1. **Design Tic-tac-toe game in artificial intelligence.**

Code –

```
import math
# Initialize the Tic-Tac-Toe board
board = [[' ' for _ in range(3)] for _ in range(3)]
# Function to print the board
def print_board(board):
    for row in board:
        print('|'.join(row))
        print("-" * 5)
# Check if the board is full
def is_board_full(board):
    for row in board:
        if ' ' in row:
            return False
    return True
# Check for a winner
def check_winner(board):
    # Check rows, columns, and diagonals for a win
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != ' ':
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != ' ':
            return board[0][i]
    if board[0][0] == board[1][1] == board[2][2] != ' ':
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != ' ':
        return board[0][2]
```

```
    return None
# Minimax function for AI
def minimax(board, depth, is_maximizing):
    # Base case: check for terminal states
    winner = check_winner(board)
    if winner == 'O':
        return 1
    elif winner == 'X':
        return -1
    elif is_board_full(board):
        return 0
    # Maximizing player (AI)
    if is_maximizing:
        best_score = -math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = 'O'
                    score = minimax(board, depth + 1, False)
                    board[i][j] = ' '
                    best_score = max(score, best_score)
        return best_score
    # Minimizing player (Human)
    else:
        best_score = math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = 'X'
```

```
                score = minimax(board, depth + 1, True)
                board[i][j] = ' '
                best_score = min(score, best_score)
        return best_score
# AI move function
def best_move(board):
    best_score = -math.inf
    move = (0, 0)
    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':
                board[i][j] = 'O'
                score = minimax(board, 0, False)
                board[i][j] = ' '
                if score > best_score:
                    best_score = score
                    move = (i, j)
    board[move[0]][move[1]] = 'O'
# Function for player's move
def player_move(board):
    while True:
        try:
            row, col = map(int, input("Enter row and column (1-3) for X (e.g., 1 1 for top-left): ").split())
            row, col = row - 1, col - 1
            if board[row][col] == ' ':
                board[row][col] = 'X'
                break
            else:
                print("Cell is already occupied. Choose another.")
```

```
        except (ValueError, IndexError):
            print("Invalid input. Enter numbers between 1 and 3.")
# Main game loop
def play_game():
    print("Welcome to Tic-Tac-Toe!")
    print_board(board)
    while True:
        # Player's turn
        player_move(board)
        print_board(board)
        if check_winner(board) == 'X':
            print("You win!")
            break
        elif is_board_full(board):
            print("It's a tie!")
            break
        # AI's turn
        print("AI is making a move...")
        best_move(board)
        print_board(board)
        if check_winner(board) == 'O':
            print("AI wins!")
            break
        elif is_board_full(board):
            print("It's a tie!")
            break
# Run the game
play_game()
```

Output –

```
Welcome to Tic-Tac-Toe!

 | |

-----

 | |

-----

 | |

-----

Enter row and column (1-3) for X (e.g., 1 1 for top-left): 1 1

X| |

-----

 | |

-----

 | |

-----

AI is making a move...

X| |

-----

 |O|

-----

 | |

-----

Enter row and column (1-3) for X (e.g., 1 1 for top-left): 2 2

Cell is already occupied. Choose another.

Enter row and column (1-3) for X (e.g., 1 1 for top-left): 2 1

X| |

-----

X|O|

-----
```

| |

AI is making a move...

X| |

X|O|

O| |

Enter row and column (1-3) for X (e.g., 1 1 for top-left): 1 2

X|X|

X|O|

O| |

AI is making a move...

X|X|O

X|O|

O| |

AI wins!

=== Code Execution Successful ===

2. **Implement BFS (Breadth First Search) and DFS (Depth First Search) algorithms with several graphs.**

