Artificial intelligence pratical program (normal)

15.decision tree

```
# Import necessary libraries
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create a Decision Tree classifier
clf = DecisionTreeClassifier()

# Train the classifier on the training data
clf.fit(X_train, y_train)

# Make predictions on the test data
y_pred = clf.predict(X_test)
```

```
# Evaluate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

3. water jug

```
from collections import deque
def Solution(a, b, target):
    m = {}
    isSolvable = False
    path = []

    q = deque()

    #Initializing with jugs being empty
    q.append((0, 0))

    while (len(q) > 0):

        # Current state
        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]])

    sz = len(path)
    for i in range(sz):
        print("(", path[i][0], ",",
            path[i][1], ")")
    break

q.append([u[0], b]) # Fill Jug2
q.append([a, u[1]]) # Fill Jug1

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("Solution not possible")

if _name_ == '_main_':

    Jug1, Jug2, target = 4, 3, 2
    print("Path from initial state "
        "to solution state ::")

    Solution(Jug1, Jug2, target)
```

output:

Path from initial state to solution state ::

(0 , 0)

(0 , 3)

(4 , 0)

(4 , 3)

(3 , 0)

(1 , 3)

(3 , 3)

(4 , 2)

(0 , 2)

>>>

3. 8 puzzle problem

```
# Python code to display the way from the root

# node to the final destination node for N*N-1 puzzle

# algorithm by the help of Branch and Bound technique

# The answer assumes that the instance of the

# puzzle can be solved


# Importing the 'copy' for deepcopy method

import copy


# Importing the heap methods from the python

# library for the Priority Queue

from heapq import heappush, heappop
```

```
# This particular var can be changed to transform
# the program from 8 puzzle(n=3) into 15
# puzzle(n=4) and so on ...
n = 3

# bottom, left, top, right
rows = [ 1, 0, -1, 0 ]
cols = [ 0, -1, 0, 1 ]

# creating a class for the Priority Queue
class priorityQueue:

    # Constructor for initializing a
    # Priority Queue
    def _init_(self):
        self.heap = []

    # Inserting a new key 'key'
    def push(self, key):
        heappush(self.heap, key)

    # funct to remove the element that is minimum,
    # from the Priority Queue
    def pop(self):
        return heappop(self.heap)
```

```
    # funct to check if the Queue is empty or not
    def empty(self):
        if not self.heap:
            return True
        else:
            return False

# structure of the node
class nodes:

    def _init_(self, parent, mats, empty_tile_posi,
               costs, levels):

        # This will store the parent node to the
        # current node And helps in tracing the
        # path when the solution is visible
        self.parent = parent

        # Useful for Storing the matrix
        self.mats = mats

        # useful for Storing the position where the
        # empty space tile is already existing in the matrix
        self.empty_tile_posi = empty_tile_posi
```

```
        # Store no. of misplaced tiles
        self.costs = costs

        # Store no. of moves so far
        self.levels = levels

    # This func is used in order to form the
    # priority queue based on
    # the costs var of objects
    def _lt_(self, nxt):
        return self.costs < nxt.costs

# method to calc. the no. of
# misplaced tiles, that is the no. of non-blank
# tiles not in their final posi
def calculateCosts(mats, final) -> int:

    count = 0
    for i in range(n):
        for j in range(n):
            if ((mats[i][j]) and
                (mats[i][j] != final[i][j])):
                count += 1

    return count
```

```
def newNodes(mats, empty_tile_posi, new_empty_tile_posi,
            levels, parent, final) -> nodes:

    # Copying data from the parent matrixes to the present matrixes
    new_mats = copy.deepcopy(mats)

    # Moving the tile by 1 position
    x1 = empty_tile_posi[0]
    y1 = empty_tile_posi[1]
    x2 = new_empty_tile_posi[0]
    y2 = new_empty_tile_posi[1]
    new_mats[x1][y1], new_mats[x2][y2] = new_mats[x2][y2], new_mats[x1][y1]

