8 QUEENS PROBLEM

Name: VANDHANA P   Reg No.: 231801184

Program:

```
N= int(input("Enter the number of queens :"))
board = [[0]*N for _ in range(N)]
def attack(I,j):
        for k in range(0,N):
                if board[i][k]==1 or board[k][j]==1:
                        return True
                for k in range(0,N):
                        for l in range(0,N):
                                if(k+l==i+j)or (k-l==i-j):
                                        if board[k][l]==1:
                                                return True
          return False
def N_queens(n):
        if n==0:
                return True
        for I in range (0,N):
                for j in range(0,N):
                        if(not(attack(I,j)))and (board[i][j]!=1):
                                board[i][j]=1
                                if N_queens(n-1)==True:
                                        return True
                                board[i][j]=0
         return False
```

```
N_queens(N)

for I in board:

        print(i)
```

OUTPUT:

```
==== RESTART: C:/Users/acer28/AppData/Local/Programs/Python/Python39/back.py ===
Enter the number of queens :8
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
>>>
```

WATER JUG PROGRAM USING BFS

Name: VANDHANA P **Reg No.:231801184**

PROGRAM:

```
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() # Use popleft to get the first element (breadth-first)
         if (u[0], u[1]) in m:
                continue
         if u[0] > a or u[1] > b or u[0] < 0 or u[1] < 0:
                continue
        path.append([u[0], u[1]])
        m[(u[0], u[1])] = 1
        if u[0] == target or u[1] == target:
                isSolvable = True
                if u[0] == target:
                        if u[1] != 0:
                                path.append([u[0], 0])
                        else:
                                if u[0] != 0:
                                        path.append([0, u[1]])
                        sz = len(path) for I in range(sz):
```

```
                    print(“(“, path[i][0], “,”, path[i][1], “)”)
                    return # Exiting the function after finding the solution
               q.append([u[0], b])
               q.append([a, u[1]])
               for ap in range(max(a, b) + 1):
                    c = u[0] + ap
                    d = u[1] – ap
                    if c == a or (d == 0 and d >= 0):
                         q.append([c, d])
                         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])
                    q.append([0, b])
                    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 from intial state to solution state:
( 0 , 0 )
( 0 , 3 )
( 4 , 0 )
( 4 , 3 )
( 3 , 0 )
( 1 , 3 )
( 3 , 3 )
( 4 , 2 )
( 4 , 0 )
>>>
```

WATER JUG PROGRAM USING DFS

Name: VANDHANA P **Reg No.:231801184**

PROGRAM:

```
from collections import deque
def DFS(a, b, target):
    m = {}
    isSolvable = False
    path = []
    q = deque()
    q.append((0, 0))
    while len(q) > 0:
        u = q.popleft()
        if (u[0], u[1]) in m:
            continue
        if u[0] > a or u[1] > b or u[0] < 0 or u[1] < 0:
            continue
        path.append([u[0], u[1]])
        m[(u[0], u[1])] = 1
        if u[0] == target or u[1] == target:
            isSolvable = True
            if u[0] == target:
                if u[1] != 0:
                    path.append([u[0], 0])
            else:
                if u[0] != 0:
                    path.append([0, u[1]])
                    q.append([u[0], b])
```

```
                                q.append([a, u[1]])
                for ap in range(max(a, b) + 1):
                    c = u[0] + ap
                    d = u[1] – ap
                    if c == a or (d == 0 and d >= 0):
                        q.append([c, d])
                        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])
                        q.append([0, b])
                    if not isSolvable:
                        print("No solution")
                    else:
                        for I in range(len(path)):
                            print("(", path[i][0], ",", path[i][1], ")")
Jug1, Jug2, target = 4, 3, 2
print("Path from initial state to solution state:")
 DFS(Jug1, Jug2, target)
```

OUTPUT:

```
=========== RESTART: C:/Users/acer28/Desktop/231801177 AIDS C/DFS.py ===========
Path from intial state to solution state:
( 0 , 0 )
( 0 , 3 )
( 4 , 0 )
( 4 , 3 )
( 3 , 0 )
( 3 , 3 )
( 4 , 2 )
>>>
```

A* SEARCH ALGORITHM

Name: VANDHANA P **Reg No.: 231801184**

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
```

OUTPUT:

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

=== Code Execution Successful ===
```

AO* SEARCH ALGORITHM

Name: VANDHANA P   Reg no.:231801184

```
import heapq
 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
                de f_value(self):
                        return self.g_value + self.h_value
                def ao_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
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)
if __name__ == "__main__":
    initial_state = (0, 0)
    path = ao_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)]

=== Code Execution Successful ===
```

