1.Genetic Algorithms for Solving the Traveling Salesman Problem

In [30]:

```python
import random

# Number of cities in TSP
V = 5

# Names of the cities
GENES = "ABCDE"

# Structure of an individual
class Individual:
    def __init__(self, gnome=None):
        if gnome is None:
            self.gnome = random.sample(GENES, V)
        else:
            self.gnome = gnome
        self.fitness = self.calculate_fitness()

    def calculate_fitness(self):
        total_distance = sum(
            distance(self.gnome[i], self.gnome[i + 1])
            for i in range(V - 1)
            if distance(self.gnome[i], self.gnome[i + 1]) != float('inf')
        )
        if distance(self.gnome[-1], self.gnome[0]) != float('inf'):
            total_distance += distance(self.gnome[-1], self.gnome[0])  # Return to the starting
        return total_distance

# Function to calculate distance between two cities
def distance(city1, city2):
    #float('inf') represents positive infinity; indicating there is no direct edge in between th
    #unbounded or undefined values and considering distances which are unreachable
    distances = {
        'A': [0, 2, float('inf'), 12, 5],
        'B': [2, 0, 4, 8, float('inf')],
        'C': [float('inf'), 4, 0, 3, 3],
        'D': [12, 8, 3, 0, 10],
        'E': [5, float('inf'), 3, 10, 0],
    }
    return distances[city1][GENES.index(city2)]

# Genetic algorithm function
def genetic_algorithm(population_size, num_generations):
    population = [Individual() for _ in range(population_size)]

    for generation in range(1, num_generations + 1):
        population.sort(key=lambda x: x.fitness)

        new_population = []

        for _ in range(population_size // 2):
            parent1, parent2 = random.sample(population[:population_size // 2], 2)
            child_gnome = crossover(parent1.gnome, parent2.gnome)
            new_population.extend([Individual(child_gnome)])

        population = new_population

        print(f"Generation {generation}: Best Fitness - {population[0].fitness}")

    best_tour = population[0]
    print("Best Tour:", best_tour.gnome)
    print("Best Fitness:", best_tour.fitness)

# Crossover function (Order Crossover)
def crossover(parent1, parent2):
    start, end = sorted(random.sample(range(V), 2))
    child_gnome = [-1] * V

    child_gnome[start:end + 1] = parent1[start:end + 1]

    remaining_indices = [i for i in range(V) if parent2[i] not in child_gnome]
    remaining_values = iter(parent2[i] for i in remaining_indices)

    for i in range(V):
        if child_gnome[i] == -1:
            child_gnome[i] = next(remaining_values)
```

```
    return child_gnome

if __name__ == "__main__":
    POP_SIZE = 10
    NUM_GENERATIONS = 5

    genetic_algorithm(POP_SIZE, NUM_GENERATIONS)
```

```
Generation 1: Best Fitness - 23
Generation 2: Best Fitness - 32
Generation 3: Best Fitness - 16
Generation 4: Best Fitness - 24
Generation 5: Best Fitness - 16
Best Tour: ['B', 'D', 'C', 'A', 'E']
Best Fitness: 16
```

3.Use Local Search Algorithms for Solving the N-Queens Problem

In [50]:

```
import random

def generate_random_board(n):
    """ Generate a random board configuration """
    board = list(range(n))
    random.shuffle(board)
    return board

def num_attacking_queens(board):
    """ Calculate the number of pairs of attacking queens """
    n = len(board)
    attacking_pairs = 0
    for i in range(n):
        for j in range(i + 1, n):
            if abs(i - j) == abs(board[i] - board[j]):
                attacking_pairs += 1
    return attacking_pairs

def get_next_board(board):
    """ Generate next board by moving one queen to minimize attacks """
    n = len(board)
    min_attacks = num_attacking_queens(board)
    next_board = board[:]

    # Try moving each queen to minimize attacks
    for i in range(n):
        for j in range(n):
            if j != board[i]:  # Try moving queen i to column j
                new_board = board[:]
                new_board[i] = j
                num_attacks = num_attacking_queens(new_board)
                if num_attacks < min_attacks:
                    min_attacks = num_attacks
                    next_board = new_board

    return next_board

def hill_climbing(n):
    """ Solve N-Queens problem using Hill Climbing """
    current_board = generate_random_board(n)
    current_attacks = num_attacking_queens(current_board)

    while current_attacks > 0:
        next_board = get_next_board(current_board)
        next_attacks = num_attacking_queens(next_board)

        if next_attacks >= current_attacks:
            break  # Stop if no better move

        current_board = next_board
        current_attacks = next_attacks

    return current_board

def print_board(board):
    """ Print board configuration """
    n = len(board)
    for row in range(n):
        line = ['Q' if board[row] == col else '.' for col in range(n)]
        print(' '.join(line))

if __name__ == "__main__":
    n = 8  # Change this to desired board size
    solution = hill_climbing(n)
    print("Solution:")
    print_board(solution)
```

```
Solution:
. . Q . . . . .
. . . . Q . . .
. Q . . . . . .
. . . Q . . . .
Q . . . . . . .
. . Q . . . . .
. . . . . . . Q
. Q . . . . . .
```

4.Depth-First Search for Solving the Tower of Hanoi Problem

In [4]:

```python
def tower_of_hanoi(n, source, target, auxiliary):
    """ Solves the Tower of Hanoi problem using Depth-First Search (DFS) """
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
        return

    # Move n-1 disks from source to auxiliary using target as temporary
    tower_of_hanoi(n - 1, source, auxiliary, target)

    # Move the nth disk from source to target
    print(f"Move disk {n} from {source} to {target}")

    # Move n-1 disks from auxiliary to target using source as temporary
    tower_of_hanoi(n - 1, auxiliary, target, source)

if __name__ == "__main__":
    n = 5  # Number of disks (change this to increase the number of disks)
    tower_of_hanoi(n, 'A', 'C', 'B')  # 'A' is source, 'C' is target, 'B' is auxiliary
```

```
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
Move disk 4 from A to B
Move disk 1 from C to B
Move disk 2 from C to A
Move disk 1 from B to A
Move disk 3 from C to B
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 5 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
Move disk 3 from B to A
Move disk 1 from C to B
Move disk 2 from C to A
Move disk 1 from B to A
Move disk 4 from B to C
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

5.Breadth-First Search for Solving the Tower of Hanoi Problem

In [5]:

```
from collections import deque

class State:
    def __init__(self, disks, source='A', target='C', auxiliary='B'):
        self.disks = disks
        self.source = source
        self.target = target
        self.auxiliary = auxiliary

    def __eq__(self, other):
        return self.disks == other.disks and \
               self.source == other.source and \
               self.target == other.target and \
               self.auxiliary == other.auxiliary

    def __hash__(self):
        return hash((self.disks, self.source, self.target, self.auxiliary))

def get_next_states(state):
    next_states = []
    if state.disks > 0:
        # Move the top disk from source to target
        next_states.append(State(state.disks - 1, state.source, state.auxiliary, state.target))
        next_states.append(State(state.disks, state.auxiliary, state.target, state.source))
    return next_states

def bfs_tower_of_hanoi(disks):
    initial_state = State(disks)
    target_state = State(0)  # All disks on target rod

    queue = deque([(initial_state, [])])  # (current_state, path)
    visited = set()
    visited.add(initial_state)

    while queue:
        current_state, path = queue.popleft()

        if current_state == target_state:
            return path

        for next_state in get_next_states(current_state):
            if next_state not in visited:
                visited.add(next_state)
                queue.append((next_state, path + [next_state]))

    return None  # No solution found

if __name__ == "__main__":
    disks = 3
    solution_path = bfs_tower_of_hanoi(disks)

    if solution_path is not None:
        print(f"Solution found with {len(solution_path)} moves:")
        for i, state in enumerate(solution_path):
            print(f"Move {i + 1}: {state.source} -> {state.target}")
    else:
        print("No solution found for the Tower of Hanoi problem with the given number of disks."
```

```
Solution found with 6 moves:
Move 1: B -> C
Move 2: B -> A
Move 3: C -> A
Move 4: C -> B
Move 5: A -> B
Move 6: A -> C
```

6.A* Search for Solving the Eight Puzzle Problem

In [6]:

```
import heapq
from collections import deque

# Define the goal state for the Eight Puzzle (1-8 representing tiles, 0 representing the empty sp
goal_state = (1, 2, 3, 4, 5, 6, 7, 8, 0)

# Define the moves (left, right, up, down) represented as (column_delta, row_delta)
moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

def get_manhattan_distance(state):
    """ Calculate the total Manhattan distance heuristic for the given state """
    distance = 0
    for i in range(9):
        if state[i] != 0:  # Ignore the empty space (represented by 0)
            current_row, current_col = i // 3, i % 3
            target_row, target_col = (state[i] - 1) // 3, (state[i] - 1) % 3
            distance += abs(current_row - target_row) + abs(current_col - target_col)
    return distance

def is_valid_move(x, y):
    """ Check if the move (x, y) is within the bounds of the 3x3 grid """
    return 0 <= x < 3 and 0 <= y < 3

def apply_move(state, move):
    """ Apply a move (column_delta, row_delta) to the state """
    empty_index = state.index(0)
    empty_row, empty_col = empty_index // 3, empty_index % 3
    new_col = empty_col + move[0]
    new_row = empty_row + move[1]
    new_index = new_row * 3 + new_col
    new_state = list(state)
    new_state[empty_index], new_state[new_index] = new_state[new_index], new_state[empty_index]
    return tuple(new_state)

def a_star_search(initial_state):
    """ Solve the Eight Puzzle using A* search with Manhattan distance heuristic """
    priority_queue = []
    heapq.heappush(priority_queue, (0 + get_manhattan_distance(initial_state), 0, initial_state)
    visited = set()
    visited.add(initial_state)

    while priority_queue:
        _, cost, current_state = heapq.heappop(priority_queue)

        if current_state == goal_state:
            return cost

        empty_index = current_state.index(0)
        empty_row, empty_col = empty_index // 3, empty_index % 3

        for move in moves:
            new_col = empty_col + move[0]
            new_row = empty_row + move[1]

            if is_valid_move(new_col, new_row):
                new_state = apply_move(current_state, move)

                if new_state not in visited:
                    visited.add(new_state)
                    new_cost = cost + 1
                    priority = new_cost + get_manhattan_distance(new_state)
                    heapq.heappush(priority_queue, (priority, new_cost, new_state))

    return -1  # No solution found

if __name__ == "__main__":
    # Example initial state (customize with your own initial state)
    initial_state = (1, 2, 3, 4, 5, 6, 7, 0, 8)  # Use 0 to represent the empty space

    # Solve the Eight Puzzle using A* search
    steps = a_star_search(initial_state)

    if steps != -1:
        print(f"Solution found in {steps} steps.")
    else:
```

```
        print("No solution found for the given initial state.")
```

```
Solution found in 1 steps.
```

7.Iterative Deepening Depth-First Search for Solving the Eight Puzzle Problem

In [8]:

```
# Define the goal state for the Eight Puzzle (1-8 representing tiles, 0 representing the empty sp
goal_state = (1, 2, 3, 4, 5, 6, 7, 8, 0)

# Define the moves (left, right, up, down) represented as (column_delta, row_delta)
moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

def is_valid_move(x, y):
    """ Check if the move (x, y) is within the bounds of the 3x3 grid """
    return 0 <= x < 3 and 0 <= y < 3

def apply_move(state, move):
    """ Apply a move (column_delta, row_delta) to the state """
    empty_index = state.index(0)
    empty_row, empty_col = empty_index // 3, empty_index % 3
    new_col = empty_col + move[0]
    new_row = empty_row + move[1]
    new_index = new_row * 3 + new_col
    new_state = list(state)
    new_state[empty_index], new_state[new_index] = new_state[new_index], new_state[empty_index]
    return tuple(new_state)

def depth_limited_dfs(current_state, depth_limit, path):
    """ Perform depth-limited DFS up to a specified depth limit """
    if current_state == goal_state:
        return path  # Return the path if the goal state is reached

    if depth_limit == 0:
        return None  # Depth limit reached without finding the goal state

    empty_index = current_state.index(0)
    empty_row, empty_col = empty_index // 3, empty_index % 3

    for move in moves:
        new_col = empty_col + move[0]
        new_row = empty_row + move[1]

        if is_valid_move(new_col, new_row):
            new_state = apply_move(current_state, move)

            if new_state not in path:  # Avoid revisiting states to prevent cycles
                result = depth_limited_dfs(new_state, depth_limit - 1, path + [new_state])
                if result is not None:
                    return result

    return None  # No solution found within the depth limit

def iterative_deepening_dfs(initial_state):
    """ Perform iterative deepening DFS to solve the Eight Puzzle """
    depth_limit = 0
    while True:
        result = depth_limited_dfs(initial_state, depth_limit, [initial_state])
        if result is not None:
            return result  # Solution found
        depth_limit += 1  # Increase depth limit for the next iteration

if __name__ == "__main__":
    # Example initial state (customize with your own initial state)
    initial_state = (1, 2, 3, 4, 5, 6, 7, 0, 8)  # Use 0 to represent the empty space