    # Setting the no. of misplaced tiles
    costs = calculateCosts(new_mats, final)

    new_nodes = nodes(parent, new_mats, new_empty_tile_posi,
                costs, levels)
    return new_nodes

# func to print the N by N matrix
def printMatsrix(mats):

    for i in range(n):
        for j in range(n):
            print("%d " % (mats[i][j]), end = " ")
```

```
        print()

# func to know if (x, y) is a valid or invalid
# matrix coordinates
def isSafe(x, y):

    return x >= 0 and x < n and y >= 0 and y < n

# Printing the path from the root node to the final node
def printPath(root):

    if root == None:
        return

    printPath(root.parent)
    printMatsrix(root.mats)
    print()

# method for solving N*N - 1 puzzle algo
# by utilizing the Branch and Bound technique. empty_tile_posi is
# the blank tile position initially.
def solve(initial, empty_tile_posi, final):

    # Creating a priority queue for storing the live
    # nodes of the search tree
```

```
pq = priorityQueue()

# Creating the root node
costs = calculateCosts(initial, final)
root = nodes(None, initial,
          empty_tile_posi, costs, 0)

# Adding root to the list of live nodes
pq.push(root)

# Discovering a live node with min. costs,
# and adding its children to the list of live
# nodes and finally deleting it from
# the list.
while not pq.empty():

   # Finding a live node with min. estimatsed
   # costs and deleting it form the list of the
   # live nodes
   minimum = pq.pop()

   # If the min. is ans node
   if minimum.costs == 0:

      # Printing the path from the root to
      # destination;
```

```
            printPath(minimum)
            return

        # Generating all feasible children
        for i in range(n):
            new_tile_posi = [
                minimum.empty_tile_posi[0] + rows[i],
                minimum.empty_tile_posi[1] + cols[i], ]

            if isSafe(new_tile_posi[0], new_tile_posi[1]):

                # Creating a child node
                child = newNodes(minimum.mats,
                                minimum.empty_tile_posi,
                                new_tile_posi,
                                minimum.levels + 1,
                                minimum, final,)

                # Adding the child to the list of live nodes
                pq.push(child)

# Main Code

# Initial configuration
# Value 0 is taken here as an empty space
initial = [ [ 1, 2, 3 ],
```

```
          [ 5, 6, 0 ],
          [ 7, 8, 4 ] ]

# Final configuration that can be solved
# Value 0 is taken as an empty space
final = [ [ 1, 2, 3 ],
        [ 5, 8, 6 ],
        [ 0, 7, 4 ] ]

# Blank tile coordinates in the
# initial configuration
empty_tile_posi = [ 1, 2 ]

# Method call for solving the puzzle
solve(initial, empty_tile_posi, final)
```

output:

```
Step: 1
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
------
Step: 2
[1, 2, 3]
[4, 5, 6]
```

[7, 0, 8]

Step: 3

[1, 2, 3]

[4, 5, 6]

[7, 8, 0]

4. 8 queen problem

```
# Taking number of queens as input from user
print ("Enter the number of queens")
N = int(input())

