```python
#1. Implement the Informed Search algorithm for real-life problems.

def astaralgo(start_node,stop_node):
    open_set=set([start_node])
    closed_set=set()
    g={}
    g[start_node]=0
    parents={}
    parents[start_node]=start_node
    while open_set:
        n=None

        for v in open_set:
            if n is None or g[v]+heuristics(v)<g[n]+heuristics(n):
                n=v

        if n is None:
            print("Path Does Not Exist!")
            return

        if n==stop_node:
            path=[]
            while parents[n]!=n:
                path.append(n)
                n=parents[n]
            path.append(start_node)
            path.reverse()
            print("Path Found! ", path)
            return

        for (m,weight) in Graph_nodes[n]:

            if m  not in open_set and m not in closed_set:
                open_set.add(m)
                parents[m]=n
                g[m]=g[n]+weight
            else:

                if g[m]>g[n]+heuristics(n):
                    g[m]=g[n]+heuristics(n)
                    parents[m]=n

                    if m in closed_set:
                        closed_set.remove(n)
                        open_set.add(m)


        open_set.remove(n)
        closed_set.add(n)
```

```
    print("PAth Not Found! ")
    return None



def heuristics(n):
    heuristics_dist={
        'A': 11,
        'B': 6,
        'C': 99,
        'D': 1,
        'E': 7,
        'G': 0
    }

    return heuristics_dist[n]

Graph_nodes={
    'A':[('B',2),('E',3)],
    'B':[('C',1),('G',9)],
    'C':[],
    'E':[('D',6)],
    'D':[('G',1)],
    'G':[]
}

astaralgo('A','G')

#2. Design an algorithm using Breadth-First Search (BFS) to find the
shortest path from a start node
#to a goal node in a maze represented as a grid graph. The maze
contains obstacles (walls) and free
#cells. Implement BFS to ensure that the first found path is the
optimal one in terms of the number
#of steps.

from collections import deque

directions=[(0,1),(1,0),(0,-1),(-1,0)]


def is_validmove(maze,visited,row,col):

    return (0<=row<len(maze)) and (0<=col<len(maze[0])) and (maze[row]
[col]!=1) and (not visited[row][col])

def bfs_shortest_path(maze,start,goal):

    queue=deque([(start,[start])])
    visited=[]
```

```
    for i in range(len(maze)):
        row=[]
        for j in range(len(maze[0])):
            row.append(False)
        visited.append(row)

    visited[start[0]][start[1]]=True

    while queue:

        (r,c),path=queue.popleft()

        if (r,c)==goal:
            print("Shortest Path Found: ",path)
            print("Number of steps taken: ",len(path)-1)
            return path


        for dr,dc in directions:
            new_r=dr+r
            new_c=dc+c

            if is_validmove(maze,visited,new_r,new_c):
                visited[new_r][new_c]=True

                queue.append(((new_r,new_c),path+[(new_r,new_c)]))

    print("No path Found! ")
    return None


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


start=(0,0)
goal=(4,4)

bfs_shortest_path(maze,start,goal)
```

3. Implement a Depth-First Search (DFS) algorithm to traverse a tree or graph representing a game
map. The goal is to explore all possible paths from the starting node to the target location,
marking visited nodes to avoid cycles, and visualize the order of traversal.

```
def dfs(graph,start,goal,visited=None,path=None):

    if visited is None:
        visited=set()

    if path is None:
        path=[]

    visited.add(start)
    path.append(start)

    if start==goal:
        print("PAth is Found! ",path)
        return True

    for neighbour in graph[start]:

        if neighbour not in visited:

            if dfs(graph,neighbour,goal,visited,path):
                return True

    path.pop()
    return False

graph={
    'A':['B','C'],
    'B':['D','E'],
    'C':['F'],
    'D':[],
    'E':['F'],
    'F':[]
}

dfs(graph,'A','F')
```

4. Develop a pathfinding solution using the A* algorithm for a maze-based game environment. The
agent must find the most cost-efficient route from the start position to the goal, considering
movement costs and a suitable heuristic function (e.g., Manhattan distance) to guide the search
efficiently.

```
from heapq import heappop , heappush

def a_star_search(maze,start,goal):
```

```
    rows,cols=len(maze),len(maze[0])

    directions=[(0,1),(1,0),(0,-1),(-1,0)]

    def heuristics(a,b):
        return abs(a[0]-b[0])+abs(a[1]-b[1])

    open_list=[]
    # f ,g (r,c) path
    heappush(open_list,(0+heuristics(start,goal),0,start,[start]))

    visited=set()

    while open_list:

        f,g,current,path=heappop(open_list)

        (r,c)=current

        if current==goal:
            print("Path is Found! ",path)
            return True

        if current in visited:
            continue

        visited.add(current)

        for dr,dc in directions:

            new_r=dr+r
            new_c=dc+c

            if 0<=new_r<len(maze) and 0<=new_c<len(maze[0]) and
maze[new_r][new_c]==0:
                neighbor=(new_r,new_c)

                if neighbor not in visited:

                    new_g=g+1
                    new_f=new_g+heuristics(neighbor,goal)

                    heappush(open_list,(new_f,new_g,neighbor,path+
[(new_r,new_c)]))

    print("Path Not Found")
    return None


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

```
    [0, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [1, 1, 0, 1, 0],
    [0, 0, 0, 0, 0]
]