    # Solve the Eight Puzzle using iterative deepening DFS
    solution_path = iterative_deepening_dfs(initial_state)

    if solution_path is not None:
        print(f"Solution found in {len(solution_path) - 1} steps:")
        for i, state in enumerate(solution_path):
            print(f"Step {i}: {state}")
    else:
        print("No solution found for the given initial state within the search depth limit.")
```

```
Solution found in 1 steps:
Step 0: (1, 2, 3, 4, 5, 6, 7, 0, 8)
Step 1: (1, 2, 3, 4, 5, 6, 7, 8, 0)
```

8.Uniform Cost Search for Solving the Eight Puzzle Problem

In [9]:

```
import heapq

# Define the goal state for the Eight Puzzle (1-8 representing tiles, 0 representing the empty sp
goal_state = (1, 2, 3, 4, 5, 6, 7, 8, 0)

# Define the moves (left, right, up, down) represented as (column_delta, row_delta)
moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

def is_valid_move(x, y):
    """ Check if the move (x, y) is within the bounds of the 3x3 grid """
    return 0 <= x < 3 and 0 <= y < 3

def apply_move(state, move):
    """ Apply a move (column_delta, row_delta) to the state """
    empty_index = state.index(0)
    empty_row, empty_col = empty_index // 3, empty_index % 3
    new_col = empty_col + move[0]
    new_row = empty_row + move[1]
    new_index = new_row * 3 + new_col
    new_state = list(state)
    new_state[empty_index], new_state[new_index] = new_state[new_index], new_state[empty_index]
    return tuple(new_state)

def calculate_path_cost(path):
    """ Calculate the path cost (number of moves) """
    return len(path) - 1  # Subtract 1 to exclude the initial state

def uniform_cost_search(initial_state):
    """ Perform Uniform Cost Search to solve the Eight Puzzle """
    priority_queue = []
    heapq.heappush(priority_queue, (0, [initial_state]))  # (path_cost, path)
    explored = set()

    while priority_queue:
        current_cost, current_path = heapq.heappop(priority_queue)
        current_state = current_path[-1]

        if current_state == goal_state:
            return current_path

        explored.add(current_state)
        empty_index = current_state.index(0)
        empty_row, empty_col = empty_index // 3, empty_index % 3

        for move in moves:
            new_col = empty_col + move[0]
            new_row = empty_row + move[1]

            if is_valid_move(new_col, new_row):
                new_state = apply_move(current_state, move)

                if new_state not in explored:
                    new_path = current_path + [new_state]
                    new_cost = calculate_path_cost(new_path)
                    heapq.heappush(priority_queue, (new_cost, new_path))

    return None  # No solution found

if __name__ == "__main__":
    # Example initial state (customize with your own initial state)
    initial_state = (1, 2, 3, 4, 5, 6, 7, 0, 8)  # Use 0 to represent the empty space

    # Solve the Eight Puzzle using Uniform Cost Search
    solution_path = uniform_cost_search(initial_state)

    if solution_path is not None:
        print(f"Solution found in {calculate_path_cost(solution_path)} steps:")
        for i, state in enumerate(solution_path):
            print(f"Step {i}: {state}")
    else:
        print("No solution found for the given initial state.")
```

```
Solution found in 1 steps:
Step 0: (1, 2, 3, 4, 5, 6, 7, 0, 8)
Step 1: (1, 2, 3, 4, 5, 6, 7, 8, 0)
```

9.Heuristic Search Algorithms for Solving the Missionaries and Cannibals Problem

In [10]:

```
import heapq

# Define the goal state and initial state
goal_state = (0, 0, 1)  # All missionaries and cannibals on the right bank
initial_state = (3, 3, 0)  # All missionaries and cannibals on the left bank

# Define moves (m, c) representing the number of missionaries and cannibals moved
moves = [
    (1, 0),  # Move one missionary from left to right
    (2, 0),  # Move two missionaries from left to right
    (0, 1),  # Move one cannibal from left to right
    (0, 2),  # Move two cannibals from left to right
    (1, 1)   # Move one missionary and one cannibal from left to right
]

def is_valid_state(state):
    """ Check if a state is valid (no missionaries eaten by cannibals) """
    missionaries_left, cannibals_left, boat_side = state
    missionaries_right = 3 - missionaries_left
    cannibals_right = 3 - cannibals_left

    # Check if missionaries are outnumbered by cannibals on either side
    if missionaries_left > 0 and missionaries_left < cannibals_left:
        return False
    if missionaries_right > 0 and missionaries_right < cannibals_right:
        return False

    return True

def apply_move(state, move):
    """ Apply a move (m, c) to a given state """
    missionaries_left, cannibals_left, boat_side = state
    m, c = move

    if boat_side == 0:
        # Moving from left to right
        new_state = (missionaries_left - m, cannibals_left - c, 1)
    else:
        # Moving from right to left
        new_state = (missionaries_left + m, cannibals_left + c, 0)

    return new_state

def heuristic(state):
    """ Heuristic function: estimate the minimum number of moves to reach the goal """
    # Use the sum of missionaries and cannibals on the left bank as the heuristic estimate
    missionaries_left, cannibals_left, _ = state
    return missionaries_left + cannibals_left

def a_star_search():
    """ A* search to solve the Missionaries and Cannibals Problem """
    priority_queue = []
    heapq.heappush(priority_queue, (heuristic(initial_state), initial_state, []))
    visited = set()

    while priority_queue:
        _, current_state, path = heapq.heappop(priority_queue)

        if current_state == goal_state:
            return path

        if current_state not in visited:
            visited.add(current_state)
            missionaries_left, cannibals_left, boat_side = current_state

            for move in moves:
                new_state = apply_move(current_state, move)

                if is_valid_state(new_state):
                    new_path = path + [new_state]
                    cost = len(new_path) + heuristic(new_state)
                    heapq.heappush(priority_queue, (cost, new_state, new_path))

    return None  # No solution found
```

```
def print_solution_path(solution_path):
    """ Print the solution path """
    if solution_path is None:
        print("No solution found.")
    else:
        print("Solution found:")
        for i, state in enumerate(solution_path):
            missionaries_left, cannibals_left, boat_side = state
            missionaries_right = 3 - missionaries_left
            cannibals_right = 3 - cannibals_left
            boat_location = "left" if boat_side == 0 else "right"
            print(f"Step {i}: {missionaries_left}M-{cannibals_left}C ({boat_location}) <-> {miss

if __name__ == "__main__":
    # Solve the Missionaries and Cannibals Problem using A* search
    solution_path = a_star_search()
    print_solution_path(solution_path)
```