# here we create a chessboard
# NxN matrix with all elements set to 0
board = [[0]*N for _ in range(N)]

def attack(i, j):
    #checking vertically and horizontally
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True
    #checking diagonally
    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:

```
# Taking number of queens as input from user
print ("Enter the number of queens")
N = int(input())
```

```
# here we create a chessboard
# NxN matrix with all elements set to 0
board = [[0]*N for _ in range(N)]

def attack(i, j):
    #checking vertically and horizontally
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True
    #checking diagonally
    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:

[1, 2, 3]

[4, 0, 5]

[6, 7, 8]

Solving...

Move: Left

[1, 2, 3]

[0, 4, 5]

[6, 7, 8]

Move: Up

[0, 2, 3]

[1, 4, 5]

[6, 7, 8]

Move: Right

[2, 0, 3]

[1, 4, 5]

[6, 7, 8]

Move: Down

[2, 4, 3]

[1, 0, 5]

[6, 7, 8]

Move: Left

[2, 4, 3]

[0, 1, 5]

[6, 7, 8]

Move: Up

[0, 4, 3]

[2, 1, 5]

[6, 7, 8]

Move: Right

[4, 0, 3]

[2, 1, 5]

[6, 7, 8]

Move: Right

[4, 3, 0]

[2, 1, 5]

[6, 7, 8]

Move: Down

[4, 3, 5]

[2, 1, 0]

[6, 7, 8]

Move: Left

[4, 3, 5]

[2, 0, 1]

[6, 7, 8]

Move: Up

[4, 0, 5]

[2, 3, 1]

[6, 7, 8]

Move: Right

[0, 4, 5]

[2, 3, 1]

[6, 7, 8]

Move: Right

[2, 4, 5]

[0, 3, 1]

[6, 7, 8]

Move: Down

[2, 4, 5]

[3, 0, 1]

[6, 7, 8]

Move: Left

[2, 4, 5]

[3, 1, 0]

[6, 7, 8]

Move: Left

[2, 4, 5]

[1, 3, 0]

[6, 7, 8]

Move: Up

[2, 4, 0]

[1, 3, 5]

[6, 7, 8]

Move: Right

[2, 0, 4]

[1, 3, 5]

[6, 7, 8]

Move: Right

[0, 2, 4]

[1, 3, 5]

[6, 7, 8]

Move: Down

[1, 2, 4]

[0, 3, 5]

[6, 7, 8]

Move: Left

[1, 2, 4]

[3, 0, 5]

[6, 7, 8]

Move: Left

[1, 2, 4]

[3, 5, 0]

[6, 7, 8]

Move: Up

[1, 2, 0]

[3, 5, 4]

[6, 7, 8]

Move: Right

[1, 0, 2]

[3, 5, 4]

[6, 7, 8]

Move: Down

[1, 5, 2]

[3, 0 ,4]

[6, 7 , 8]

5. **MAP COLOURING to implement CSP**

```
class MapColoringCSP:
    def __init__(self, variables, domains, constraints):
        self.variables = variables
        self.domains = domains
        self.constraints = constraints

    def is_consistent(self, variable, value, assignment):
        for neighbor in self.constraints.get(variable, []):
            if neighbor in assignment and assignment[neighbor] == value:
                return False
        return True

    def backtracking_search(self, assignment={}):
        if len(assignment) == len(self.variables):
            return assignment  # Solution found
        unassigned_variables = [var for var in self.variables if var not in
assignment]
```

```
        current_variable = unassigned_variables[0]

        for value in self.domains[current_variable]:
            if self.is_consistent(current_variable, value, assignment):
                assignment[current_variable] = value
                result = self.backtracking_search(assignment)
                if result is not None:
                    return result
                del assignment[current_variable]

        return None  # No solution found

def main():
    variables = ['WA', 'NT', 'SA', 'Q', 'NSW', 'V', 'T']
    domains = {var: ['R', 'G', 'B'] for var in variables}  # Possible colors: Red,
Green, Blue

    constraints = {
        'WA': ['NT', 'SA'],
        'NT': ['WA', 'SA', 'Q'],
        'SA': ['WA', 'NT', 'Q', 'NSW', 'V'],
        'Q': ['NT', 'SA', 'NSW'],
        'NSW': ['Q', 'SA', 'V'],
        'V': ['SA', 'NSW']
    }

    map_coloring_csp = MapColoringCSP(variables, domains, constraints)
```

```
    solution = map_coloring_csp.backtracking_search()

    if solution:
        print("Solution found:")
        for variable, color in solution.items():
            print(f"{variable}: {color}")
    else:
        print("No solution found.")

if __name__ == "__main__":
    main()
```

OUTPUT:

Solution found:

WA: R

NT: G

SA: B

Q: R

N.SW: G

V: R

T: R

6. minmax algorithm

```
# Tic-Tac-Toe board represented as a list
board = [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
```