Code –

from collections import deque, defaultdict

```
# Graph class using an adjacency list
class Graph:
    def _init_(self):
        self.graph = defaultdict(list)
    # Add an edge to the graph
    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)  # For undirected graphs; remove for directed graphs
        # Breadth-First Search (BFS)
    def bfs(self, start):
        visited = set()         # Set to keep track of visited nodes
        queue = deque([start])   # Initialize a queue with the start node
        bfs_order = []          # List to keep the BFS traversal order
        while queue:
            vertex = queue.popleft()
            if vertex not in visited:
                visited.add(vertex)
                bfs_order.append(vertex)
                # Enqueue all unvisited neighbors
                for neighbor in self.graph[vertex]:
                    if neighbor not in visited:
                        queue.append(neighbor)
                     return bfs_order
    # Depth-First Search (DFS) using stack
    def dfs(self, start):
        visited = set()         # Set to keep track of visited nodes
        stack = [start]         # Initialize a stack with the start node
        dfs_order = []          # List to keep the DFS traversal order
```

```
        while stack:
            vertex = stack.pop()
            if vertex not in visited:
                visited.add(vertex)
                dfs_order.append(vertex)
                # Push all unvisited neighbors onto the stack
                for neighbor in reversed(self.graph[vertex]):  # reverse for typical DFS order
                    if neighbor not in visited:
                        stack.append(neighbor)
        return dfs_order
    # Depth-First Search (DFS) using recursion
    def dfs_recursive(self, start, visited=None, dfs_order=None):
        if visited is None:
            visited = set()
        if dfs_order is None:
            dfs_order = []
        visited.add(start)
        dfs_order.append(start)
        for neighbor in self.graph[start]:
            if neighbor not in visited:
                self.dfs_recursive(neighbor, visited, dfs_order)
        return dfs_order
# Sample Graphs and Tests
if _name_ == "_main_":
    # Create a new graph
    graph = Graph()
    # Add edges to the graph
    edges = [
```

```
        (0, 1), (0, 2), (1, 2), (1, 3),
        (2, 4), (3, 4), (4, 5), (3, 6)
    ]
    for u, v in edges:
        graph.add_edge(u, v)
        # Perform BFS and DFS traversals
    start_node = 0
    print("Graph adjacency list representation:")
    for node, neighbors in graph.graph.items():
        print(f"{node}: {neighbors}")
        # BFS traversal
    print("\nBFS traversal starting from node 0:")
    print(graph.bfs(start_node))  # Output: BFS traversal order
    # DFS traversal (iterative)
    print("\nDFS traversal (iterative) starting from node 0:")
    print(graph.dfs(start_node))  # Output: DFS traversal order
    # DFS traversal (recursive)
    print("\nDFS traversal (recursive) starting from node 0:")
    print(graph.dfs_recursive(start_node))  # Output: DFS traversal order
```

Output –

```
Graph adjacency list representation:
0: [1, 2]
1: [0, 2, 3]
2: [0, 1, 4]
3: [1, 4, 6]
4: [2, 3, 5]
5: [4]
6: [3]
```

BFS traversal starting from node 0:

[0, 1, 2, 3, 4, 6, 5]

DFS traversal (iterative) starting from node 0:

[0, 1, 2, 4, 3, 6, 5]

DFS traversal (recursive) starting from node 0:

[0, 1, 2, 4, 3, 6, 5]

=== Code Execution Successful ===

3. Implement uniform cost search algorithm (Dijkstra algorithm).

Code –

```
import heapq
from collections import defaultdict
class Graph:
    def _init_(self):
        # Graph representation using an adjacency list
        self.graph = defaultdict(list)
    # Add an edge to the graph
    def add_edge(self, u, v, weight):
        self.graph[u].append((v, weight))
        self.graph[v].append((u, weight))  # For undirected graphs; remove for directed graphs
    # Uniform Cost Search (UCS) / Dijkstra's Algorithm
    def uniform_cost_search(self, start, goal):
        # Priority queue to store (cost, node) and start with the start node at cost 0
        priority_queue = [(0, start)]
        # Dictionary to track the minimum cost to reach each node
        costs = {start: 0}
        # Dictionary to store the path taken
        predecessors = {start: None}
        while priority_queue:
            # Pop the node with the lowest cost
```

```
            current_cost, current_node = heapq.heappop(priority_queue)

            # If we reach the goal, reconstruct the path and return
            if current_node == goal:
                path = []
                while current_node is not None:
                    path.append(current_node)
                    current_node = predecessors[current_node]
                return path[::-1], current_cost

            # Explore each neighbor of the current node
            for neighbor, weight in self.graph[current_node]:
                new_cost = current_cost + weight
                # If the new cost to reach neighbor is lower, update and push to the queue
                if neighbor not in costs or new_cost < costs[neighbor]:
                    costs[neighbor] = new_cost
                    predecessors[neighbor] = current_node
                    heapq.heappush(priority_queue, (new_cost, neighbor))

        # If the goal is unreachable, return an empty path and infinite cost
        return [], float('inf')

# Sample Graph and Test
if _name_ == "_main_":
    # Create a new graph
    graph = Graph()

    # Add edges to the graph with weights
    edges = [
        (0, 1, 2), (0, 2, 4), (1, 2, 1),
        (1, 3, 7), (2, 4, 3), (3, 4, 2),
        (4, 5, 5), (3, 5, 1)
    ]
    for u, v, weight in edges:
```

```
        graph.add_edge(u, v, weight)
    # Define start and goal nodes
    start_node = 0
    goal_node = 5
    # Perform Uniform Cost Search / Dijkstra's Algorithm
    path, cost = graph.uniform_cost_search(start_node, goal_node)
    # Output results
    print(f"Graph adjacency list representation with weights:")
    for node, neighbors in graph.graph.items():
        print(f"{node}: {neighbors}")
        print(f"\nUniform Cost Search from {start_node} to {goal_node}:")
    print(f"Path: {path}")
    print(f"Total Cost: {cost}")
```