# Start and Goal positions
start = (0, 0)
goal = (4, 4)

# Run the A* algorithm
a_star_search(maze, start, goal)
```

5. Implementation of 8 puzzles game

```
from heapq import heappop , heappush

def show(board):
    for i in range(0,9,3):
        print(board[i], board[i+1], board[i+2])
    print()

def next_states(state):
    res=[]
    zero=state.index(0)
    x,y=divmod(zero,3)

    moves= [(-1, 0), (1, 0), (0, -1), (0, 1)]

    for dx,dy in moves:
        nx=x+dx
        ny=y+dy

        if 0<=nx<3 and 0<=ny<3:
            ni=nx*3+ny
            new=list(state)
            new[zero],new[ni]=new[ni],new[zero]
            res.append(tuple(new))

    return res

def h(curr,goal):
    dist=0

    for n in range(1,9):
        x1,y1=divmod(curr.index(n),3)
        x2,y2=divmod(goal.index(n),3)
        dist+=abs(x1-x2)+abs(y1-y2)
    return dist

def solve(start,goal):
```

```
    pq=[]

    heappush(pq,(h(start,goal),0,start,[start]))

    seen=set()

    while pq:
        f,g,current,path=heappop(pq)

        if current in seen:
            continue
        seen.add(current)

        if current==goal:
            return path

        for next in next_states(current):
            if next not in seen:
                g2=g+1
                f2=g2+h(next,goal)
                heappush(pq,(f2,g2,next,path+[next]))

    return None

print(" Simple 8-Puzzle Solver using A* Algorithm")
print("Use 0 for the empty space.\n")

start = tuple(map(int, input("Enter START state (9 numbers separated
by space): ").split()))
goal = tuple(map(int, input("Enter GOAL state (9 numbers separated by
space): ").split()))

print("\nSolving...\n")
path = solve(start, goal)

if path:
    print(f"Solved in {len(path) - 1} moves!\n")
    for step in path:
        show(step)
else:
    print("No solution found for the given puzzle.")


6. Implementation of Tic-Tac-Toe game

def check_winner(board, player):
    win_combination = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],
        [0, 3, 6], [1, 4, 7], [2, 5, 8],
        [0, 4, 8], [2, 4, 6]
    ]
```

```
    for combo in win_combination:
        if all(board[i] == player for i in combo):
            return True
    return False

def is_board_full(board):
    return all(space != " " for space in board)

def print_board(board):
    print("\n")
    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]} ")
    print("\n")

def tic_tac_toe():
    board = [" "] * 9
    current_player = "X"

    print("Welcome to Tic-Tac-Toe")
    print("Player 1 is 'X' and Player 2 is 'O'")
    print_board(board)

    while True:
        try:
            move = int(input(f"Player {current_player}, enter your
move (1-9): ")) - 1
        except ValueError:
            print("Invalid input! Please enter a number between 1 and
9.")
            continue

        if move < 0 or move > 8 or board[move] != " ":
            print("Invalid move! Try again.")
            continue

        board[move] = current_player
        print_board(board)

        if check_winner(board, current_player):
            print(f"Congratulations! Player {current_player} wins!")
            break

        if is_board_full(board):
            print("It's a draw!")
            break

        current_player = "O" if current_player == "X" else "X"
```

```
tic_tac_toe()
```

7. Implementation of Tower of Hanoi game.

```
moves=0

def tower_of_hanoi(n,source,auxiliary,destination):
    global moves
    if n==1:
        moves+=1
        print(f"Move Disk 1 from {source} to {destination}")
        return 1

    tower_of_hanoi(n-1,source,destination,auxiliary)

    moves+=1
    print(f"Move Disk {n} from {source} to {destination}")

    tower_of_hanoi(n-1,auxiliary,source,destination)


n=int(input("Enter the number of disks: "))

tower_of_hanoi(n,'A','B','C')
print("Total Moves Required is ",moves)
```

8. Implementation of Water jug problems.

```
from collections import deque


def water_jug_problem(jug1_capacity,jug2_capacity,target):
    queue = deque([(0, 0, [], "Start")]) # (jug1, jug2, path, action)
    visited=set([0,0])

    while queue:
        jug1,jug2,path,action=queue.popleft()
        path = path + [(action, (jug1, jug2))]

        if jug1==target or jug2==target:
            return path

        next_moves=[]

        next_moves.append((jug1_capacity, jug2, "Fill Jug1"))

        # Fill Jug2
        next_moves.append((jug1, jug2_capacity, "Fill Jug2"))

        # Empty Jug1
        next_moves.append((0, jug2, "Empty Jug1"))
```

```
        # Empty Jug2
        next_moves.append((jug1, 0, "Empty Jug2"))

        # Pour Jug1 → Jug2
        pour = min(jug1, jug2_capacity - jug2)
        next_moves.append((jug1 - pour, jug2 + pour, "Pour Jug1 →
Jug2"))

        # Pour Jug2 → Jug1
        pour = min(jug2, jug1_capacity - jug1)
        next_moves.append((jug1 + pour, jug2 - pour, "Pour Jug2 →
Jug1"))

        for next_jug1, next_jug2, act in next_moves:
            if (next_jug1, next_jug2) not in visited:
                visited.add((next_jug1, next_jug2))
                queue.append((next_jug1, next_jug2, path, act))

    return None

jug1_cap = int(input("Enter capacity of Jug 1: "))
jug2_cap = int(input("Enter capacity of Jug 2: "))
target_amount = int(input("Enter target amount: "))

solution_path=water_jug_problem(jug1_cap, jug2_cap, target_amount)

if solution_path:
    print(f"\n Solution to measure {target_amount} liters using jugs
of {jug1_cap} and {jug2_cap} liters:\n")
    for i, (action, (j1, j2)) in enumerate(solution_path):
        print(f"Step {i}: {action:20s} --> (Jug1 = {j1}, Jug2 =
{j2})")
    print(f"\nTotal steps: {len(solution_path) - 1}")
else:
    print(f"\n No solution found for target {target_amount}.")
```