```
# Function to print the Tic-Tac-Toe board
def print_board():
    print(f"{board[0]} | {board[1]} | {board[2]}")
    print("---------")
    print(f"{board[3]} | {board[4]} | {board[5]}")
    print("---------")
    print(f"{board[6]} | {board[7]} | {board[8]}")

# Function to check if the current player has won
def is_winner(player):
    # Check rows, columns, and diagonals
    for i in range(3):
        if all(board[i*3 + j] == player for j in range(3)) or \
           all(board[j*3 + i] == player for j in range(3)):
            return True
    if all(board[i] == player for i in [0, 4, 8]) or \
       all(board[i] == player for i in [2, 4, 6]):
        return True
    return False

# Function to check if the board is full
def is_board_full():
    return ' ' not in board

# Minimax algorithm
```

```python
def minimax(depth, maximizing_player):
    if is_winner('X'):
        return -1
    elif is_winner('O'):
        return 1
    elif is_board_full():
        return 0

    if maximizing_player:
        max_eval = float('-inf')
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'O'
                eval = minimax(depth + 1, False)
                board[i] = ' '
                max_eval = max(max_eval, eval)
        return max_eval
    else:
        min_eval = float('inf')
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'X'
                eval = minimax(depth + 1, True)
                board[i] = ' '
                min_eval = min(min_eval, eval)
        return min_eval
```

```
# Function to find the best move using Minimax
def find_best_move():
    best_val = float('-inf')
    best_move = -1

    for i in range(9):
        if board[i] == ' ':
            board[i] = 'O'
            move_val = minimax(0, False)
            board[i] = ' '

            if move_val > best_val:
                best_val = move_val
                best_move = i

    return best_move

# Main game loop
while True:
    print_board()

    # Player's move
    player_move = int(input("Enter your move (1-9): ")) - 1
    if board[player_move] == ' ':
        board[player_move] = 'X'
```

```
else:
    print("Invalid move. Try again.")
    continue

# Check if player wins
if is_winner('X'):
    print_board()
    print("Congratulations! You win!")
    break

# Check if the board is full
if is_board_full():
    print_board()
    print("It's a tie!")
    break

# AI's move
ai_move = find_best_move()
board[ai_move] = 'O'

# Check if AI wins
if is_winner('O'):
    print_board()
    print("You lose! Better luck next time.")
    break
```

```
    # Check if the board is full
    if is_board_full():
        print_board()
        print("It's a tie!")
        break
```

OUTPUT:

```
  |   |
---------
  |   |
---------
  |   |
Enter your move (1-9): 1
X |   |
---------
  | O |
---------
  |   |
Enter your move (1-9): 2
X | X | O
---------
  | O |
---------
  |   |
Enter your move (1-9): 3
```

Invalid move. Try again.

X | X | O

  | O |

  | |

Enter your move (1-9): 7

X | X | O

O | O |

X | |

Enter your move (1-9): 6

X | X | O

O | O | X

X | O |

7. tic tac toe game

```
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 5)
```

```
def check_winner(board):
    # Check rows
    for row in board:
        if row.count(row[0]) == len(row) and row[0] != ' ':
            return True

    # Check columns
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] and board[0][col] != ' ':
            return True

    # Check diagonals
    if board[0][0] == board[1][1] == board[2][2] and board[0][0] != ' ':
        return True
    if board[0][2] == board[1][1] == board[2][0] and board[0][2] != ' ':
        return True

    return False

def is_board_full(board):
    for row in board:
        if ' ' in row:
            return False
    return True

def play_game():
```

```
    board = [[' ' for _ in range(3)] for _ in range(3)]
    current_player = 'X'

    while True:
        print_board(board)

        row = int(input("Enter the row (0, 1, or 2): "))
        col = int(input("Enter the column (0, 1, or 2): "))

        if board[row][col] == ' ':
            board[row][col] = current_player

            if check_winner(board):
                print_board(board)
                print(f"Player {current_player} wins!")
                break
            elif is_board_full(board):
                print_board(board)
                print("It's a tie!")
                break

            current_player = 'O' if current_player == 'X' else 'X'
        else:
            print("Cell already occupied. Try again.")