Output –

```
Graph adjacency list representation with weights:
0: [(1, 2), (2, 4)]
1: [(0, 2), (2, 1), (3, 7)]
2: [(0, 4), (1, 1), (4, 3)]
3: [(1, 7), (4, 2), (5, 1)]
4: [(2, 3), (3, 2), (5, 5)]
5: [(4, 5), (3, 1)]
Uniform Cost Search from 0 to 5:
Path: [0, 1, 2, 4, 3, 5]
Total Cost: 9
=== Code Execution Successful ===
```

4. Implement the A* algorithm in of a graph with given heuristic.

Code –

```
import heapq
from collections import defaultdict
```

```
class Graph:
    def _init_(self):
        # Graph representation using an adjacency list
        self.graph = defaultdict(list)
    # Add an edge to the graph
    def add_edge(self, u, v, weight):
        self.graph[u].append((v, weight))
        self.graph[v].append((u, weight))  # For undirected graphs; remove for directed graphs
    # A* Search Algorithm
    def a_star(self, start, goal, heuristic):
        # Priority queue to store (f_cost, current_cost, node)
        priority_queue = [(0, 0, start)]
        # Dictionary to track the minimum cost to reach each node
        g_cost = {start: 0}
        # Dictionary to store the path taken
        predecessors = {start: None}
        while priority_queue:
            # Pop the node with the lowest f_cost
            _, current_cost, current_node = heapq.heappop(priority_queue)

            # If we reach the goal, reconstruct the path and return
            if current_node == goal:
                path = []
                while current_node is not None:
                    path.append(current_node)
                    current_node = predecessors[current_node]
                return path[::-1], g_cost[goal]
```

```
        # Explore each neighbor of the current node
        for neighbor, weight in self.graph[current_node]:
            new_cost = current_cost + weight
            # If the new cost to reach neighbor is lower, update and push to the queue
            if neighbor not in g_cost or new_cost < g_cost[neighbor]:
                g_cost[neighbor] = new_cost
                f_cost = new_cost + heuristic[neighbor]  # f(n) = g(n) + h(n)
                predecessors[neighbor] = current_node
                heapq.heappush(priority_queue, (f_cost, new_cost, neighbor))
    # If the goal is unreachable, return an empty path and infinite cost
    return [], float('inf')
# Sample Graph and Test
if _name_ == "_main_":
    # Create a new graph
    graph = Graph()
    # Add edges to the graph with weights
    edges = [
        (0, 1, 1), (0, 2, 4), (1, 2, 2),
        (1, 3, 5), (2, 3, 1), (2, 4, 7),
        (3, 4, 3), (3, 5, 8), (4, 5, 2)
    ]
    for u, v, weight in edges:
        graph.add_edge(u, v, weight)
    # Define start and goal nodes
    start_node = 0
    goal_node = 5
    # Define heuristic values for each node (for example purposes)
    heuristic = {
        0: 7,  # Estimated cost from node 0 to goal
```

```
        1: 6,  # Estimated cost from node 1 to goal
        2: 2,  # Estimated cost from node 2 to goal
        3: 1,  # Estimated cost from node 3 to goal
        4: 3,  # Estimated cost from node 4 to goal
        5: 0   # Goal node heuristic is always 0
    }
    # Perform A* Search
    path, cost = graph.a_star(start_node, goal_node, heuristic)
    # Output results
    print(f"Graph adjacency list representation with weights:")
    for node, neighbors in graph.graph.items():
        print(f"{node}: {neighbors}")
    print(f"\nA* Search from {start_node} to {goal_node}:")
    print(f"Path: {path}")
    print(f"Total Cost: {cost}")
```

Output –

```
Graph adjacency list representation with weights:
0: [(1, 1), (2, 4)]
1: [(0, 1), (2, 2), (3, 5)]
2: [(0, 4), (1, 2), (3, 1), (4, 7)]
3: [(1, 5), (2, 1), (4, 3), (5, 8)]
4: [(2, 7), (3, 3), (5, 2)]
5: [(3, 8), (4, 2)]
A* Search from 0 to 5:
Path: [0, 1, 2, 3, 4, 5]
Total Cost: 9
=== Code Execution Successful ===
```

5. Implement a search algorithm for solving the 8-puzzle problem with following assumptions: I. g(n) = least cost from source state to current state so far. II. Heuristics a. h1(n) = 0. b. h2(n) =

number of tiles displaced from their destined position. c. h3(n) = sum of Manhattan distance of each tiles from the goal position. d. h4(n) = Devise a heuristics such that h(n) > h*(n)

Code –

```
import heapq

class PuzzleState:

    def _init_(self, tiles, empty_tile_index, g, h):

        self.tiles = tiles

        self.empty_tile_index = empty_tile_index  # Index of the blank tile (0)

        self.g = g  # Cost from the start to this state

        self.h = h  # Heuristic cost to the goal

        self.f = g + h  # Total cost

    def _lt_(self, other):

        return self.f < other.f  # For priority queue

    def is_goal(self):

        """Check if the current state is the goal state."""

        return self.tiles == [1, 2, 3, 4, 5, 6, 7, 8, 0]

    def generate_neighbors(self):

        """Generate all valid neighbor states by moving the blank tile."""

        neighbors = []

        row, col = divmod(self.empty_tile_index, 3)

        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right

        for dr, dc in directions:

            new_row, new_col = row + dr, col + dc

            if 0 <= new_row < 3 and 0 <= new_col < 3:

                new_empty_index = new_row * 3 + new_col

                new_tiles = list(self.tiles)

                # Swap the empty tile with the adjacent tile
```

```
            new_tiles[self.empty_tile_index], new_tiles[new_empty_index] =
new_tiles[new_empty_index], new_tiles[self.empty_tile_index]
            neighbors.append((new_tiles, new_empty_index))
        return neighbors
def h1(state):
    """Heuristic 1: Always returns 0."""
    return 0
def h2(state):
    """Heuristic 2: Number of displaced tiles."""
    return sum(1 for i in range(9) if state[i] != 0 and state[i] != i + 1)
def h3(state):
    """Heuristic 3: Sum of Manhattan distances."""
    distance = 0
    goal_positions = {val: (i // 3, i % 3) for i, val in enumerate(range(1, 9))}
    for i in range(9):
        tile = state[i]
        if tile != 0:
            goal_x, goal_y = goal_positions[tile]
            current_x, current_y = divmod(i, 3)
            distance += abs(goal_x - current_x) + abs(goal_y - current_y)
    return distance
def h4(state):
    """Heuristic 4: Sum of Manhattan distance + an arbitrary constant (e.g., 5)."""
    return h3(state) + 5
def a_star(start_state, heuristic):
    """A* search algorithm."""
    empty_tile_index = start_state.index(0)
    initial_state = PuzzleState(start_state, empty_tile_index, 0, heuristic(start_state))
    open_set = []
```

```
    heapq.heappush(open_set, initial_state)
    closed_set = set()
    while open_set:
        current_state = heapq.heappop(open_set)
        if current_state.is_goal():
            return current_state
        closed_set.add(tuple(current_state.tiles))
        for neighbor_tiles, neighbor_empty_index in current_state.generate_neighbors():
            if tuple(neighbor_tiles) in closed_set:
                continue
            g_cost = current_state.g + 1  # All moves have a cost of 1
            h_cost = heuristic(neighbor_tiles)
            neighbor_state = PuzzleState(neighbor_tiles, neighbor_empty_index, g_cost, h_cost)
            # If neighbor is not in the open set or has a lower f cost, add it
            heapq.heappush(open_set, neighbor_state)
    return None  # No solution found
# Example usage
  start_state = [1, 2, 3, 4, 0, 5, 6, 7, 8]  # Example starting state
  print("Start State:")
  print(start_state)
  # A* Search using h2 (displaced tiles)
  solution = a_star(start_state, h2)
  if solution:
    print("Goal State Found:")
    print(solution.tiles)
  else:
    print("No solution found.")
  # A* Search using h3 (Manhattan distance)
  solution = a_star(start_state, h3)
```

```
if solution:
    print("Goal State Found:")
    print(solution.tiles)
else:
    print("No solution found.")
# A* Search using h4 (custom heuristic)
solution = a_star(start_state, h4)
if solution:
    print("Goal State Found:")
    print(solution.tiles)
else:
    print("No solution found.")
```

Output –

```
Start State:
[1, 2, 3, 4, 0, 5, 6, 7, 8]
Goal State Found:
[1, 2, 3, 4, 5, 6, 7, 8, 0]
Goal State Found:
[1, 2, 3, 4, 5, 6, 7, 8, 0]
Goal State Found:
[1, 2, 3, 4, 5, 6, 7, 8, 0]


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

6. Implement AO* algorithm to solve a game tree.

Code –

```
import math
class Node:
    def _init_(self, name, is_and_node=False, heuristic=math.inf):
        self.name = name                # Name of the node (e.g., 'A', 'B', 'C')
```

```
        self.is_and_node = is_and_node      # If True, this node is an AND node; otherwise, it's an OR node
        self.heuristic = heuristic         # Heuristic value for this node (initially high for non-leaf nodes)
        self.children = []                # List of child nodes (with path cost)
        self.optimal_child = None          # Optimal child to follow in case of an OR node
        self.solved = False               # True if this node is considered solved
    def add_child(self, child, cost=1):
        self.children.append((child, cost))
    def _repr_(self):
        return f"Node({self.name}, H={self.heuristic})"
def ao_star(node, path_cost=0):
    """
    Recursively applies the AO* algorithm to find the optimal solution path in an AND-OR tree.
    """
    if node.solved:  # If the node is already solved, return its heuristic
        return node.heuristic
    if not node.children:  # Leaf node; assume its heuristic is the end cost
        node.solved = True
        return node.heuristic
    # Recursive step for AND-OR nodes
    costs = []  # Store the costs of all children paths
    if node.is_and_node:
        total_cost = 0  # AND node: sum of all child costs
        for child, cost in node.children:
            child_cost = ao_star(child, path_cost + cost)
            total_cost += cost + child_cost
        node.heuristic = total_cost
        costs.append(total_cost)
    else:
```

```
        min_cost = math.inf
        for child, cost in node.children:
            child_cost = ao_star(child, path_cost + cost)
            total_cost = cost + child_cost
            if total_cost < min_cost:
                min_cost = total_cost
                node.optimal_child = child
        node.heuristic = min_cost
        costs.append(min_cost)
    # Backtracking to update the cost to reflect the current state
    if node.is_and_node:
        node.solved = all(child.solved for child, _ in node.children)
    else:
        node.solved = node.optimal_child.solved if node.optimal_child else False
    return node.heuristic
# Example: Constructing a sample AND-OR tree
# Define nodes
A = Node("A")              # Root OR node
B = Node("B", is_and_node=True)    # AND node
C = Node("C")              # OR node
D = Node("D")              # Leaf node with heuristic
E = Node("E")              # Leaf node with heuristic
F = Node("F", heuristic=2)   # Leaf node with heuristic
G = Node("G", heuristic=4)   # Leaf node with heuristic
H = Node("H", heuristic=1)   # Leaf node with heuristic
I = Node("I", heuristic=3)   # Leaf node with heuristic
# Construct tree by adding children and specifying path costs
A.add_child(B, cost=1)      # A -> B (cost 1)
A.add_child(C, cost=3)      # A -> C (cost 3)
```

```
B.add_child(D, cost=1)      # B (AND) -> D (cost 1)
B.add_child(E, cost=1)      # B (AND) -> E (cost 1)
C.add_child(F, cost=2)      # C -> F (cost 2)
C.add_child(G, cost=4)      # C -> G (cost 4)
D.add_child(H, cost=1)      # D -> H (cost 1)
E.add_child(I, cost=3)      # E -> I (cost 3)
# Run AO* algorithm on the root node A
print("Running AO* Algorithm on the AND-OR Tree...")
solution_cost = ao_star(A)
print(f"Optimal Solution Cost: {solution_cost}")
print(f"Root Node Heuristic after AO*: {A.heuristic}")
```

Output –

```
Running AO* Algorithm on the AND-OR Tree...
Optimal Solution Cost: 7
Root Node Heuristic after AO*: 7
=== Code Execution Successful ===
```

7. Implement water jug problem is described as: There are two jugs of capacity 4 litres and 3 litres with no marking. You have to measure out exactly 2 litres from a vat containing 20 litters and more water.

Code –

```
from collections import deque
def water_jug_bfs(target, jug1_capacity=4, jug2_capacity=3):
    # Initialize the queue with the starting state (0, 0) and an empty path
    queue = deque([((0, 0), [])])
    visited = set()  # To keep track of visited states
    # Run BFS
    while queue:
        (jug1, jug2), path = queue.popleft()
        # If we reach the target in either of the jugs, return the path
```

```
        if jug1 == target or jug2 == target:
            return path + [(jug1, jug2)]
        # Mark the state as visited
        if (jug1, jug2) in visited:
            continue
        visited.add((jug1, jug2))
        # List all possible operations and add resulting states to the queue
        # 1. Fill Jug1
        queue.append(((jug1_capacity, jug2), path + [(jug1, jug2)]))
        # 2. Fill Jug2
        queue.append(((jug1, jug2_capacity), path + [(jug1, jug2)]))
        # 3. Empty Jug1
        queue.append(((0, jug2), path + [(jug1, jug2)]))
        # 4. Empty Jug2
        queue.append(((jug1, 0), path + [(jug1, jug2)]))
        # 5. Pour water from Jug1 to Jug2 until Jug2 is full or Jug1 is empty
        pour_to_jug2 = min(jug1, jug2_capacity - jug2)
        queue.append(((jug1 - pour_to_jug2, jug2 + pour_to_jug2), path + [(jug1, jug2)]))
        # 6. Pour water from Jug2 to Jug1 until Jug1 is full or Jug2 is empty
        pour_to_jug1 = min(jug2, jug1_capacity - jug1)
        queue.append(((jug1 + pour_to_jug1, jug2 - pour_to_jug1), path + [(jug1, jug2)]))
    return None  # If there's no solution (shouldn't happen for this problem)
# Example Usage
target_amount = 2
solution_path = water_jug_bfs(target_amount)
# Display the solution path if found
if solution_path:
    print("Steps to measure exactly 2 liters:")
    for step in solution_path:
```