```
Solution found:
Step 0: 2M-2C (right) <-> 1M-1C
Step 1: 3M-2C (left) <-> 0M-1C
Step 2: 3M-0C (right) <-> 0M-3C
Step 3: 3M-1C (left) <-> 0M-2C
Step 4: 1M-1C (right) <-> 2M-2C
Step 5: 2M-2C (left) <-> 1M-1C
Step 6: 0M-2C (right) <-> 3M-1C
Step 7: 0M-3C (left) <-> 3M-0C
Step 8: -1M-2C (right) <-> 4M-1C
Step 9: 0M-2C (left) <-> 3M-1C
Step 10: 0M-0C (right) <-> 3M-3C
```

10.Use Breadth-First Search (BFS) for solving the Missionaries and Cannibals problem

In [11]:

```
from collections import deque

# Define the goal state and initial state
goal_state = (0, 0, 1)  # All missionaries and cannibals on the right bank
initial_state = (3, 3, 0)  # All missionaries and cannibals on the left bank

# Define moves (m, c) representing the number of missionaries and cannibals moved
moves = [
    (1, 0),  # Move one missionary from left to right
    (2, 0),  # Move two missionaries from left to right
    (0, 1),  # Move one cannibal from left to right
    (0, 2),  # Move two cannibals from left to right
    (1, 1)   # Move one missionary and one cannibal from left to right
]

def is_valid_state(state):
    """ Check if a state is valid (no missionaries eaten by cannibals) """
    missionaries_left, cannibals_left, boat_side = state
    missionaries_right = 3 - missionaries_left
    cannibals_right = 3 - cannibals_left

    # Check if missionaries are outnumbered by cannibals on either side
    if missionaries_left > 0 and missionaries_left < cannibals_left:
        return False
    if missionaries_right > 0 and missionaries_right < cannibals_right:
        return False

    return True

def apply_move(state, move):
    """ Apply a move (m, c) to a given state """
    missionaries_left, cannibals_left, boat_side = state
    m, c = move

    if boat_side == 0:
        # Moving from left to right
        new_state = (missionaries_left - m, cannibals_left - c, 1)
    else:
        # Moving from right to left
        new_state = (missionaries_left + m, cannibals_left + c, 0)

    return new_state

def bfs_search():
    """ Breadth-First Search to solve the Missionaries and Cannibals Problem """
    queue = deque([(initial_state, [])])  # (current_state, path)
    visited = set()

    while queue:
        current_state, path = queue.popleft()

        if current_state == goal_state:
            return path

        if current_state not in visited:
            visited.add(current_state)
            missionaries_left, cannibals_left, boat_side = current_state

            for move in moves:
                new_state = apply_move(current_state, move)

                if is_valid_state(new_state):
                    new_path = path + [new_state]
                    queue.append((new_state, new_path))

    return None  # No solution found

def print_solution_path(solution_path):
    """ Print the solution path """
    if solution_path is None:
        print("No solution found.")
    else:
        print("Solution found:")
        for i, state in enumerate(solution_path):
            missionaries_left, cannibals_left, boat_side = state
```

```
            missionaries_right = 3 - missionaries_left
            cannibals_right = 3 - cannibals_left
            boat_location = "left" if boat_side == 0 else "right"
            print(f"Step {i}: {missionaries_left}M-{cannibals_left}C ({boat_location}) <-> {miss

if __name__ == "__main__":
    # Solve the Missionaries and Cannibals Problem using BFS
    solution_path = bfs_search()
    print_solution_path(solution_path)
```

```
Solution found:
Step 0: 3M-1C (right) <-> 0M-2C
Step 1: 4M-1C (left) <-> -1M-2C
Step 2: 3M-0C (right) <-> 0M-3C
Step 3: 3M-1C (left) <-> 0M-2C
Step 4: 1M-1C (right) <-> 2M-2C
Step 5: 2M-2C (left) <-> 1M-1C
Step 6: 0M-2C (right) <-> 3M-1C
Step 7: 0M-3C (left) <-> 3M-0C
Step 8: 0M-1C (right) <-> 3M-2C
Step 9: 1M-1C (left) <-> 2M-2C
Step 10: 0M-0C (right) <-> 3M-3C
```

11.Use Depth-First Search (DFS) for solving the Missionaries and Cannibals problem

In [32]:

```
class State:
    def __init__(self, left_m, left_c, boat, right_m, right_c):
        self.left_m = left_m
        self.left_c = left_c
        self.boat = boat
        self.right_m = right_m
        self.right_c = right_c

    def is_valid(self):
        # Check if the state is valid (no missionaries eaten)
        return (self.left_m == 0 or self.left_m >= self.left_c) and \
               (self.right_m == 0 or self.right_m >= self.right_c)

    def is_goal(self):
        # Check if the state is the goal state
        return self.left_m == 0 and self.left_c == 0

    def __str__(self):
        return f"Left: {self.left_m}M {self.left_c}C Boat: {self.boat} Right: {self.right_m}M {s

def move(state, m, c):
    if state.boat == 'left':
        return State(state.left_m - m, state.left_c - c, 'right', state.right_m + m, state.right
    else:
        return State(state.left_m + m, state.left_c + c, 'left', state.right_m - m, state.right_

def dfs(current_state, visited_states, path):
    if current_state.is_goal():
        path.append(current_state)
        return True

    visited_states.add(str(current_state))

    for m, c in [(0, 1), (1, 0), (1, 1), (2, 0), (0, 2)]:
        next_state = move(current_state, m, c)

        if next_state.is_valid() and str(next_state) not in visited_states:
            path.append(current_state)
            if dfs(next_state, visited_states, path):
                return True
            path.pop()

    return False

def solve():
    initial_state = State(3, 3, 'left', 0, 0)
    visited_states = set()
    path = []

    if dfs(initial_state, visited_states, path):
        print("Solution found:")
        for step, state in enumerate(path):
            print(f"Step {step + 1}: {state}")
    else:
        print("No solution found.")

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

```
Solution found:
Step 1: Left: 3M 3C Boat: left Right: 0M 0C
Step 2: Left: 2M 2C Boat: right Right: 1M 1C
Step 3: Left: 3M 2C Boat: left Right: 0M 1C
Step 4: Left: 3M 0C Boat: right Right: 0M 3C
Step 5: Left: 3M 1C Boat: left Right: 0M 2C
Step 6: Left: 1M 1C Boat: right Right: 2M 2C
Step 7: Left: 2M 2C Boat: left Right: 1M 1C
Step 8: Left: 0M 2C Boat: right Right: 3M 1C
Step 9: Left: 0M 3C Boat: left Right: 3M 0C
Step 10: Left: 0M 1C Boat: right Right: 3M 2C
Step 11: Left: 0M 2C Boat: left Right: 3M 1C
Step 12: Left: 0M 0C Boat: right Right: 3M 3C
```

12.Use Iterative Deepening Depth-First Search (IDDFS)for solving the Missionaries and Cannibals problem

In [18]:

```python
# Define the goal state and initial state
goal_state = (0, 0, 1)  # All missionaries and cannibals on the right bank
initial_state = (3, 3, 0)  # All missionaries and cannibals on the left bank

# Define moves (m, c) representing the number of missionaries and cannibals moved
moves = [
    (1, 0),  # Move one missionary from left to right
    (2, 0),  # Move two missionaries from left to right
    (0, 1),  # Move one cannibal from left to right
    (0, 2),  # Move two cannibals from left to right
    (1, 1)   # Move one missionary and one cannibal from left to right
]

def is_valid_state(state):
    """ Check if a state is valid (no missionaries eaten by cannibals) """
    missionaries_left, cannibals_left, boat_side = state
    missionaries_right = 3 - missionaries_left
    cannibals_right = 3 - cannibals_left

    # Check if missionaries are outnumbered by cannibals on either side
    if missionaries_left > 0 and missionaries_left < cannibals_left:
        return False
    if missionaries_right > 0 and missionaries_right < cannibals_right:
        return False

    return True

def apply_move(state, move):
    """ Apply a move (m, c) to a given state """
    missionaries_left, cannibals_left, boat_side = state
    m, c = move

    if boat_side == 0:
        # Moving from left to right
        new_state = (missionaries_left - m, cannibals_left - c, 1)
    else:
        # Moving from right to left
        new_state = (missionaries_left + m, cannibals_left + c, 0)

    return new_state

def dls_search(current_state, path, depth_limit):
    """ Depth-Limited Search (DLS) for a specific depth limit """
    if depth_limit == 0 and current_state != goal_state:
        return None

    if current_state == goal_state:
        return path

    missionaries_left, cannibals_left, boat_side = current_state

    for move in moves:
        new_state = apply_move(current_state, move)

        if is_valid_state(new_state):
            new_path = path + [new_state]
            result = dls_search(new_state, new_path, depth_limit - 1)
            if result is not None:
                return result

    return None

def iddfs_search():
    """ Iterative Deepening Depth-First Search (IDDFS) to solve the Missionaries and Cannibals P
    depth_limit = 0

    while True:
        initial_path = [(initial_state)]
        result = dls_search(initial_state, initial_path, depth_limit)
        if result is not None:
            return result
        depth_limit += 1

def print_solution_path(solution_path):
    """ Print the solution path """
```

```
    if solution_path is None:
        print("No solution found.")
    else:
        print("Solution found:")
        for i, state in enumerate(solution_path):
            missionaries_left, cannibals_left, boat_side = state
            missionaries_right = 3 - missionaries_left
            cannibals_right = 3 - cannibals_left
            boat_location = "left" if boat_side == 0 else "right"
            print(f"Step {i}: {missionaries_left}M-{cannibals_left}C ({boat_location}) <-> {miss

if __name__ == "__main__":
    # Solve the Missionaries and Cannibals Problem using IDDFS
    solution_path = iddfs_search()
    print_solution_path(solution_path)
```

```
Solution found:
Step 0: 3M-3C (left) <-> 0M-0C
Step 1: 3M-1C (right) <-> 0M-2C
Step 2: 4M-1C (left) <-> -1M-2C
Step 3: 3M-0C (right) <-> 0M-3C
Step 4: 3M-1C (left) <-> 0M-2C
Step 5: 1M-1C (right) <-> 2M-2C
Step 6: 2M-2C (left) <-> 1M-1C
Step 7: 0M-2C (right) <-> 3M-1C
Step 8: 0M-3C (left) <-> 3M-0C
Step 9: 0M-1C (right) <-> 3M-2C
Step 10: 1M-1C (left) <-> 2M-2C
Step 11: 0M-0C (right) <-> 3M-3C
```

13.Use Uniform Cost Search (UCS)for solving the Missionaries and Cannibals problem: Brignt

In [19]:

```python
import heapq

# Define the goal state and initial state
goal_state = (0, 0, 1)  # All missionaries and cannibals on the right bank
initial_state = (3, 3, 0)  # All missionaries and cannibals on the left bank

# Define moves (m, c) representing the number of missionaries and cannibals moved
moves = [
    (1, 0),  # Move one missionary from left to right
    (2, 0),  # Move two missionaries from left to right
    (0, 1),  # Move one cannibal from left to right
    (0, 2),  # Move two cannibals from left to right
    (1, 1)   # Move one missionary and one cannibal from left to right
]

def is_valid_state(state):
    """ Check if a state is valid (no missionaries eaten by cannibals) """
    missionaries_left, cannibals_left, boat_side = state
    missionaries_right = 3 - missionaries_left
    cannibals_right = 3 - cannibals_left

    # Check if missionaries are outnumbered by cannibals on either side
    if missionaries_left > 0 and missionaries_left < cannibals_left:
        return False
    if missionaries_right > 0 and missionaries_right < cannibals_right:
        return False

    return True

def apply_move(state, move):
    """ Apply a move (m, c) to a given state """
    missionaries_left, cannibals_left, boat_side = state
    m, c = move

    if boat_side == 0:
        # Moving from left to right
        new_state = (missionaries_left - m, cannibals_left - c, 1)
    else:
        # Moving from right to left
        new_state = (missionaries_left + m, cannibals_left + c, 0)

    return new_state

def ucs_search():
    """ Uniform Cost Search (UCS) to solve the Missionaries and Cannibals Problem """
    priority_queue = []
    heapq.heappush(priority_queue, (0, [initial_state]))  # (cost, path)
    visited = set()

    while priority_queue:
        current_cost, current_path = heapq.heappop(priority_queue)
        current_state = current_path[-1]

        if current_state == goal_state:
            return current_path

        if current_state not in visited:
            visited.add(current_state)
            missionaries_left, cannibals_left, boat_side = current_state

            for move in moves:
                new_state = apply_move(current_state, move)

                if is_valid_state(new_state):
                    new_cost = current_cost + 1  # Uniform cost search (each move cost = 1)
                    new_path = current_path + [new_state]
                    heapq.heappush(priority_queue, (new_cost, new_path))

    return None  # No solution found

def print_solution_path(solution_path):
    """ Print the solution path """
    if solution_path is None:
        print("No solution found.")
    else:
```

```
        print("Solution found:")
        for i, state in enumerate(solution_path):
            missionaries_left, cannibals_left, boat_side = state
            missionaries_right = 3 - missionaries_left
            cannibals_right = 3 - cannibals_left
            boat_location = "left" if boat_side == 0 else "right"
            print(f"Step {i}: {missionaries_left}M-{cannibals_left}C ({boat_location}) <-> {miss

if __name__ == "__main__":
    # Solve the Missionaries and Cannibals Problem using Uniform Cost Search (UCS)
    solution_path = ucs_search()
    print_solution_path(solution_path)
```

```
Solution found:
Step 0: 3M-3C (left) <-> 0M-0C
Step 1: 2M-2C (right) <-> 1M-1C
Step 2: 3M-2C (left) <-> 0M-1C
Step 3: 3M-0C (right) <-> 0M-3C
Step 4: 3M-1C (left) <-> 0M-2C
Step 5: 1M-1C (right) <-> 2M-2C
Step 6: 2M-2C (left) <-> 1M-1C
Step 7: 0M-2C (right) <-> 3M-1C
Step 8: 0M-3C (left) <-> 3M-0C
Step 9: -1M-2C (right) <-> 4M-1C
Step 10: 0M-2C (left) <-> 3M-1C
Step 11: 0M-0C (right) <-> 3M-3C
```

14.Use Greedy Best-First Search for solving the Missionaries and Cannibals problem

In [38]:

```python
import heapq

# Define the goal state
goal_state = (0, 0)

# Define the initial state
initial_state = (3, 3)

# Define the heuristic function
def heuristic(state):
    return state[0] + state[1]

# Define a function to check if a state is valid
def is_valid(state):
    missionaries, cannibals = state
    if missionaries < 0 or missionaries > 3 or cannibals < 0 or cannibals > 3:
        return False
    if missionaries < cannibals and missionaries > 0:
        return False
    if 3 - missionaries < 3 - cannibals and missionaries < 3:
        return False
    return True

# Define a function to find the possible moves from a given state
def find_moves(state):
    moves = []
    for i in range(3):
        for j in range(3):
            if i + j > 0 and i + j <= 2:
                moves.append((i, j))
    return moves

# Define a function to apply a move to a state
def apply_move(state, move):
    missionaries, cannibals = state
    move_m, move_c = move
    if state == initial_state:  # Check the side of the boat
        new_state = (missionaries - move_m, cannibals - move_c)
    else:
        new_state = (missionaries + move_m, cannibals + move_c)
    return new_state

# Greedy Best-First Search algorithm
def greedy_best_first_search():
    frontier = [(heuristic(initial_state), initial_state, [])]  # Priority queue sorted by heuri
    explored = set()  # Set to keep track of explored states

    while frontier:
        _, current_state, path = heapq.heappop(frontier)
        if current_state == goal_state:
            return path

        explored.add(current_state)

        for move in find_moves(current_state):
            new_state = apply_move(current_state, move)
            if new_state not in explored and is_valid(new_state):
                heapq.heappush(frontier, (heuristic(new_state), new_state, path + [move]))

    return None  # No solution found

# Example usage
if __name__ == "__main__":
    solution = greedy_best_first_search()
    if solution:
        print("Moves to solve the Missionaries and Cannibals problem:")
        for i, move in enumerate(solution):
            print("Step", i + 1, ": Move", move)
    else:
        print("No solution found.")
```

```
No solution found.
```

