sandwich .

?- dinner(sandwich).
false.

?-
```

<u>K</u>
<u>B</u>
<u>5:</u>

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



```
?-
|    owns(john,X).
X = car(chevy).

?- owns(john,_).
true.

?- owns(Who,car(chevy)).
Who = john .

?- owns(jane,X),sedan(X).
false.

?- owns(jane,X),truck(X).
X = car(chevy).
```

KB6: Find minimum maximum of two numbers

```
find_max(X,Y,X):-X>=Y,!.
find_max(X,Y,Y):-X<Y.
find_min(X,Y,X):-X=<Y,!.
find_min(X,Y,Y):-X>Y.
```

Output:

```
| ?- find_max(100,200,Max).
Max = 200
yes
| ?- find_max(40,10,Max).
Max = 40 yes
| ?- find_min(40,10,Min).
Min = 10 yes
| ?- find_min(100,200,Min).
Min = 100
yes
| ?-
```

KB7:

Here are some simple clauses.

likes(mary,food).

likes(mary,wine).

likes(john,wine).

likes(john,mary).

The following queries yield the specified answers.

| ?- likes(mary,food). yes.

| ?- likes(john,wine). yes.

| ?- likes(john,food). no.

RESULT:

EX.N0 : <u>10</u>	**<u>UNIFICATION AND RESOLUTION</u>**
DATE :	

<u>AIM:</u>

Example1: 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 looksfor the value of X asked in query and it returns answer X = janei.e.Instantiation - X is instantiated to jane.

Example2: 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 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 cannot 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 proposition al logic like this-

(1) strawberry_picking←pleasant

(2) happy←strawberry_picking

And again the sestatement scan be written in CNF like this-

(1) (strawberry_picking∨~pleasant)∧

(2) (happy∨~strawberry_picking)

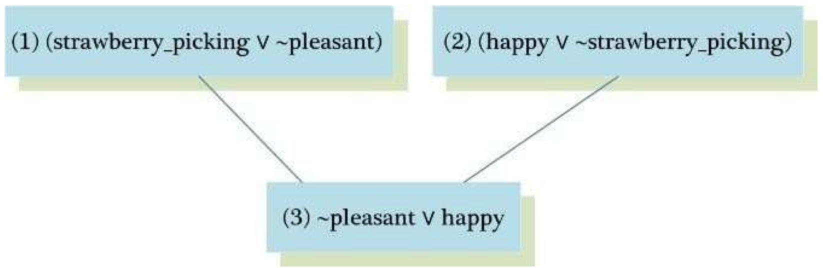
(3) By resolving these two clauses and cancelling out the conflicting terms ;strawberry_picking;and;~strawberry_picking;,wecanhaveonenew clause,

(4) ~pleasant ∨happy

How ?Seethefigureonright.

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

;pleasant→happy;or;happy←pleasant;i.e.If it is a pleasant day you are happy.


(1) (strawberry_picking ∨ ~pleasant)
(2) (happy ∨ ~strawberry_picking)
(3) ~pleasant ∨ happy

- But sometimes from the collection of the statements we have, we want to know the answer of this question- Is it possible to prove some other statements from what we actually know In order to prove this we need to make someinferences 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) andPart(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)

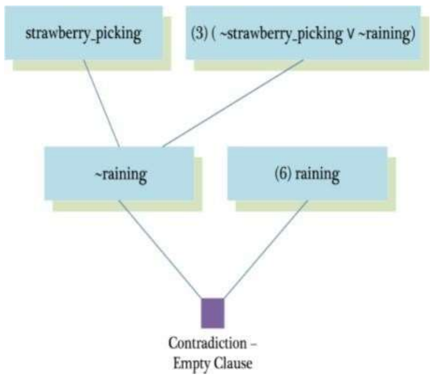
Part(IV):Otherstatementswewant toprovebyRefutation

(Goal 1) You are not doing strawberry picking. (Goal2)You will enjoy.

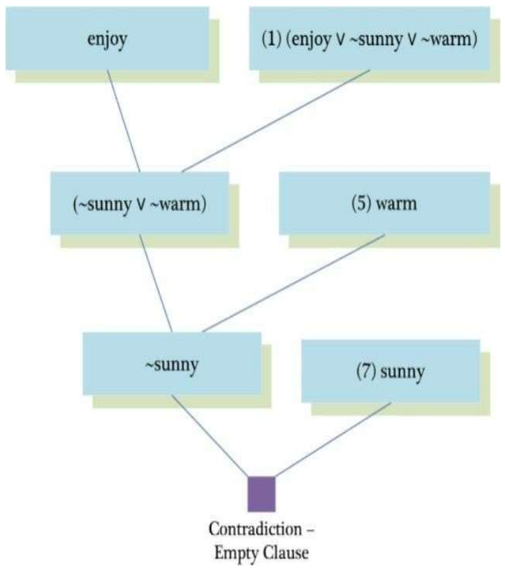
(Goal 3) Try it yourself : You will get wet.

Goal1:Youarenotdoingstrawberrypicking.

Prove :~strawberry_picking
Assume:strawberry_picking(negatethegoalandaddittogivenclauses).



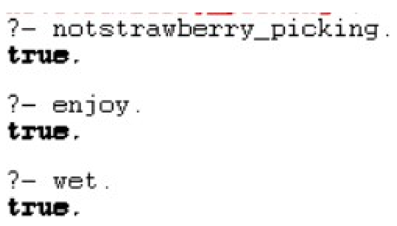
Goal2:You will enjoy. Prove :enjoy
Assume:~enjoy(negate the goal and add it to given clauses)



SOURCECODE:

enjoy:-sunny,warm. strawberrry_picking:-
warm,plesant. notstrawberry_picking:-raining. wet:-
raining.
warm. raining. sunny.

OUTPUT:



```
?- notstrawberry_picking.
true.

?- enjoy.
true.

?- wet.
true.
```

RESULT:

EX.N0 : 11	FUZZYLOGIC – IMAGE PROCESSING
DATE :	

Fuzzy inference system:

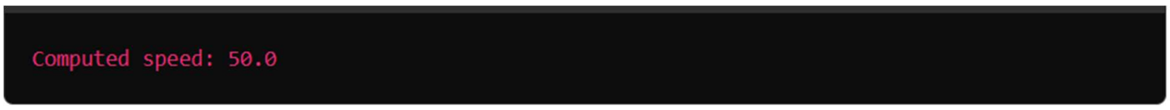
You can implement a fuzzy inference system in Python using the **scikit-fuzzy** library, which provides tools for fuzzy logic. Here's a simple example of how you might create a fuzzy inference system for controlling the speed of a car based on the distance to an obstacle:

```python
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

# Create fuzzy variables
distance = ctrl.Antecedent(np.arange(0, 11, 1), 'distance')
speed = ctrl.Consequent(np.arange(0, 101, 1), 'speed')

# Define membership functions for distance
distance['near'] = fuzz.trimf(distance.universe, [0, 0, 5])
distance['medium'] = fuzz.trimf(distance.universe, [0, 5, 10])
distance['far'] = fuzz.trimf(distance.universe, [5, 10, 10])

# Define membership functions for speed
speed['slow'] = fuzz.trimf(speed.universe, [0, 0, 50])
speed['medium'] = fuzz.trimf(speed.universe, [0, 50, 100])
speed['fast'] = fuzz.trimf(speed.universe, [50, 100, 100])

# Define rules
rule1 = ctrl.Rule(distance['near'], speed['slow'])
rule2 = ctrl.Rule(distance['medium'], speed['medium'])
rule3 = ctrl.Rule(distance['far'], speed['fast'])

# Create the control system
speed_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
car_speed = ctrl.ControlSystemSimulation(speed_ctrl)

# Input distance and compute speed
car_speed.input['distance'] = 7
car_speed.compute()

# Print the computed speed
print("Computed speed:", car_speed.output['speed'])
```

In this example, we first define the fuzzy variables **distance** and **speed** using the **Antecedent** and **Consequent** classes, respectively. We then define the membership functions for each variable using **fuzz.trimf**. Next, we define the rules that determine the speed based on the distance using the **ctrl.Rule** class. After defining the rules, we create the control system using **ctrl.ControlSystem** and **ctrl.ControlSystemSimulation**. Finally, we input a distance value and compute the speed using **compute()**, and print the computed speed.

This is a simple example to demonstrate the basic structure of a fuzzy inference system in Python using **scikit-fuzzy**. You can expand on this example by adding more variables, membership functions, and rules to create a more complex fuzzy inference system for your specific application.

OUTPUT:



```
Computed speed: 50.0
```

RESULT: