Netaji Subhas University of Technology

PRACTICAL FILE

INITC16 – ARTIFICIAL INTELLIGENCE

Submitted to: Ms. Meena Jha

Name: Chirag Agarwal
Roll No.: 2021UIN3338
Branch: ITNS

INDEX

Practical – 1

AIM

Implementation of Tic Tac Toe Game

CODE

```
from os import system
from time import sleep
import random

def clear_screen():
    system("cls")

def print_board(board):
    print("\n".join([
        "╔══════╦══════╦══════╗",
        "║  {}  ║  {}  ║  {}  ║".format(*board[0:3]),
        "╠══════╬══════╬══════╣",
        "║  {}  ║  {}  ║  {}  ║".format(*board[3:6]),
        "╠══════╬══════╬══════╣",
        "║  {}  ║  {}  ║  {}  ║".format(*board[6:9]),
        "╚══════╩══════╩══════╝",
        "\n"
    ]))

def check_win(board, mark):
    possible_wins = [[0, 1, 2], [3, 4, 5], [6, 7, 8],
                     [0, 3, 6], [1, 4, 7], [2, 5, 8],
                     [0, 4, 8], [2, 4, 6]]
    return any(all(board[i] == mark for i in win) for win in
possible_wins)

def check_draw(board):
    return "  " not in board

def make_human_move(board, player, mark):
    try:
        row, column = map(int, input(f"Enter the row and column to place
{mark}: ").split(","))
        position = 3 * (row - 1) + (column - 1)
        if 0 <= position < 9 and board[position] == "  ":
            board[position] = mark
            clear_screen()
            print_board(board)
        else:
            print("Invalid position!")
            make_human_move(board, player, mark)
    except (ValueError, KeyboardInterrupt):
        print("Invalid input! Please enter row and column separated by a
comma.")
        make_human_move(board, player, mark)

def find_winning_move(board, mark):
    for i in range(9):
        if board[i] == "  ":
            board[i] = mark
            if check_win(board, mark):
                board[i] = "  "
                return i
            board[i] = "  "
    return None

def make_computer_move(board, mark):
    winning_move = find_winning_move(board, mark)
    if winning_move is not None:
        board[winning_move] = mark
    else:
        opponent_mark = "X " if mark == "O " else "O "
        blocking_move = find_winning_move(board, opponent_mark)
        if blocking_move is not None:
            board[blocking_move] = mark
        else:
            available_positions = [i for i in range(9) if board[i] == "
"]
            random_move = random.choice(available_positions)
            board[random_move] = mark
    clear_screen()
    print_board(board)

def main():
    try:
        system("cls")
        print("TIC TAC TOE GAME!")
        print("Player 1 will play with X and computer will play with O")
        sleep(1)
        print("The positions are numbered as shown below:")
        print("\n".join([
            "╔═════╦═════╦═════╗",
            "║ 2,1 ║ 2,2 ║ 2,3 ║",
            "╠═════╬═════╬═════╣",
            "║ 1,1 ║ 1,2 ║ 1,3 ║",
            "╠═════╬═════╬═════╣",
            "║ 3,1 ║ 3,2 ║ 3,3 ║",
            "╚═════╩═════╩═════╝",
            "\n"
        ]))
        sleep(1)
        print("LETS START!!")
        sleep(3)
        system("cls")
        player1 = input("Enter the name of Player 1: ")
        player2 = "Computer"
        sleep(1)
        player = random.choice([player1, player2])
        board = ["  "] * 9
        print_board(board)

        while True:
            print('\n' + player + "'s turn")
            mark = "X " if player == player1 else "O "

            if player == player1:
                make_human_move(board, player, mark)
            else:
                make_computer_move(board, mark)
            if check_win(board, mark):
                print(player, "wins!")
                print()
                return
            if check_draw(board):
                print("It's a draw!")
                print()
                return
            player = player2 if player == player1 else player1
    except KeyboardInterrupt:
        print("\n\nGame interrupted!")

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

OUTPUT

```
PS C:\Users\chira\OneDrive\Desktop\AI practical file> python -u
"c:\Users\chira\OneDrive\Desktop\AI practical file\TicTacToe.py"

TIC TAC TOE GAME!
Player 1 will play with X and computer will play with O
The positions are numbered as shown below:
╔═════╦═════╦═════╗
║ 2,1 ║ 2,2 ║ 2,3 ║
╠═════╬═════╬═════╣
║ 1,1 ║ 1,2 ║ 1,3 ║
╠═════╬═════╬═════╣
║ 3,1 ║ 3,2 ║ 3,3 ║
╚═════╩═════╩═════╝


LETS START!!
Enter the name of Player 1: chirag
╔════╦════╦════╗
║    ║    ║    ║
╠════╬════╬════╣
║    ║    ║    ║
╠════╬════╬════╣
║    ║    ║    ║
╚════╩════╩════╝


chirag's turn
Enter the row and column to place X : 1,1
╔════╦════╦════╗
║ X  ║    ║    ║
╠════╬════╬════╣
║    ║    ║    ║
╠════╬════╬════╣
║    ║    ║    ║
╚════╩════╩════╝


Computer's turn
╔════╦════╦════╗
║ X  ║    ║ O  ║
╠════╬════╬════╣
║    ║    ║    ║
╠════╬════╬════╣
║    ║    ║    ║
╚════╩════╩════╝
```

```
chirag's turn
Enter the row and column to place X : 3,3


╔═══╦═══╦═══╗
║ X ║   ║ O ║
╠═══╬═══╬═══╣
║   ║   ║   ║
╠═══╬═══╬═══╣
║   ║   ║ X ║
╚═══╩═══╩═══╝


Computer's turn
╔═══╦═══╦═══╗
║ X ║   ║ O ║
╠═══╬═══╬═══╣
║   ║ O ║   ║
╠═══╬═══╬═══╣
║   ║   ║ X ║
╚═══╩═══╩═══╝


chirag's turn
Enter the row and column to place X : 3,1
╔═══╦═══╦═══╗
║ X ║   ║ O ║
╠═══╬═══╬═══╣
║   ║ O ║   ║
╠═══╬═══╬═══╣
║ X ║   ║ X ║
╚═══╩═══╩═══╝


Computer's turn
╔═══╦═══╦═══╗
║ X ║   ║ O ║
╠═══╬═══╬═══╣
║ O ║ O ║   ║
╠═══╬═══╬═══╣
║ X ║   ║ X ║
╚═══╩═══╩═══╝


chirag's turn
Enter the row and column to place X : 3,2
╔═══╦═══╦═══╗
║ X ║   ║ O ║
╠═══╬═══╬═══╣
║ O ║ O ║   ║
╠═══╬═══╬═══╣
║ X ║ X ║ X ║
╚═══╩═══╩═══╝

chirag wins!
```

Practical – 2

AIM

Implementation of Tile Slide Puzzle

CODE

```
import time
import copy
from heapq import heappush, heappop

n = 3

rows = [ 1, 0, -1, 0 ]
cols = [ 0, -1, 0, 1 ]

class priorityQueue:
    def __init__(self):
        self.heap = []

    def push(self, key):
        heappush(self.heap, key)
    def pop(self):
        return heappop(self.heap)

    def empty(self):
        if not self.heap:
            return True
        else:
            return False

class nodes:
    def __init__(self, parent, mats, empty_tile_posi,
                costs, levels):
        self.parent = parent
        self.mats = mats
        self.empty_tile_posi = empty_tile_posi
        self.costs = costs
        self.levels = levels

    def __lt__(self, nxt):
        return self.costs < nxt.costs

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:

    new_mats = copy.deepcopy(mats)

    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]

    costs = calculateCosts(new_mats, final)

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

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

        print()


def isSafe(x, y):
    return x >= 0 and x < n and y >= 0 and y < n

def printPath(root):
    if root == None:
        return
    printPath(root.parent)
    printMatrix(root.mats)
    print()


def solve(initial, empty_tile_posi, final):
    pq = priorityQueue()
    visited = set()

    costs = calculateCosts(initial, final)
    root = nodes(None, initial,
                empty_tile_posi, costs, 0)

    pq.push(root)
    start_time = time.time()

    while not pq.empty():
        end_time = time.time()
        if ((end_time - start_time) > 2):
            print("The 8 puzzle is unsolvable ! \n")
            break
        minimum = pq.pop()
        if minimum.costs == 0:
            print("Solution found ! \n")
            printPath(minimum)
            steps_to_goal = minimum.levels
            print('Steps to reach goal state:', steps_to_goal)
            return

        for i in range(n+1):
            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]):
                child = newNodes(minimum.mats,
                                minimum.empty_tile_posi,
                                new_tile_posi,
                                minimum.levels + 1,
                                minimum, final,)
                if str(child.mats) not in visited: # check if node has
already been visited
                    pq.push(child)
                    visited.add(str(child.mats))

if __name__ == "__main__":
    initial = []
    final = []
    print('Enter initial state:')
    for i in range(n):
        row = list(map(int, input().split()))
        initial.append(row)
    print('Enter goal state:')
    for i in range(n):
        row = list(map(int, input().split()))
        final.append(row)

    #find position of empty tile
    for i in range(n):
        for j in range(n):
            if initial[i][j] == 0:
                empty_tile_posi = [i, j]
                break

    print('\n')
    solve(initial, empty_tile_posi, final)
```

OUTPUT

```
PS C:\Users\chira\OneDrive\Desktop\AI practical file> python -u
"c:\Users\chira\OneDrive\Desktop\AI practical file\8Puzzle.py"

Enter initial state:
1 2 3
5 6 0
7 8 4
Enter goal state:
1 2 3
0 8 6
5 7 4


Solution found !

1  2  3
5  6  0
7  8  4

1  2  3
5  0  6
7  8  4

1  2  3
5  8  6
7  0  4

1  2  3
5  8  6
0  7  4

1  2  3
0  8  6
5  7  4

Steps to reach goal state: 4
```

Practical – 3

AIM
Implementation of Water Jug Problem

CODE

```
from collections import deque

def solve(tar_a, tar_b, final_target):
    if final_target == 0:
        return

    q = deque([((0, 0), [(0, 0)], "")])
    vis = [[False] * (tar_b + 1) for _ in range(tar_a + 1)]
    vis[0][0] = True

    while q:
        (x_jug, y_jug), path, path_by_st = q.popleft()

        if x_jug == final_target or y_jug == final_target:
            print("Path:", path)
            print("Steps to reach:", path_by_st)
            print("FOUND")
            continue

        # option1: Completely fill A
        if x_jug < tar_a and not vis[tar_a][y_jug]:
            n_path = path + [(tar_a, y_jug)]
            n_path_by_st = path_by_st + " Fill A, "
            q.append(((tar_a, y_jug), n_path, n_path_by_st))
            vis[tar_a][y_jug] = True

        # option2: Completely fill B
        if y_jug < tar_b and not vis[x_jug][tar_b]:
            n_path = path + [(x_jug, tar_b)]
            n_path_by_st = path_by_st + " Fill B, "
            q.append(((x_jug, tar_b), n_path, n_path_by_st))
            vis[x_jug][tar_b] = True

        # option3: Completely Empty A
        if x_jug > 0 and not vis[0][y_jug]:
            n_path = path + [(0, y_jug)]
            n_path_by_st = path_by_st + " Empty A, "
            q.append(((0, y_jug), n_path, n_path_by_st))
            vis[0][y_jug] = True

        # option4: Completely Empty B
        if y_jug > 0 and not vis[x_jug][0]:
            n_path = path + [(x_jug, 0)]
            n_path_by_st = path_by_st + " Empty B, "
            q.append(((x_jug, 0), n_path, n_path_by_st))
            vis[x_jug][0] = True

        # option 5: Pour from A to B
        if x_jug > 0 and y_jug < tar_b:
            pour_amount = min(x_jug, tar_b - y_jug)
            n_path = path + [(x_jug - pour_amount, y_jug + pour_amount)]
            n_path_by_st = path_by_st + " Move A to B, "
            if not vis[x_jug - pour_amount][y_jug + pour_amount]:
                q.append(((x_jug - pour_amount, y_jug + pour_amount),
n_path, n_path_by_st))
                vis[x_jug - pour_amount][y_jug + pour_amount] = True

        # option 6: Pour from B to A
        if y_jug > 0 and x_jug < tar_a:
            pour_amount = min(y_jug, tar_a - x_jug)
            n_path = path + [(x_jug + pour_amount, y_jug - pour_amount)]
            n_path_by_st = path_by_st + " Move B to A, "
            if not vis[x_jug + pour_amount][y_jug - pour_amount]:
                q.append(((x_jug + pour_amount, y_jug - pour_amount),
n_path, n_path_by_st))
                vis[x_jug + pour_amount][y_jug - pour_amount] = True

if __name__ == "__main__":
    solve(4, 3, 2)
```

OUTPUT

```
PS C:\Users\chira\OneDrive\Desktop\AI practical file> python -u
"c:\Users\chira\OneDrive\Desktop\AI practical file\WaterJug.py"

Path: [(0, 0), (0, 3), (3, 0), (3, 3), (4, 2)]

Steps to reach:  Fill B,  Move B to A,  Fill B,  Move B to A,

FOUND

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

Steps to reach:  Fill A,  Move A to B,  Empty B,  Move A to B,  Fill A,
Move A to B,

FOUND
```

Practical – 4

AIM

Implementation of Generate and Test on Rat in a Maze problem.

CODE

```
def solve_maze(maze):
    # Get the dimensions of the maze
    rows = len(maze)
    cols = len(maze[0])