15.Use A* Search for solving the Missionaries and Cannibals problem

In [24]:

```
import heapq

# Define the goal state and initial state
goal_state = (0, 0, 1)  # All missionaries and cannibals on the right bank
initial_state = (3, 3, 0)  # All missionaries and cannibals on the left bank

# Define moves (m, c) representing the number of missionaries and cannibals moved
moves = [
    (1, 0),  # Move one missionary from left to right
    (2, 0),  # Move two missionaries from left to right
    (0, 1),  # Move one cannibal from left to right
    (0, 2),  # Move two cannibals from left to right
    (1, 1)   # Move one missionary and one cannibal from left to right
]

def is_valid_state(state):
    """ Check if a state is valid (no missionaries eaten by cannibals) """
    missionaries_left, cannibals_left, boat_side = state
    missionaries_right = 3 - missionaries_left
    cannibals_right = 3 - cannibals_left

    # Check if missionaries are outnumbered by cannibals on either side
    if missionaries_left > 0 and missionaries_left < cannibals_left:
        return False
    if missionaries_right > 0 and missionaries_right < cannibals_right:
        return False

    return True

def apply_move(state, move):
    """ Apply a move (m, c) to a given state """
    missionaries_left, cannibals_left, boat_side = state
    m, c = move

    if boat_side == 0:
        # Moving from left to right
        new_state = (missionaries_left - m, cannibals_left - c, 1)
    else:
        # Moving from right to left
        new_state = (missionaries_left + m, cannibals_left + c, 0)

    return new_state

def a_star_search():
    """ A* Search using a heuristic to solve the Missionaries and Cannibals Problem """
    priority_queue = []
    initial_heuristic = heuristic(initial_state)
    heapq.heappush(priority_queue, (initial_heuristic, [initial_state]))  # (heuristic_value, pa
    visited = set()
    cost_so_far = {initial_state: 0}

    while priority_queue:
        _, current_path = heapq.heappop(priority_queue)
        current_state = current_path[-1]

        if current_state == goal_state:
            return current_path

        if current_state not in visited:
            visited.add(current_state)
            missionaries_left, cannibals_left, boat_side = current_state

            for move in moves:
                new_state = apply_move(current_state, move)

                if is_valid_state(new_state):
                    new_path = current_path + [new_state]
                    new_cost = cost_so_far[current_state] + 1  # Assuming uniform cost per move

                    if new_state not in cost_so_far or new_cost < cost_so_far[new_state]:
                        cost_so_far[new_state] = new_cost
                        priority = new_cost + heuristic(new_state)
                        heapq.heappush(priority_queue, (priority, new_path))

    return None  # No solution found
```

```
def heuristic(state):
    """ Heuristic function for the Missionaries and Cannibals Problem """
    missionaries_left, cannibals_left, _ = state
    return missionaries_left + cannibals_left  # Heuristic: sum of missionaries and cannibals on

def print_solution_path(solution_path):
    """ Print the solution path """
    if solution_path is None:
        print("No solution found.")
    else:
        print("Solution found:")
        for i, state in enumerate(solution_path):
            missionaries_left, cannibals_left, boat_side = state
            missionaries_right = 3 - missionaries_left
            cannibals_right = 3 - cannibals_left
            boat_location = "left" if boat_side == 0 else "right"
            print(f"Step {i}: {missionaries_left}M-{cannibals_left}C ({boat_location}) <-> {miss

if __name__ == "__main__":
    # Solve the Missionaries and Cannibals Problem using A* Search
    solution_path = a_star_search()
    print_solution_path(solution_path)
```

```
Solution found:
Step 0: 3M-3C (left) <-> 0M-0C
Step 1: 2M-2C (right) <-> 1M-1C
Step 2: 3M-2C (left) <-> 0M-1C
Step 3: 3M-0C (right) <-> 0M-3C
Step 4: 3M-1C (left) <-> 0M-2C
Step 5: 1M-1C (right) <-> 2M-2C
Step 6: 2M-2C (left) <-> 1M-1C
Step 7: 0M-2C (right) <-> 3M-1C
Step 8: 0M-3C (left) <-> 3M-0C
Step 9: -1M-2C (right) <-> 4M-1C
Step 10: 0M-2C (left) <-> 3M-1C
Step 11: 0M-0C (right) <-> 3M-3C
```

16.Water Jug Problem solving by using production system approach

In [27]:

```
def water_jug_problem_with_steps(jug_a_capacity, jug_b_capacity, target_volume):
    """ Solve the Water Jug Problem and display the steps using a production system approach """
    initial_state = (0, 0)  # Start with both jugs empty
    visited = set()  # To track visited states
    state_queue = [(initial_state, [])]  # Queue of (state, steps) tuples

    # Production rules for filling, emptying, and pouring water between jugs
    production_rules = [
        ("Fill Jug A", lambda x, y: (jug_a_capacity, y)),  # Fill Jug A
        ("Fill Jug B", lambda x, y: (x, jug_b_capacity)),  # Fill Jug B
        ("Empty Jug A", lambda x, y: (0, y)),  # Empty Jug A
        ("Empty Jug B", lambda x, y: (x, 0)),  # Empty Jug B
        ("Pour B to A", lambda x, y: (min(x + y, jug_a_capacity), max(0, x + y - jug_a_capacity)
        ("Pour A to B", lambda x, y: (max(0, x + y - jug_b_capacity), min(x + y, jug_b_capacity)
    ]

    # Explore states using a breadth-first search approach
    while state_queue:
        current_state, steps = state_queue.pop(0)  # Get the first (state, steps) tuple from the

        if current_state in visited:
            continue  # Skip visited states

        visited.add(current_state)  # Mark current state as visited

        # Check if the goal state is reached
        if current_state[0] == target_volume or current_state[1] == target_volume:
            return steps  # Return the sequence of steps (actions)

        # Apply each production rule to generate new states
        for action_name, rule in production_rules:
            new_state = rule(current_state[0], current_state[1])

            if new_state not in visited:
                new_steps = steps + [action_name]  # Append the current action to the sequence o
                state_queue.append((new_state, new_steps))  # Add new (state, steps) tuple to th

    return None  # No solution found

# Example usage:
jug_a_capacity = 4
jug_b_capacity = 3
target_volume = 2

steps = water_jug_problem_with_steps(jug_a_capacity, jug_b_capacity, target_volume)
if steps:
    print(f"Steps to measure {target_volume} liters using jugs {jug_a_capacity} and {jug_b_capac
    for i, step in enumerate(steps, 1):
        print(f"Step {i}: {step}")
else:
    print(f"Target volume of {target_volume} liters cannot be measured using the given jugs.")
```

```
Steps to measure 2 liters using jugs 4 and 3:
Step 1: Fill Jug B
Step 2: Pour B to A
Step 3: Fill Jug B
Step 4: Pour B to A
```