if __name__ == "__main__":
```

```
    play_game()
```

OUTPUT:

```
 |  |
-----
  |  |
-----
  |  |
-----
Enter the row (0, 1, or 2): 1
Enter the column (0, 1, or 2): 2
  |  |
-----
  |  | X
-----
  |  |
-----
Enter the row (0, 1, or 2): 0
Enter the column (0, 1, or 2): 1
  | O |
-----
  |  | X
-----
  |  |
```

8.a*algorithm

```
import heapq

def astar(graph, start, end):
    open_list = []
    closed_set = set()
    heapq.heappush(open_list, (0, start))
    while open_list:
        cost, current_node = heapq.heappop(open_list)
        if current_node == end:
            return cost
        if current_node in closed_set:
            continue
        closed_set.add(current_node)
        for neighbor, neighbor_cost in graph[current_node].items():
            if neighbor not in closed_set:
                heapq.heappush(open_list, (cost + neighbor_cost, neighbor))
    return float('inf')

# Example Usage
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}
```

```
start_node = 'A'
end_node = 'D'
shortest_path_cost = astar(graph, start_node, end_node)
print(f"The shortest path cost from {start_node} to {end_node} is: {shortest_path_cost}")
```

output:

The shortest path cost from A to D is: 4

9. alpha beta punning

```
class GameState:
    def __init__(self, value):
        self.value = value
        self.children = []

def minimax_alpha_beta(node, depth, alpha, beta, maximizing_player):
    if depth == 0 or not node.children:
        return node.value

    if maximizing_player:
        max_eval = float('-inf')
        for child in node.children:
            eval_child = minimax_alpha_beta(child, depth - 1, alpha, beta, False)
            max_eval = max(max_eval, eval_child)
            alpha = max(alpha, eval_child)
```

```
            if beta <= alpha:
                break  # Beta pruning
        return max_eval
    else:
        min_eval = float('inf')
        for child in node.children:
            eval_child = minimax_alpha_beta(child, depth - 1, alpha, beta, True)
            min_eval = min(min_eval, eval_child)
            beta = min(beta, eval_child)
            if beta <= alpha:
                break  # Alpha pruning
        return min_eval

# Example usage
if __name__ == "__main__":
    # Construct a sample game tree
    root = GameState(3)
    child1 = GameState(5)
    child2 = GameState(2)
    child3 = GameState(9)

    root.children = [child1, child2, child3]

    child1.children = [GameState(7), GameState(8)]
    child2.children = [GameState(1), GameState(4)]
    child3.children = [GameState(6), GameState(3)]
```

```
    # Set initial alpha and beta values
    alpha = float('-inf')
    beta = float('inf')

    # Call alpha-beta pruning function
    result = minimax_alpha_beta(root, 3, alpha, beta, True)

    print("Optimal value:", result)
```

output:

Optimal value: 7

10. breadth first search

```
class Graph:
    def __init__(self):
        self.graph = {}
    def add_edge(self, node, adjacent):
        if node not in self.graph:
            self.graph[node] = []
        self.graph[node].append(adjacent)
    def bfs(self, start):
        visited = set()
        queue = [start]
        visited.add(start)
```

```
        while queue:
            vertex = queue.pop(0)
            print(vertex, end=" ")
            for neighbor in self.graph.get(vertex, []):
                if neighbor not in visited:
                    queue.append(neighbor)
                    visited.add(neighbor)
if __name__ == "__main__":
    g = Graph()
    g.add_edge(0, 1)
    g.add_edge(0, 2)
    g.add_edge(1, 2)
    g.add_edge(2, 0)
    g.add_edge(2, 3)
    g.add_edge(3, 3)
    print("BFS Traversal:")
    g.bfs(2)  # Starting BFS from vertex 2
```

output:

BFS Traversal:

2 0 3 1

11. cript arithmetic prblm

```
from ortools.sat.python import cp_model
model = cp_model.CpModel()
```

```
base = 10

c = model.NewIntVar(1, 9, 'C')
p = model.NewIntVar(0, 9, 'P')
i = model.NewIntVar(1, 9, 'I')
s = model.NewIntVar(0, 9, 'S')
f = model.NewIntVar(1, 9, 'F')
u = model.NewIntVar(0, 9, 'U')
n = model.NewIntVar(0, 9, 'N')
t = model.NewIntVar(1, 9, 'T')
r = model.NewIntVar(0, 9, 'R')
e = model.NewIntVar(0, 9, 'E')

# List of variables
letters = [c, p, i, s, f, u, n, t, r, e]
# Define the constraints
model.AddAllDifferent(letters)

# CP + IS + FUN = TRUE
model.Add(c * 10 + p + i * 10 + s + f * base**2 + u * 10 + n == t * 10**3 + r *
10**2 + u * 10 + e)
# Solution Printer Class
class VarArraySolutionPrinter(cp_model.CpSolverSolutionCallback):

    def __init__(self, variables):
        cp_model.CpSolverSolutionCallback.__init__(self)
        self.__variables = variables
```

```python
        self.__solution_count = 0

    def on_solution_callback(self):
        self.__solution_count += 1
        for v in self.__variables:
            print('%s=%i ' %(v, self.Value(v)), end='')
        print()

    def solution_count(self):
        return self.__solution_count
# Solution Printer Class
class VarArraySolutionPrinter(cp_model.CpSolverSolutionCallback):

    def __init__(self, variables):
        cp_model.CpSolverSolutionCallback.__init__(self)
        self.__variables = variables
        self.__solution_count = 0

    def on_solution_callback(self):
        self.__solution_count += 1
        for v in self.__variables:
            print('%s=%i ' %(v, self.Value(v)), end='')
        print()

    def solution_count(self):
        return self.__solution_count
```

12. deapth first search

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, node, adjacent):
        if node not in self.graph:
            self.graph[node] = []
        self.graph[node].append(adjacent)

    def dfs_util(self, vertex, visited):
        visited.add(vertex)
        print(vertex, end=' ')

        for neighbor in self.graph.get(vertex, []):
            if neighbor not in visited:
                self.dfs_util(neighbor, visited)

    def dfs(self, start):
        visited = set()
        self.dfs_util(start, visited)

# Example usage:
if __name__ == "__main__":
```

```
    g = Graph()
    g.add_edge(0, 1)
    g.add_edge(0, 2)
    g.add_edge(1, 2)
    g.add_edge(2, 0)
    g.add_edge(2, 3)
    g.add_edge(3, 3)

    print("DFS Traversal:")
    g.dfs(2)
```

output:

```
DFS Traversal:
2 0 1 3
>>>
```

13. feed forward neural network

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def feed_forward_nn(input_data, weights, biases):
    layer_input = input_data
    for w, b in zip(weights, biases):
        layer_output = np.dot(w, layer_input) + b
```

```
        layer_input = sigmoid(layer_output)
    return layer_input

# Example Usage
input_data = np.array([0.1, 0.2, 0.7])
weights = [np.array([[0.1, 0.2, 0.3], [0.2, 0.3, 0.4], [0.3, 0.4, 0.5]]),
           np.array([[0.5, 0.6, 0.7], [0.6, 0.7, 0.8]])]
biases = [np.array([0.1, 0.2, 0.3]), np.array([0.4, 0.5])]

output = feed_forward_nn(input_data, weights, biases)
print(output)
```

output:

```
Epoch 1/100
120/120 [==============================] - 1s 1ms/step - loss: 1.1951 - accuracy: 0.3000
Epoch 2/100
120/120 [==============================] - 0s 591us/step - loss: 0.9777 - accuracy: 0.6750
...
Epoch 100/100
120/120 [==============================] - 0s 458us/step - loss: 0.0650 - accuracy: 0.9750
Accuracy: 0.9666666666666667
```

14. travelling salesman prblm