    # Helper function to check if a position is valid and not visited
    def is_valid(row, col, visited):
        return 0 <= row < rows and 0 <= col < cols and maze[row][col] ==
1 and (row, col) not in visited

    # Helper function to recursively generate and test paths
    def generate_and_test(row, col, path, visited):
        # Base case: reached the bottom right corner
        if row == rows - 1 and col == cols - 1:
            return path

        # Mark the current position as visited
        visited.add((row, col))

        # Try moving down
        if is_valid(row + 1, col, visited):
            result = generate_and_test(row + 1, col, path + [(row + 1,
col)], visited)
            if result:
                return result

        # Try moving right
        if is_valid(row, col + 1, visited):
            result = generate_and_test(row, col + 1, path + [(row, col +
1)], visited)
            if result:
                return result

        # Try moving up
        if is_valid(row - 1, col, visited):
            result = generate_and_test(row - 1, col, path + [(row - 1,
col)], visited)
            if result:
                return result
        # Try moving left
        if is_valid(row, col - 1, visited):
            result = generate_and_test(row, col - 1, path + [(row, col -
1)], visited)
            if result:
                return result

        # If no valid moves, mark the current position as unvisited
        visited.remove((row, col))

    # Start the generate and test algorithm from the top left corner
    start_row = 0
    start_col = 0
    start_path = [(start_row, start_col)]
    start_visited = set()
    solution = generate_and_test(start_row, start_col, start_path,
start_visited)

    if solution:
        # Create a solution matrix with the same dimensions as the maze
        solution_matrix = [[0] * cols for _ in range(rows)]

        # Mark the path taken by the algorithm with 'X'
        for row, col in solution:
            solution_matrix[row][col] = 'X'

        return solution_matrix
    else:
        return "No solution"

maze = [
    [1, 0, 1, 1, 1],
    [1, 1, 1, 0, 1],
    [0, 0, 0, 0, 1],
    [1, 1, 1, 1, 1]
]

solution = solve_maze(maze)
if solution == "No solution":
    print("No solution found")
else:
    print("Path exists, here is the path: ")
    for row in solution:
        print(*row)
```

OUTPUT

```
PS C:\Users\chira\OneDrive\Desktop\AI practical file> python -u
"c:\Users\chira\OneDrive\Desktop\AI practical file\GenerateAndTest.py"

Path exists, here is the path:

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

Practical – 5

AIM

Implementation of Systematic Generate and Test on Rat in a Maze Problem

CODE

```
def printSolution(sol):
    for i in sol:
        for j in i:
            print(str(j) + " ", end="")
        print("")

def isSafe(maze, x, y):
    if x >= 0 and x < N and y >= 0 and y < N and maze[x][y] == 1:
        return True
    return False

def solveMaze(maze):
    sol = [ [ 0 for j in range(N) ] for i in range(N) ]
    if solveMazeUtil(maze, 0, 0, sol) == False:
        print("Solution doesn't exist")
        return False
    print("Path exists, here is the path: ")
    printSolution(sol)
    return True

def solveMazeUtil(maze, x, y, sol):
    if x == N - 1 and y == N - 1 and maze[x][y]== 1:
        sol[x][y] = 1
        return True
    if isSafe(maze, x, y) == True:
        sol[x][y] = 1
        if solveMazeUtil(maze, x + 1, y, sol) == True:
            return True
        if solveMazeUtil(maze, x, y + 1, sol) == True:
            return True
        sol[x][y] = 0
        return False

if __name__ == "__main__":
    maze = [ [1, 0, 0, 0],
             [1, 1, 0, 1],
             [0, 1, 0, 0],
             [1, 1, 1, 1] ]
    N = len(maze)
    solveMaze(maze)
```

OUTPUT

```
PS C:\Users\chira\OneDrive\Desktop\AI practical file> python -u
"c:\Users\chira\OneDrive\Desktop\AI practical file\RatinMaze.py"

Path exists, here is the path:

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

Practical – 6

AIM

Implementation of Simple Hill Climbing (Minimizing cost of Living)

CODE

```
elevations = [10, 15, 12, 20, 25, 18, 22, 28]
graph = {
    0: [1, 2],
    1: [3, 4],
    2: [5, 6],
    4: [7]
}
current = 0
destination = 7

while True:
    if current == destination:
        break
    previous = current
    for neighbor in graph.get(current, []):
        if elevations[neighbor - 1] > elevations[current]:
            print(f"Found a state with higher elevation, climbing to node
{neighbor} (Elevation: {elevations[neighbor - 1]})")
            current = neighbor - 1
            break
    if current == previous:
        print("No state found with a higher elevation than the current
state")
        break

print(f"Global Maximum Elevation: {max(elevations)}")
print(f"Elevation Reached in Simple Hill Climbing:
{elevations[current]}")
```

OUTPUT

```
PS C:\Users\chira\OneDrive\Desktop\AI practical file> python -u
"c:\Users\chira\OneDrive\Desktop\AI practical file\HillClimbing.py"

Found a state with higher elevation, climbing to node 2 (Elevation: 15)

Found a state with higher elevation, climbing to node 4 (Elevation: 20)

No state found with a higher elevation than the current state

Global Maximum Elevation: 28

Elevation Reached in Simple Hill Climbing: 20
```

Practical – 7

AIM

Implementation of steepest ascent hill climbing (minimize cost of travel)

CODE

```
elevations = [0, 3, 2, 4, 5, 3, 6, 9]
graph = {
    0: [1, 2],
    1: [3, 4],
    2: [5, 6],
    4: [7]
}

names = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
current = 0
destination = 7

while True:
    if current == destination:
        break
    temp = current
    maxi = -1
    max_node = -1
    print(f"\n\nStates Available at this state are: ", end="")
    for neighbor in graph.get(current, []):
        print(f"{names[neighbor]}({elevations[neighbor]}), ", end="")
        if elevations[neighbor] > elevations[current]:
            max_node = neighbor
            maxi = elevations[neighbor]
    if maxi == -1:
        print("\nNo state found which is better than the current state")
        print()
        break
    else:
        print(f"\nBest Move among Available is: 
{names[max_node]}({maxi}), hence moving to {names[max_node]}")
        current = max_node

print(f"Global Maximum Elevation: {max(elevations)}")
print(f"In Steepest Hill Climbing, Elevation Reached: {names[current]} at 
distance {elevations[current]}")
```

OUTPUT

```
PS C:\Users\chira\OneDrive\Desktop\AI practical file> python -u
"c:\Users\chira\OneDrive\Desktop\AI practical file\SteepestAscent.py"


States Available at this state are: B(3), C(2),

Best Move among Available is: C(2), hence moving to C


States Available at this state are: F(3), G(6),

Best Move among Available is: G(6), hence moving to G


States Available at this state are:

No state found which is better than the current state


Global Maximum Elevation: 9

In Steepest Hill Climbing, Elevation Reached: G at distance 6
```

Practical – 8

AIM

Implementation of Best First Search for finding the Shortest Path

CODE

```
import heapq

def show_open_close(open_list, close_list, node_names):
    print("Open List:", end=" ")
    for node in open_list:
        print(node_names[node], end=" ")
    print("\nClosed List:", end=" ")
    for node in close_list:
        print(node_names[node], end=" ")
    print()

def main():
    pq = []  # priority queue (heuristic, node)
    node_names = ['S', 'A', 'B', 'D', 'C']

    open_list = []
    close_list = []
    adj = [
        [(1, 2), (2, 4)],
        [(3, 10), (2, 2), (4, 8)],
        [(4, 3)],
        [],
        [(3, 6)]
    ]

    heur = [0, 5, 3, 0, 4]

    vis = [0] * 5
    dest = 3

    heapq.heappush(pq, (heur[0], 0))
    path = {0: []}

    while pq:
        heuristic, node = heapq.heappop(pq)
        vis[node] = 1
        print(f"\n{node_names[node]} > {heuristic}\n")
        show_open_close(open_list, close_list, node_names)
        if node == dest:
            print()
            print(f"D is the Destination, and the heuristic value is
{heuristic}")
```

```
            print("Final Path:", ' -> '.join(node_names[i] for i in
path[node] + [node]))
            break
        for neighbor, distance in adj[node]:
            push_heuristic = heur[neighbor] + heuristic
            if not vis[neighbor]:
                heapq.heappush(pq, (push_heuristic, neighbor))
                open_list.append(neighbor)
                path[neighbor] = path[node] + [node]
        close_list.append(node)
        if node in open_list:
            open_list.remove(node)

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

OUTPUT

```
PS C:\Users\chira\OneDrive\Desktop\AI practical file> python -u
"c:\Users\chira\OneDrive\Desktop\AI practical file\BestFirstSearch.py"


S > 0

Open List:

Closed List:


B > 3

Open List: A B

Closed List: S


A > 5

Open List: A C

Closed List: S B


D > 5

Open List: C D C

Closed List: S B A


D is the Destination, and the heuristic value is 5

Final Path: S -> A -> D
```

Practical – 9

AIM

Implementation of A* Algorithm for finding the Shortest Path

CODE

```
import heapq

def show_open_close(open_set, close_list, node_names):
    print("Open Set:", end=" ")
    for node in open_set:
        print(node_names[node], end=" ")
    print("\nClosed List:", end=" ")
    for node in close_list:
        print(node_names[node], end=" ")
    print()

def print_final_path(parent, start, dest, node_names):
    path = [dest]
    current = dest
    while current != start:
        current = parent[current]
        path.insert(0, current)
    print("Final Path:", ' -> '.join(node_names[i] for i in path))

def main():
    pq = []  # priority queue (f, node)
    node_names = ['S', 'A', 'B', 'D', 'C']

    open_set = set()
    close_list = []
    adj = [
        [(1, 1), (2, 4)],
        [(3, 8), (2, 2), (4, 5)],
        [(4, 3)],
        [],
        [(3, 6)]
    ]

    heur = [0, 6, 2, 0, 1]

    vis = [False] * 5
    dest = 4

    g_values = [float('inf')] * 5
    g_values[0] = 0

    parent = [-1] * 5
    heapq.heappush(pq, (heur[0], 0))
    open_set.add(0)

    while pq:
        f_value, node = heapq.heappop(pq)
        vis[node] = True
        open_set.discard(node)
        print(f"\n{node_names[node]} > f:{f_value} g:{g_values[node]}
h:{heur[node]}\n")
        show_open_close(open_set, close_list, node_names)
        if node == dest:
            print(f"D is the Destination, and the optimal distance is
{g_values[node]}\n")
            print_final_path(parent, 0, dest, node_names)
            break
        for neighbor, weight in adj[node]:
            tentative_g_value = g_values[node] + weight
            if not vis[neighbor] and tentative_g_value <
g_values[neighbor]:
                g_values[neighbor] = tentative_g_value
                heapq.heappush(pq, (g_values[neighbor] + heur[neighbor],
neighbor))
                open_set.add(neighbor)
                parent[neighbor] = node
        close_list.append(node)

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

OUTPUT

```
PS C:\Users\chira\OneDrive\Desktop\AI practical file> python -u
"c:\Users\chira\OneDrive\Desktop\AI practical file\AStar.py"

f->f(n) value   g->g(n) value  h->h(n) value

S > f:0 g:0 h:0

Open Set:

Closed List:


B > f:6 g:4 h:2


Open Set: A

Closed List: S


A > f:7 g:1 h:6
```

```

Open Set: C

Closed List: S B


C > f:7 g:6 h:1


Open Set: D

Closed List: S B A

D is the Destination, and the optimal distance is 6


Final Path: S -> A -> C

```

Practical – 10

AIM

Implementation of AO* Algorithm for finding the Shortest Path

CODE

```
# AO* algorithm
def Cost(H, condition, weight=1):
    cost = {}

    if 'AND' in condition:
        AND_nodes = condition['AND']
        Path_A = 'AND'.join(AND_nodes)
        PathA = sum([H[node] + weight for node in AND_nodes])
        cost[Path_A] = PathA

    if 'OR' in condition:
        OR_nodes = condition['OR']
        Path_B = 'OR'.join(OR_nodes)
        PathB = min([H[node] + weight for node in OR_nodes])
        cost[Path_B] = PathB
    return cost

def update_cost(H, Conditions, weight=1):
    Main_nodes = list(Conditions.keys())
    Main_nodes.reverse()
    least_cost = {}

    for key in Main_nodes:
        condition = Conditions[key]
        c = Cost(H, condition, weight)
        H[key] = min(c.values())
        least_cost[key] = c

    return least_cost

def shortest_path(Start, Updated_cost, H):
    Path = Start

    if Start not in Updated_cost.keys():
        Min_cost = min(Updated_cost[Start].values())
        key = list(Updated_cost[Start].keys())
        values = list(Updated_cost[Start].values())

        Index = values.index(Min_cost)
        Next = key[Index].split()
        if len(Next) == 1:
            Start = Next[0]
            Path += '<-' + shortest_path(Start, Updated_cost, H)
        else:
            Start = Next[0]
            Path += '[' + shortest_path(Start, Updated_cost, H) + '+'
            Start = Next[-1]
            Path += shortest_path(Start, Updated_cost, H) + ']'

    return Path

H = {
    'A': 3,
    'B': 7,
    'C': 1,
    'D': 6,
    'E': 8,
    'F': 9,
    'G': 4,
    'H': 0,
    'I': 0,
    'J': 0
}

Conditions = {
    'A': {'OR': ['B'], 'AND': ['C', 'D']},
    'B': {'OR': ['E', 'F']},
    'C': {'OR': ['G'], 'AND': ['H', 'I']},
    'D': {'OR': ['J']}
}

weight = 2

print('Updated Cost:')
Updated_cost = update_cost(H, Conditions, weight=2)
print(Updated_cost)

print('Shortest Path in', shortest_path('A', Updated_cost, H))
```

OUTPUT:

```
PS C:\Users\chira\OneDrive\Desktop\AI practical file> python -u
"c:\Users\chira\OneDrive\Desktop\AI practical file\AOstar.py"

Updated Cost:

{'D': {'J': 2}, 'C': {'HANDI': 4, 'G': 6}, 'B': {'EORF': 10}, 'A':
{'CANDD': 10, 'B': 12}}

Shortest Path in A
```

Practical – 11

AIM

Implementation of Sentiment Analysis using NLP

CODE

```
from textblob import TextBlob

def analyze_sentiment(text):
    # Create a TextBlob object
    blob = TextBlob(text)

    # Get the sentiment polarity (ranges from -1 to 1)
    sentiment_polarity = blob.sentiment.polarity

    # Classify the sentiment
    if sentiment_polarity > 0:
        sentiment = 'Positive'
    elif sentiment_polarity < 0:
        sentiment = 'Negative'
    else:
        sentiment = 'Neutral'

    return sentiment, sentiment_polarity

# Example usage
text_to_analyze1 = "Python is a great language!"
text_to_analyze2 = "Python is a terrible language!"
text_to_analyze3 = "Python is a programming language!"
sentiment_result1, polarity1 = analyze_sentiment(text_to_analyze1)
sentiment_result2, polarity2 = analyze_sentiment(text_to_analyze2)
sentiment_result3, polarity3 = analyze_sentiment(text_to_analyze3)

print(f"Text: {text_to_analyze1}")
print(f"Sentiment: {sentiment_result1}")
print(f"Polarity: {polarity1}")
print()
print(f"Text: {text_to_analyze2}")
print(f"Sentiment: {sentiment_result2}")
print(f"Polarity: {polarity2}")
print()
print(f"Text: {text_to_analyze3}")
print(f"Sentiment: {sentiment_result3}")
print(f"Polarity: {polarity3}")
```

OUTPUT:

```
PS C:\Users\chira\OneDrive\Desktop\AI practical file> python -u
"c:\Users\chira\OneDrive\Desktop\AI practical file\NLP.py"
Text: Python is a great language!
Sentiment: Positive
Polarity: 1.0

Text: Python is a terrible language!
Sentiment: Negative
Polarity: -1.0

Text: python is a programming language!
Sentiment: Neutral
Polarity: 0.0
```

Practical – 12

AIM

Implementation of Tic Tac Toe Game using minimax with α-β pruning

CODE

```
def display_board(board):
    print("╔═══╦═══╦═══╗")
    print(f'║ {board[0][0]} ║ {board[0][1]} ║ {board[0][2]} ║')
    print("╟───╫───╫───╢")
    print(f'║ {board[1][0]} ║ {board[1][1]} ║ {board[1][2]} ║')
    print("╟───╫───╫───╢")
    print(f'║ {board[2][0]} ║ {board[2][1]} ║ {board[2][2]} ║')
    print("╚═══╩═══╩═══╝")
    print()

def is_space_free(board, position):
    row, col = divmod(position - 1, 3)
    return board[row][col] == ' '

def insert_letter(board, letter, position):
    row, col = divmod(position - 1, 3)
    if is_space_free(board, position):
        board[row][col] = letter
        display_board(board)
        if check_draw(board):
            print("It's a Draw!")
            print()
            exit()
        if check_for_win(board, letter):
            if letter == 'X':
                print("Bot wins!")
            else:
                print("Player wins!")
            print()
            exit()
    else:
        print("Can't insert there!")
        print()

def check_for_win(board, letter):
    for row in board:
        if all(cell == letter for cell in row):
            return True

    for col in range(3):
        if all(board[row][col] == letter for row in range(3)):
            return True

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

    return False

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

def player_move(board):
    position = int(input("Please enter a position (1-9): "))
    insert_letter(board, player, position)

def comp_move(board):
    best_score = -800
    best_move = 0
    alpha = -800  # Initialize alpha and beta
    beta = 800
    for position in range(1, 10):
        if is_space_free(board, position):
            row, col = divmod(position - 1, 3)
            board[row][col] = bot
            score = minimax(board, 0, alpha, beta, False)
            board[row][col] = ' '
            if score > best_score:
                best_score = score
                best_move = position
            alpha = max(alpha, best_score)
    insert_letter(board, bot, best_move)

def minimax(board, depth, alpha, beta, is_maximizing):
    if check_for_win(board, bot):
        return 1
    if check_for_win(board, player):
        return -1
    if check_draw(board):
        return 0