```
        print(f"Jug1: {step[0]} liters, Jug2: {step[1]} liters")
else:
    print("No solution found.")
```

Output –

```
Steps to measure exactly 2 liters:
Jug1: 0 liters, Jug2: 0 liters
Jug1: 0 liters, Jug2: 3 liters
Jug1: 3 liters, Jug2: 0 liters
Jug1: 3 liters, Jug2: 3 liters
Jug1: 4 liters, Jug2: 2 liters
=== Code Execution Successful ===
```

8. Implement wolf-goat-cabbage (WGC) problem is described in such way: A farmer has a wolf, a goat, and a cabbage with him and is on left bank of a river. He has a boat to ferry them across which can carry at most one of three with him. He must transport these to the right bank. But the problem is he dare not leave the wolf with goat or goat with cabbage. How does he do the transport?

Code –

```
from collections import deque
# Helper function to check if a state is valid
def is_valid_state(state):
    F, W, G, C = state
    # Check if the farmer is not leaving the wolf with the goat or the goat with the cabbage
    if (W == G != F) or (G == C != F):
        return False
    return True
# BFS to solve the problem
def solve_wgc_problem():
    # Initial state: all on the left bank
    initial_state = (0, 0, 0, 0)  # (Farmer, Wolf, Goat, Cabbage)
    goal_state = (1, 1, 1, 1)     # Goal state: all on the right bank
```

```
    # Queue for BFS: each element is (current_state, path_taken)
    queue = deque([(initial_state, [initial_state])])
    visited = set([initial_state])  # To keep track of visited states
    # BFS
    while queue:
        current_state, path = queue.popleft()
        # If we've reached the goal state, return the path
        if current_state == goal_state:
            return path
        F, W, G, C = current_state
        # Possible moves (0->1 for crossing right, 1->0 for crossing left)
        possible_moves = [
            (1 - F, W, G, C),        # Farmer crosses alone
            (1 - F, 1 - W, G, C) if F == W else None,  # Farmer takes the wolf
            (1 - F, W, 1 - G, C) if F == G else None,  # Farmer takes the goat
            (1 - F, W, G, 1 - C) if F == C else None   # Farmer takes the cabbage
        ]
        # Explore each possible move
        for new_state in possible_moves:
            if new_state and new_state not in visited and is_valid_state(new_state):
                visited.add(new_state)
                queue.append((new_state, path + [new_state]))
    return None  # If no solution is found
# Solve the problem and print the steps
solution = solve_wgc_problem()
if solution:
    print("Solution steps to transport the wolf, goat, and cabbage safely:")
    for step in solution:
        F, W, G, C = step
```

```
        print(f"Farmer: {'Right' if F else 'Left'}, Wolf: {'Right' if W else 'Left'}, "
              f"Goat: {'Right' if G else 'Left'}, Cabbage: {'Right' if C else 'Left'}")
else:
    print("No solution found.")
```

Output –

Solution steps to transport the wolf, goat, and cabbage safely:

Farmer: Left, Wolf: Left, Goat: Left, Cabbage: Left

Farmer: Right, Wolf: Left, Goat: Right, Cabbage: Left

Farmer: Left, Wolf: Left, Goat: Right, Cabbage: Left

Farmer: Right, Wolf: Right, Goat: Right, Cabbage: Left

Farmer: Left, Wolf: Right, Goat: Left, Cabbage: Left

Farmer: Right, Wolf: Right, Goat: Left, Cabbage: Right

Farmer: Left, Wolf: Right, Goat: Left, Cabbage: Right

Farmer: Right, Wolf: Right, Goat: Right, Cabbage: Right

=== Code Execution Successful ===

9. Implement Hill Climbing Search Algorithm for solving the 8-puzzle problem. Your start state can be anything and the goal state will be {123; 456; 78B}, where B is blank tile. Heuristics can be checked: i. h1(n)= Number of displaced titles. ii. h2(n)= Total Manhatton distance.

Code –

```
import random

# Define the goal state and possible moves

goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]

goal_positions = {val: (i // 3, i % 3) for i, val in enumerate(goal_state)}  # Position map for Manhattan distance

def display(state):
    # Display the puzzle state in a 3x3 grid format
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()

# Heuristic 1: Number of misplaced tiles
```

```python
def h1_displaced_tiles(state):
    return sum(1 for i in range(9) if state[i] != goal_state[i] and state[i] != 0)
# Heuristic 2: Total Manhattan distance
def h2_manhattan_distance(state):
    distance = 0
    for i in range(9):
        tile = state[i]
        if tile != 0:
            goal_x, goal_y = goal_positions[tile]
            current_x, current_y = i // 3, i % 3
            distance += abs(goal_x - current_x) + abs(goal_y - current_y)
    return distance
# Generate possible moves (neighbors) by moving the blank tile
def get_neighbors(state):
    neighbors = []
    index = state.index(0)  # Blank tile position
    x, y = index // 3, index % 3
    # Define move directions (Up, Down, Left, Right)
    moves = {'Up': (x - 1, y), 'Down': (x + 1, y), 'Left': (x, y - 1), 'Right': (x, y + 1)}
    for move, (new_x, new_y) in moves.items():
        if 0 <= new_x < 3 and 0 <= new_y < 3:  # Check bounds
            new_index = new_x * 3 + new_y
            new_state = state[:]
            # Swap blank with the target tile
            new_state[index], new_state[new_index] = new_state[new_index], new_state[index]
            neighbors.append(new_state)
    return neighbors
# Hill Climbing Search Algorithm
def hill_climbing(start_state, heuristic):
```

```
    current_state = start_state
    current_cost = heuristic(current_state)
    while True:
        neighbors = get_neighbors(current_state)
        next_state = None
        next_cost = float('inf')
        # Evaluate each neighbor
        for neighbor in neighbors:
            cost = heuristic(neighbor)
            if cost < next_cost:
                next_state, next_cost = neighbor, cost
        # If no better neighbor, stop (local minimum reached)
        if next_cost >= current_cost:
            break
        # Move to the neighbor with the lower cost
        current_state, current_cost = next_state, next_cost

    return current_state, current_cost
# Generate a random starting state for testing
def generate_random_start():
    state = goal_state[:]
    random.shuffle(state)
    return state
# Example usage with both heuristics
start_state = generate_random_start()
print("Start State:")
display(start_state)
# Using Heuristic 1: Displaced Tiles
print("Using Heuristic h1 (Displaced Tiles):")
```

```
final_state, final_cost = hill_climbing(start_state, h1_displaced_tiles)
display(final_state)
print("Final Cost (Displaced Tiles):", final_cost)
# Using Heuristic 2: Manhattan Distance
print("Using Heuristic h2 (Manhattan Distance):")
final_state, final_cost = hill_climbing(start_state, h2_manhattan_distance)
display(final_state)
print("Final Cost (Manhattan Distance):", final_cost)
```

Output –

```
Start State:
[4, 8, 0]
[6, 1, 7]
[5, 2, 3]
Using Heuristic h1 (Displaced Tiles):
[4, 8, 0]
[6, 1, 7]
[5, 2, 3]
Final Cost (Displaced Tiles): 8
Using Heuristic h2 (Manhattan Distance):
[4, 8, 0]
[6, 1, 7]
[5, 2, 3]
Final Cost (Manhattan Distance): 16
=== Code Execution Successful ===
```

10. Implement Simulated Annealing Search Algorithm for solving the 8-puzzle problem. All other constraints are same as Assignment-9.

Code –

```
import random
```

```
import math

# Define the goal state and possible moves

goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]

goal_positions = {val: (i // 3, i % 3) for i, val in enumerate(goal_state)}  # Position map for Manhattan distance

def display(state):

    """Display the puzzle state in a 3x3 grid format."""

    for i in range(0, 9, 3):

        print(state[i:i+3])

    print()

def h1_displaced_tiles(state):

    """Heuristic 1: Number of misplaced tiles."""

    return sum(1 for i in range(9) if state[i] != goal_state[i] and state[i] != 0)

def h2_manhattan_distance(state):

    """Heuristic 2: Total Manhattan distance."""

    distance = 0

    for i in range(9):

        tile = state[i]

        if tile != 0:

            goal_x, goal_y = goal_positions[tile]

            current_x, current_y = i // 3, i % 3

            distance += abs(goal_x - current_x) + abs(goal_y - current_y)

    return distance

def get_neighbors(state):

    """Generate possible moves (neighbors) by moving the blank tile."""

    neighbors = []

    index = state.index(0)  # Blank tile position

    x, y = index // 3, index % 3

    # Define move directions (Up, Down, Left, Right)
```

```
    moves = {'Up': (x - 1, y), 'Down': (x + 1, y), 'Left': (x, y - 1), 'Right': (x, y + 1)}
    for move, (new_x, new_y) in moves.items():
        if 0 <= new_x < 3 and 0 <= new_y < 3:  # Check bounds
            new_index = new_x * 3 + new_y
            new_state = state[:]
            # Swap blank with the target tile
            new_state[index], new_state[new_index] = new_state[new_index], new_state[index]
            neighbors.append(new_state)
    return neighbors
def simulated_annealing(start_state, initial_temp=1000, cooling_rate=0.95, max_iterations=10000,
heuristic=h2_manhattan_distance):
    """Simulated Annealing Search Algorithm."""
    current_state = start_state
    current_cost = heuristic(current_state)
    temperature = initial_temp
    for iteration in range(max_iterations):
        if current_cost == 0:  # Goal state reached
            break
        neighbors = get_neighbors(current_state)
        next_state = random.choice(neighbors)
        next_cost = heuristic(next_state)
        # If the next state is better, move to it
        if next_cost < current_cost:
            current_state, current_cost = next_state, next_cost
        else:
            # Calculate probability of acceptance of worse state
            acceptance_probability = math.exp((current_cost - next_cost) / temperature)
            if random.random() < acceptance_probability:
                current_state, current_cost = next_state, next_cost
```