```
import itertools

def tsp_brute_force(graph, start):
    all_nodes = set(graph.keys())
    all_nodes.remove(start)
    min_cost = float('inf')
    best_path = None

    for path in itertools.permutations(all_nodes):
        path = (start,) + path + (start,)
        cost = sum(graph[path[i]][path[i + 1]] for i in range(len(path) - 1))

        if cost < min_cost:
            min_cost = cost
            best_path = path

    return best_path, min_cost

# Example Usage
graph = {
    'A': {'A': 0, 'B': 2, 'C': 9, 'D': 6},
    'B': {'A': 1, 'B': 0, 'C': 3, 'D': 7},
    'C': {'A': 4, 'B': 5, 'C': 0, 'D': 8},
```

```
    'D': {'A': 3, 'B': 6, 'C': 2, 'D': 0}
}

start_node = 'A'
best_path, min_cost = tsp_brute_force(graph, start_node)
print("Best Path:", best_path)
print("Minimum Cost:", min_cost)
```

output:

```
Best Path: ('A', 'D', 'C', 'B', 'A')
Minimum Cost: 14
>>>
```

15. vaccum cleaner prblm

```
import random
class VacuumCleaner:
    def __init__(self, grid_size):
        self.grid_size = grid_size
        self.current_row = random.randint(0, grid_size - 1)
        self.current_col = random.randint(0, grid_size - 1)
        self.grid = [[random.choice([0, 1]) for _ in range(grid_size)] for _ in
range(grid_size)]
    def clean(self):
        print("Initial Grid:")
        self.print_grid()
```

```
        while True:
            print(f"Current Position: ({self.current_row}, {self.current_col})")
            if self.grid[self.current_row][self.current_col] == 1:
                print("Cell is dirty. Cleaning...")
                self.grid[self.current_row][self.current_col] = 0
            else:
                print("Cell is already clean.")
            self.move_randomly()
            print("\nUpdated Grid:")
            self.print_grid()
            if self.check_all_clean():
                print("All cells are clean. Cleaning process completed.")
                break
    def move_randomly(self):
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]  # Right, Left, Down, Up
        random_direction = random.choice(directions)
        new_row = self.current_row + random_direction[0]
        new_col = self.current_col + random_direction[1]
        if 0 <= new_row < self.grid_size and 0 <= new_col < self.grid_size:
            self.current_row = new_row
            self.current_col = new_col
    def print_grid(self):
        for row in self.grid:
            print(row)
    def check_all_clean(self):
        for row in self.grid:
```

```
            if 1 in row:
                return False
        return True
if __name__ == "__main__":
    grid_size = 3
    cleaner = VacuumCleaner(grid_size)
    cleaner.clean()
```

output:

```
  Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 23:03:10) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:\Users\haritha\OneDrive\Documents\New folder\vaccum cleaner prblm.py
Initial Grid:
[1, 0, 0]
[0, 1, 1]
[0, 1, 1]
Current Position: (1, 2)
Cell is dirty. Cleaning...

