**LAB 1: Write a program to solve the Water Jug Problem using Breadth Frist Search (BFS).**

from collections import deque

def water\_jug\_problem(capacity\_x, capacity\_y, target):

visited\_states = set()

initial\_state = (0, 0)

queue = deque([(initial\_state, [])])

while queue:

state, actions = queue.popleft()

x, y = state

if x == target or y == target:

print("Found a solution:", actions)

return

visited\_states.add(state)

# Fill jug X

if x < capacity\_x:

new\_state = (capacity\_x, y)

if new\_state not in visited\_states:

queue.append((new\_state, actions + ["Fill X"]))

# Fill jug Y

if y < capacity\_y:

new\_state = (x, capacity\_y)

if new\_state not in visited\_states:

queue.append((new\_state, actions + ["Fill Y"]))

# Empty jug X

if x > 0:

new\_state = (0, y)

if new\_state not in visited\_states:

queue.append((new\_state, actions + ["Empty X"]))

# Empty jug Y

if y > 0:

new\_state = (x, 0)

if new\_state not in visited\_states:

queue.append((new\_state, actions + ["Empty Y"]))

# Pour from X to Y

if x > 0 and y < capacity\_y:

amount = min(x, capacity\_y - y)

new\_state = (x - amount, y + amount)

if new\_state not in visited\_states:

queue.append((new\_state, actions + ["Pour X to Y"]))

# Pour from Y to X

if y > 0 and x < capacity\_x:

amount = min(y, capacity\_x - x)

new\_state = (x + amount, y - amount)

if new\_state not in visited\_states:

queue.append((new\_state, actions + ["Pour Y to X"]))

print("No solution found.")

# Example usage

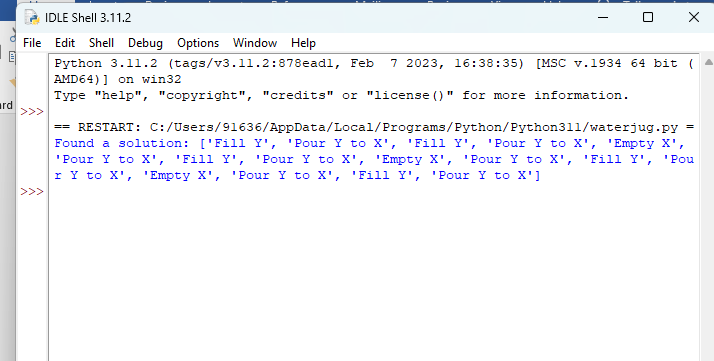
capacity\_x = 14

capacity\_y = 13

target = 9

water\_jug\_problem(capacity\_x, capacity\_y, target)

output :



**Lab 3: Write a program to solve the 4 – Queens Problem.**

**4 – queen problem**

def is\_safe(board, row, col):

# Check the left side of the current row

for i in range(col):

if board[row][i] == 1:

return False

# Check upper-left diagonal

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

# Check lower-left diagonal

for i, j in zip(range(row, len(board), 1), range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solve\_n\_queens\_util(board, col):

if col == len(board):

return True

for i in range(len(board)):

if is\_safe(board, i, col):

board[i][col] = 1

if solve\_n\_queens\_util(board, col + 1):

return True

board[i][col] = 0

return False

def solve\_n\_queens(n):

board = [[0 for \_ in range(n)] for \_ in range(n)]

if not solve\_n\_queens\_util(board, 0):

print("No solution exists.")

return

print\_board(board)

def print\_board(board):

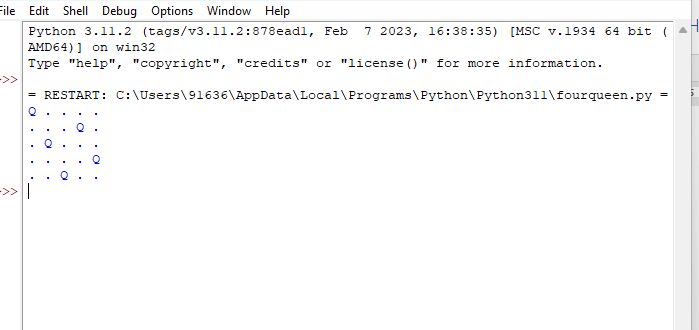
for row in board:

print(" ".join("Q" if cell == 1 else "." for cell in row))

# Example usage for 4-Queens problem

solve\_n\_queens(4)