    if is_maximizing:
        best_score = -800
        for position in range(1, 10):
            if is_space_free(board, position):
                row, col = divmod(position - 1, 3)
                board[row][col] = bot
                score = minimax(board, depth + 1, alpha, beta, False)
                board[row][col] = ' '
                best_score = max(score, best_score)
                alpha = max(alpha, best_score)
                if beta <= alpha:
                    break
        return best_score
    else:
        best_score = 800
        for position in range(1, 10):
            if is_space_free(board, position):
                row, col = divmod(position - 1, 3)
                board[row][col] = player
                score = minimax(board, depth + 1, alpha, beta, True)
                board[row][col] = ' '
                best_score = min(score, best_score)
                beta = min(beta, best_score)
                if beta <= alpha:
                    break
        return best_score

board = [[' ' for _ in range(3)] for _ in range(3)]
display_board(board)
print("Computer goes first! Good luck.")
print("Positions are as follows:")
print("╔═══╦═══╦═══╗")
print("║ 1 ║ 2 ║ 3 ║")
print("╠═══╬═══╬═══╣")
print("║ 4 ║ 5 ║ 6 ║")
print("╠═══╬═══╬═══╣")
print("║ 7 ║ 8 ║ 9 ║")
print("╚═══╩═══╩═══╝")
print("")

player = 'O'
bot = 'X'

while not check_for_win(board, player):
    comp_move(board)
    player_move(board)
```

OUTPUT

```
PS C:\Users\chira\OneDrive\Desktop\AI practical file> python -u
"c:\Users\chira\OneDrive\Desktop\AI practical file\TicTacToeMinimax.py"

╔═══╦═══╦═══╗
║   ║   ║   ║
╠═══╬═══╬═══╣
║   ║   ║   ║
╠═══╬═══╬═══╣
║   ║   ║   ║
╚═══╩═══╩═══╝


Computer goes first! Good luck.

Positions are as follows:
```

```
╔═══╦═══╦═══╗
║ 1 ║ 2 ║ 3 ║
╠═══╬═══╬═══╣
║ 4 ║ 5 ║ 6 ║
╠═══╬═══╬═══╣
║ 7 ║ 8 ║ 9 ║
╚═══╩═══╩═══╝

╔═══╦═══╦═══╗
║ X ║   ║   ║
╠═══╬═══╬═══╣
║   ║   ║   ║
╠═══╬═══╬═══╣
║   ║   ║   ║
╚═══╩═══╩═══╝

Please enter a position (1-9): 5
╔═══╦═══╦═══╗
║ X ║   ║   ║
╠═══╬═══╬═══╣
║   ║ O ║   ║
╠═══╬═══╬═══╣
║   ║   ║   ║
╚═══╩═══╩═══╝

╔═══╦═══╦═══╗
║ X ║ X ║   ║
╠═══╬═══╬═══╣
║   ║ O ║   ║
╠═══╬═══╬═══╣
║   ║   ║   ║
╚═══╩═══╩═══╝

Please enter a position (1-9): 3
╔═══╦═══╦═══╗
║ X ║ X ║ O ║
╠═══╬═══╬═══╣
║   ║ O ║   ║
╠═══╬═══╬═══╣
║   ║   ║   ║
╚═══╩═══╩═══╝

╔═══╦═══╦═══╗
║ X ║ X ║ O ║
╠═══╬═══╬═══╣
║   ║ O ║   ║
╠═══╬═══╬═══╣
║ X ║   ║   ║
╚═══╩═══╩═══╝
```

```
Please enter a position (1-9): 4

╔═══╦═══╦═══╗
║ X ║ X ║ O ║
╠═══╬═══╬═══╣
║ O ║ O ║   ║
╠═══╬═══╬═══╣
║ X ║   ║   ║
╚═══╩═══╩═══╝


╔═══╦═══╦═══╗
║ X ║ X ║ O ║
╠═══╬═══╬═══╣
║ O ║ O ║ X ║
╠═══╬═══╬═══╣
║ X ║   ║   ║
╚═══╩═══╩═══╝


Please enter a position (1-9): 9

╔═══╦═══╦═══╗
║ X ║ X ║ O ║
╠═══╬═══╬═══╣
║ O ║ O ║ X ║
╠═══╬═══╬═══╣
║ X ║   ║ O ║
╚═══╩═══╩═══╝

╔═══╦═══╦═══╗
║ X ║ X ║ O ║
╠═══╬═══╬═══╣
║ O ║ O ║ X ║
╠═══╬═══╬═══╣
║ X ║ X ║ O ║
╚═══╩═══╩═══╝


It's a Draw!
```