```
        # Cool down the temperature
        temperature *= cooling_rate
    return current_state, current_cost
# Generate a random starting state for testing
def generate_random_start():
    state = goal_state[:]
    random.shuffle(state)
    return state
# Example usage
start_state = generate_random_start()
print("Start State:")
display(start_state)
# Simulated Annealing with Heuristic 1 (Displaced Tiles)
print("Using Heuristic h1 (Displaced Tiles):")
final_state, final_cost = simulated_annealing(start_state, heuristic=h1_displaced_tiles)
display(final_state)
print("Final Cost (Displaced Tiles):", final_cost)
# Simulated Annealing with Heuristic 2 (Manhattan Distance)
print("Using Heuristic h2 (Manhattan Distance):")
final_state, final_cost = simulated_annealing(start_state, heuristic=h2_manhattan_distance)
display(final_state)
print("Final Cost (Manhattan Distance):", final_cost)
```

Output –

```
Start State:
[0, 2, 8]
[1, 6, 3]
[5, 7, 4]
Using Heuristic h1 (Displaced Tiles):
[0, 2, 1]
```

[6, 5, 4]

[3, 8, 7]

Final Cost (Displaced Tiles): 5

Using Heuristic h2 (Manhattan Distance):

[1, 8, 3]

[6, 5, 4]

[7, 2, 0]

Final Cost (Manhattan Distance): 8

=== Code Execution Successful ===

11. Implement genetic algorithm (GA) to solve 8-queen problem.

Code –

```
import random
class Queen:
  def _init_(self, board_size=8):
    self.board_size = board_size
    self.genome = random.sample(range(board_size), board_size)  # Randomly initialize the
genome
    self.fitness = self.calculate_fitness()
  def calculate_fitness(self):
    """Calculate fitness as the number of non-attacking pairs of queens."""
    conflicts = 0
    for i in range(self.board_size):
      for j in range(i + 1, self.board_size):
        if self.genome[i] == self.genome[j] or abs(self.genome[i] - self.genome[j]) == abs(i - j):
          conflicts += 1
    return (self.board_size * (self.board_size - 1)) // 2 - conflicts  # Max pairs - conflicts
  def mutate(self):
    """Randomly swap two queens to create a mutation."""
    idx1, idx2 = random.sample(range(self.board_size), 2)
```

```
        self.genome[idx1], self.genome[idx2] = self.genome[idx2], self.genome[idx1]
        self.fitness = self.calculate_fitness()

    def crossover(self, other):
        """Perform one-point crossover with another individual."""
        crossover_point = random.randint(1, self.board_size - 1)
        child_genome = self.genome[:crossover_point] + other.genome[crossover_point:]
        child = Queen(self.board_size)
        child.genome = child_genome
        child.fitness = child.calculate_fitness()
        return child

def genetic_algorithm(population_size=100, generations=1000, mutation_rate=0.1):
    """Run the Genetic Algorithm."""
    # Initialize population
    population = [Queen() for _ in range(population_size)]
    for generation in range(generations):
        # Sort population by fitness
        population.sort(key=lambda q: q.fitness, reverse=True)
        print(f"Generation {generation}: Best fitness = {population[-1].fitness}")
        # Check for a solution
        if population[-1].fitness == (8 * (8 - 1)) // 2:  # If fitness is maximum
            print("Solution found:")
            print(population[-1].genome)
            return population[-1]
        # Create a new generation
        new_population = []
        while len(new_population) < population_size:
            # Select parents using tournament selection
            parent1 = random.choice(population[-20:])  # Select from the best 20
            parent2 = random.choice(population[-20:])  # Select from the best 20
```

```
            # Crossover to create a child
            child = parent1.crossover(parent2)
            # Mutate the child with a given probability
            if random.random() < mutation_rate:
                child.mutate()
            new_population.append(child)
        population = new_population
    print("No solution found in given generations.")
    return None
# Run the Genetic Algorithm
result = genetic_algorithm(population_size=200, generations=1000, mutation_rate=0.1)
```

Output –

```
Generation 0: Best fitness = 14
Generation 1: Best fitness = 14
...
Generation 42: Best fitness = 28
Solution found:
[3, 6, 2, 5, 1, 4, 0, 7]
```