**output :**



**Lab 4: minimax search for 2 players**

def print\_board(board):

for row in board:

print(" | ".join(row))

print("-" \* 9)

def is\_winner(board, player):

for row in board:

if all(cell == player for cell in row):

return True

for col in range(3):

if all(board[row][col] == player for row in range(3)):

return True

if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in range(3)):

return True

return False

def is\_full(board):

return all(cell != ' ' for row in board for cell in row)

def minimax(board, depth, is\_maximizing):

if is\_winner(board, 'X'):

return -1

if is\_winner(board, 'O'):

return 1

if is\_full(board):

return 0

if is\_maximizing:

max\_eval = -float('inf')

for i in range(3):

for j in range(3):

if board[i][j] == ' ':

board[i][j] = 'O'

eval = minimax(board, depth + 1, False)

board[i][j] = ' '

max\_eval = max(max\_eval, eval)

return max\_eval

else:

min\_eval = float('inf')

for i in range(3):

for j in range(3):

if board[i][j] == ' ':

board[i][j] = 'X'

eval = minimax(board, depth + 1, True)

board[i][j] = ' '

min\_eval = min(min\_eval, eval)

return min\_eval

def find\_best\_move(board):

best\_move = None

best\_eval = -float('inf')

for i in range(3):

for j in range(3):

if board[i][j] == ' ':

board[i][j] = 'O'

eval = minimax(board, 0, False)

board[i][j] = ' '

if eval > best\_eval:

best\_eval = eval

best\_move = (i, j)

return best\_move

def play\_game():

board = [[' ' for \_ in range(3)] for \_ in range(3)

]

print\_board(board)

for \_ in range(9):

if \_ % 2 == 0:

x, y = map(int, input("Enter your move (row and column): ").split())

if board[x][y] != ' ':

print("Invalid move. Try again.")

\_ -= 1

continue

board[x][y] = 'X'

else:

x, y = find\_best\_move(board)

board[x][y] = 'O'

print\_board(board)

if is\_winner(board, 'X'):

print("Player X wins!")

break

if is\_winner(board, 'O'):

print("Player O wins!")

break

if is\_full(board):

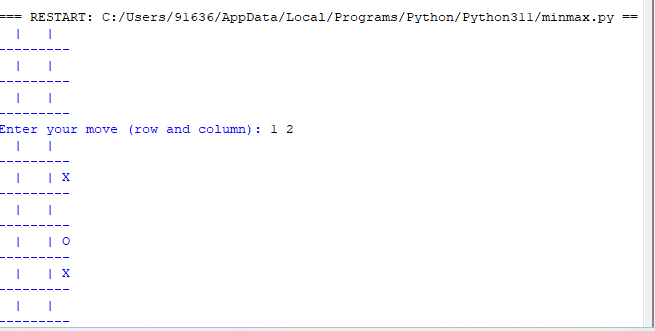
print("It's a tie!")

break

if \_\_name\_\_ == "\_\_main\_\_":

play\_game()

**output:**



**lab exercise 2: write a program in python to find out the optimum path from source to destination using a\* search technique**

import heapq

def astar(graph, start, goal):

open\_list = [(0, start)]

came\_from = {}

g\_score = {node: float('inf') for node in graph}

g\_score[start] = 0

f\_score = {node: float('inf') for node in graph}

f\_score[start] = heuristic(start, goal)

while open\_list:

\_, current = heapq.heappop(open\_list)

if current == goal:

return reconstruct\_path(came\_from, current)

for neighbor, cost in graph[current]:

tentative\_g\_score = g\_score[current] + cost

if tentative\_g\_score < g\_score[neighbor]:

came\_from[neighbor] = current

g\_score[neighbor] = tentative\_g\_score

f\_score[neighbor] = g\_score[neighbor] + heuristic(neighbor, goal)

heapq.heappush(open\_list, (f\_score[neighbor], neighbor))

return None # No path found

def heuristic(node, goal):

# Replace this with your specific heuristic function, e.g., Manhattan distance, Euclidean distance, etc.

return 0

def reconstruct\_path(came\_from, current):

path = [current]

while current in came\_from:

current = came\_from[current]

path.insert(0, current)

return path

# Example graph representation

graph = {

'A': [('B', 1), ('C', 3)],

'B': [('A', 1), ('D', 4), ('E', 2)],

'C': [('A', 3), ('F', 7)],

'D': [('B', 4), ('G', 5)],

'E': [('B', 2), ('H', 3)],

'F': [('C', 7), ('I', 2)],

'G': [('D', 5), ('H', 2)],

'H': [('E', 3), ('G', 2), ('I', 4)],

'I': [('F', 2), ('H', 4)]

}

start\_node = 'A'

goal\_node = 'I'

optimum\_path = astar(graph, start\_node, goal\_node)

if optimum\_path:

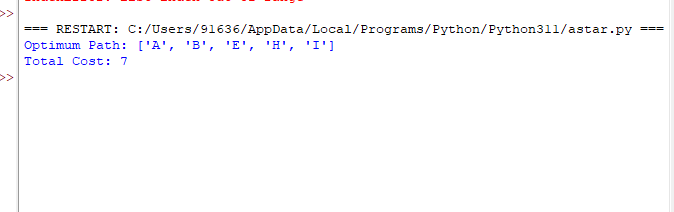
print("Optimum Path:", optimum\_path)

print("Total Cost:", sum(graph[node][0][1] for node in optimum\_path[:-1]))

else:

print("No path found.")

**output:**

****