RECURSIVE BREADTH FIRST SEARCH

Name: VANDHANA P   Reg No.:231801184

PROGRAM:

```
class Node:
    def __init__(self, state, parent=None, cost=0, heuristic=0):
        self.state = state
        self.parent = parent
        self.cost = cost
        self.heuristic = heuristic
        self.f = cost + heuristic
    def is_goal(state, goal):
        return state == goal
    def generate_successors(node, goal):
        successors = []
        for i in range(node.state + 1, goal + 1):
        successors.append(Node(i, node, node.cost + 1, heuristic(i, goal)))
        return successors
    def heuristic(state, goal):
        return abs(goal - state)
    def rbfs(node, f_limit, goal):
        if is_goal(node.state, goal):
            return node successors = generate_successors(node, goal)
        if not successors:
            return None
        while True:
            successors.sort(key=lambda x: x.f)
            best = successors[0]
```

```
if best.f > f_limit:
    return None
if len(successors) > 1:
    alternative = successors[1].f
else:
    alternative = float('inf')
    result = rbfs(best, min(f_limit, alternative), goal)
    if result is not None:
        return result initial_state = 0 goal_state = 5 initial_node =
    Node(initial_state, None, 0, heuristic(initial_state, goal_state))
    solution = rbfs(initial_node, float('inf'), goal_state)
    if solution is not None:
        path = []
    while solution is not None:
        path.append(solution.state)
        solution = solution.parent path.reverse()
        print("RBFS Path:", path)
        else:
            print("No solution found.")
```

OUTPUT:

```
================== RESTART: C:/Users/Lenovo/Documents/rbfs.py =======
RBFS Path: [0, 5]
```

CSP MAP COLOURING

Name: VANDHANA P Reg No.:231801184

PROGRAM:

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for _ in range(vertices)] for _ 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
    def graphColouring(self, m):
        colour = [0] * self.V
        if not self.graphColourUtil(m, colour, 0):
            print("Solution does not exist")
```

```
                return False
            print("Solution exists and 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 and Following are the assigned colours:
1 2 3 2
```

MIN MAX ALGORITHM

Name: VANDHANA P Reg No.: 231801184

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 = math.log(len(scores), 2)
print("The optimal value is:", end=" ")
print(minimax(0, 0, True, scores, treeDepth))
```

OUTPUT:

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

ALPHA BETA PRUNING

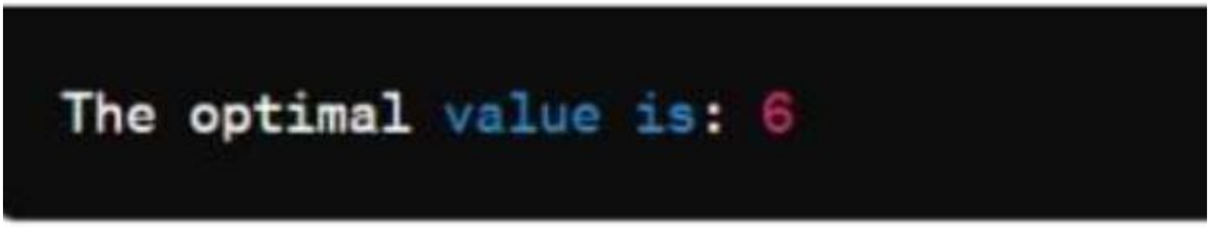
Name: VANDHANA P Reg No.: 231801184

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:



INTRODUCTION TO PROLOG

Name: VANDHANA P **Reg No.:231801184**

```
% Rules to find the minimum of two numbers
 min(X, Y, Min) :-
            X =< Y,
           Min is X.
min(X, Y, Min
           X > Y,
           Min is Y.


 % Rules to find the maximum of two numbers
max(X, Y, Max) :-
            X >= Y,
           Max is X.
max(X, Y, Max) :-
           X < Y,
           Max is Y.
```

OUTPUT:

```
% Existing facts

likes(mary, food).

likes(mary, wine).

likes(john, wine).

likes(john, mary).

% New facts

likes(john, X) :- likes(mary, X).        % John likes anything that Mary like

likes(john, Y) :- likes(Y, wine).        % John likes anyone who likes wine

likes(john, Z) :- likes(Z, Z).           % John  likes anyone who likes themselves
```

query:

```
?- likes(mary,food).

yes.

?- likes(john,wine).

yes.

?- likes(john,food).

no.

?- likes(john,X).

X = wine .

?- likes(john,Y).

Y = wine .

?- likes(john,Z).

Z = wine
```

UNIFICATION AND RESOLUTION

Name: VANDHANA P **Reg No.:231801184**

PROGRAM:

enjoy:-
sunny,warm.
strawberrry_pic
king;-
warm,plesant.
notstrawberry_p
icking:-raining.
wet:-raining.

warm.

raining .

sunny.

OUTPUT:

```
?- notstrawberry_picking.
true.

?- enjoy.
true.

?- wet.
true.
```

Fuzzy inference system

Name : VANDHANA P **Reg No.: 231801184**

PROGRAM:

```
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'])
```

OUTPUT:

```
Computed speed: 50.0
```