Updated Grid:
[1, 0, 0]
[0, 1, 0]
[0, 1, 1]
Current Position: (1, 1)
Cell is dirty. Cleaning...
```

Updated Grid:

[1, 0, 0]

[0, 0, 0]

[0, 1, 1]

Current Position: (0, 1)

Cell is already clean.

Updated Grid:

[1, 0, 0]

[0, 0, 0]

[0, 1, 1]

Current Position: (0, 2)

Cell is already clean.

Updated Grid:

[1, 0, 0]

[0, 0, 0]

[0, 1, 1]

Current Position: (0, 2)

Cell is already clean.

Updated Grid:

[1, 0, 0]

[0, 0, 0]

[0, 1, 1]

Current Position: (0, 2)

Cell is already clean.

Updated Grid:

[1, 0, 0]

[0, 0, 0]

[0, 1, 1]

Current Position: (0, 2)

Cell is already clean.

Updated Grid:

[1, 0, 0]

[0, 0, 0]

[0, 1, 1]

Current Position: (0, 1)

Cell is already clean.

Updated Grid:

[1, 0, 0]

[0, 0, 0]

[0, 1, 1]

Current Position: (1, 1)

Cell is already clean.

Updated Grid:

[1, 0, 0]

[0, 0, 0]

[0, 1, 1]

Current Position: (0, 1)

Cell is already clean.

Updated Grid:

[1, 0, 0]

[0, 0, 0]

[0, 1, 1]

Current Position: (0, 0)

Cell is dirty. Cleaning...

Updated Grid:

[0, 0, 0]

[0, 0, 0]

[0, 1, 1]

Current Position: (1, 0)

Cell is already clean.

Updated Grid:

[0, 0, 0]

[0, 0, 0]

[0, 1, 1]

Current Position: (1, 0)

Cell is already clean.

Updated Grid:

[0, 0, 0]

[0, 0, 0]

[0, 1, 1]

Current Position: (2, 0)

Cell is already clean.

Updated Grid:

[0, 0, 0]

[0, 0, 0]

[0, 1, 1]

Current Position: (2, 1)

Cell is dirty. Cleaning...

Updated Grid:

[0, 0, 0]

[0, 0, 0]

[0, 0, 1]

Current Position: (2, 0)

Cell is already clean.

Updated Grid:

[0, 0, 0]

[0, 0, 0]

[0, 0, 1]

Current Position: (1, 0)

Cell is already clean.

Updated Grid:

[0, 0, 0]

[0, 0, 0]

[0, 0, 1]

Current Position: (1, 1)

Cell is already clean.

Updated Grid:

[0, 0, 0]

[0, 0, 0]

[0, 0, 1]

Current Position: (1, 2)

Cell is already clean.

Updated Grid:

[0, 0, 0]

[0, 0, 0]

[0, 0, 1]

Current Position: (1, 2)

Cell is already clean.

Updated Grid:

[0, 0, 0]

[0, 0, 0]

[0, 0, 1]

Current Position: (2, 2)

Cell is dirty. Cleaning...

Updated Grid:

[0, 0, 0]

[0, 0, 0]

[0, 0, 0]

All cells are clean. Cleaning process completed.

>>>

16. missionaries cannibals

```
def valid_state(m_left, c_left, m_right, c_right):
    if m_left < 0 or c_left < 0 or m_right < 0 or c_right < 0:
        return False
    if m_left > 0 and m_left < c_left:
        return False
    if m_right > 0 and m_right < c_right:
        return False
    return True
def dfs(m_left, c_left, m_right, c_right, path, visited):
    if not valid_state(m_left, c_left, m_right, c_right):
        return False
    if (m_left, c_left, m_right, c_right) in visited:
        return False
```

```
    visited.add((m_left, c_left, m_right, c_right))
    path.append((m_left, c_left, m_right, c_right))
    if m_left == 0 and c_left == 0:
        return True
    moves = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]
    for move in moves:
        next_m_left = m_left - move[0]
        next_c_left = c_left - move[1]
        next_m_right = m_right + move[0]
        next_c_right = c_right + move[1]
        if dfs(next_m_left, next_c_left, next_m_right, next_c_right, path, visited):
            return True
    path.pop()
    return False
def print_solution(path):
    print("Missionaries and Cannibals Problem Solution:")
    for state in path:
        print(state)
def solve_missionaries_cannibals():
    m_left, c_left, m_right, c_right = 3, 3, 0, 0
    path = []
    visited = set()
    dfs(m_left, c_left, m_right, c_right, path, visited)
    print_solution(path)
if __name__ == "__main__":
    solve_missionaries_cannibals()
```

output:

Missionaries and Cannibals Problem Solution:

```
(3, 3, 0, 0)

(3, 2, 0, 1)

(2, 2, 1, 1)

(0, 2, 3, 1)

(0, 1, 3, 2)

(0, 0, 3, 3)

>>>
```