17.Tic Tac Toe game implementation by Magic Square Method

In [31]:
```
import random

def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 13)

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 ra
        return True

    return False

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

def get_user_move():
    while True:
        try:
            move = int(input("Enter your move (1-9): "))
            if 1 <= move <= 9:
                return move
            else:
                print("Invalid move. Please enter a number between 1 and 9.")
        except ValueError:
            print("Invalid input. Please enter a number.")

def calculate_computer_move(board, player_symbol, computer_symbol):
    magic_square = [
        [8, 3, 4],
        [1, 5, 9],
        [6, 7, 2]
    ]

    empty_cells = [(i, j) for i in range(3) for j in range(3) if board[i][j] == ' ']

    for i, j in empty_cells:
        temp_board = [row[:] for row in board]
        temp_board[i][j] = computer_symbol
        if is_winner(temp_board, computer_symbol):
            return i * 3 + j + 1

    for i, j in empty_cells:
        temp_board = [row[:] for row in board]
        temp_board[i][j] = player_symbol
        if is_winner(temp_board, player_symbol):
            return i * 3 + j + 1

    return random.choice(empty_cells)[0] * 3 + random.choice(empty_cells)[1] + 1

def play_tic_tac_toe():
    board = [[' ' for _ in range(3)] for _ in range(3)]
    user_symbol, computer_symbol = 'X', 'O'
    print("Welcome to Tic-Tac-Toe using Magic Square technique!")
    print_board(board)

    for move_num in range(1, 10):
        current_player = user_symbol if move_num % 2 == 1 else computer_symbol

        if current_player == user_symbol:
            user_move = get_user_move()
            row, col = divmod(user_move - 1, 3)
        else:
            computer_move = calculate_computer_move(board, user_symbol, computer_symbol)
            row, col = divmod(computer_move - 1, 3)
            print(f"Computer chooses position {computer_move}")
```

```
        while board[row][col] != ' ':
            print("ERROR! That position is already taken. Choose a different one.")
            if current_player == user_symbol:
                user_move = get_user_move()
                row, col = divmod(user_move - 1, 3)
            else:
                computer_move = calculate_computer_move(board, user_symbol, computer_symbol)
                row, col = divmod(computer_move - 1, 3)

        board[row][col] = user_symbol if current_player == user_symbol else computer_symbol
        print_board(board)

        if is_winner(board, current_player):
            print(f"{current_player} wins!")
            break

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


play_tic_tac_toe()
```

```
Welcome to Tic-Tac-Toe using Magic Square technique!
  |   |
------------
  |   |
------------
  |   |
------------
Enter your move (1-9): 4
  |   |
------------
X |   |
------------
  |   |
------------
Computer chooses position 5
  |   |
------------
X | O |
------------
  |   |
------------
Enter your move (1-9): 1
X |   |
------------
X | O |
------------
  |   |
------------
Computer chooses position 7
X |   |
------------
X | O |
------------
O |   |
------------
Enter your move (1-9): 3
X |   | X
------------
X | O |
------------
O |   |
------------
Computer chooses position 2
X | O | X
------------
X | O |
------------
O |   |
------------
Enter your move (1-9): 8
X | O | X
------------
X | O |
------------
O | X |
------------
Computer chooses position 6
X | O | X
------------
X | O | O
------------
O | X |
------------
Enter your move (1-9): 9
X | O | X
------------
X | O | O
------------
O | X | X
------------
It's a tie!
```

In [*]:

```
magic_square = [
    [2, 7, 6],
    [9, 5, 1],
    [4, 3, 8]
]
def check_win(board, player):
    for i in range(3):
        if sum(board[i][j] == player for j in range(3)) == 3:
            return True
        if sum(board[j][i] == player for j in range(3)) == 3:
            return True
    if sum(board[i][i] == player for i in range(3)) == 3:
        return True
    if sum(board[i][2 - i] == player for i in range(3)) == 3:
        return True
    return False
def print_board(board):
    for row in board:
        print("|".join(str(cell) for cell in row))
        print("-" * 5)
def main():
    board = [[0] * 3 for _ in range(3)]
    player = 1
    while True:
        print_board(board)
        print(f"Player {player}'s turn")
        i = int(input("Enter row (1-3): ")) - 1
        j = int(input("Enter column (1-3): ")) - 1
        if board[i][j] == 0:
            board[i][j] = player
            if check_win(board, player):
                print(f"Player {player} wins!")
                break
            if player == 1:
                player = 2
            else:
                player = 1
        else:
            print("Cell already occupied. Try again.")
if __name__ == "__main__":
    main()
```

```
0|0|0
-----
0|0|0
-----
0|0|0
-----
Player 1's turn
Enter row (1-3): 1
Enter column (1-3): 2
0|1|0
-----
0|0|0
-----
0|0|0
-----
Player 2's turn
Enter row (1-3): 2
Enter column (1-3): 2
0|1|0
-----
0|2|0
-----
0|0|0
-----
Player 1's turn
Enter row (1-3): 1
Enter column (1-3): 3
0|1|1
-----
0|2|0
-----
0|0|0
-----
Player 2's turn
Enter row (1-3): 1
Enter column (1-3): 1
2|1|1
-----
0|2|0
-----
0|0|0
-----
Player 1's turn
Enter row (1-3): 3
Enter column (1-3): 3
2|1|1
-----
0|2|0
-----
0|0|1
-----
Player 2's turn
Enter row (1-3): 2
Enter column (1-3): 3
2|1|1
-----
0|2|2
-----
0|0|1
-----
Player 1's turn
Enter row (1-3): 2
Enter column (1-3): 1
2|1|1
-----
1|2|2
-----
0|0|1
-----
Player 2's turn
Enter row (1-3): 3
Enter column (1-3): 2
2|1|1
-----
1|2|2
```

```
-----
0|2|1
-----
Player 1's turn
Enter row (1-3): 3
Enter column (1-3): 1
2|1|1
-----
1|2|2
-----
1|2|1
-----
Player 2's turn
Enter row (1-3): 1
Enter column (1-3): 1
Cell already occupied. Try again.
2|1|1
-----
1|2|2
-----
1|2|1
-----
Player 2's turn
Enter row (1-3):
```

18.Tic Tac Toe Problem solving by using Adversarial Search approach.

In [29]:

```
import math

class TicTacToe:
    def __init__(self):
        self.board = [' ' for _ in range(9)]  # Initialize an empty board
        self.current_player = 'X'  # 'X' starts first

    def display_board(self):
        print('-------------')
        for i in range(3):
            print(f'| {self.board[3*i]} | {self.board[3*i+1]} | {self.board[3*i+2]} |')
            print('-------------')

    def get_empty_cells(self):
        return [i for i, val in enumerate(self.board) if val == ' ']

    def is_winner(self, player):
        win_positions = [
            [0, 1, 2], [3, 4, 5], [6, 7, 8],  # Rows
            [0, 3, 6], [1, 4, 7], [2, 5, 8],  # Columns
            [0, 4, 8], [2, 4, 6]             # Diagonals
        ]
        for pos in win_positions:
            if all(self.board[i] == player for i in pos):
                return True
        return False

    def is_board_full(self):
        return all(val != ' ' for val in self.board)

    def make_move(self, index, player):
        self.board[index] = player

    def minimax(self, depth, is_maximizer):
        if self.is_winner('X'):
            return 1
        if self.is_winner('O'):
            return -1
        if self.is_board_full():
            return 0

        if is_maximizer:
            max_eval = -math.inf
            for cell in self.get_empty_cells():
                self.make_move(cell, 'X')
                eval = self.minimax(depth + 1, False)
                self.make_move(cell, ' ')
                max_eval = max(max_eval, eval)
            return max_eval
        else:
            min_eval = math.inf
            for cell in self.get_empty_cells():
                self.make_move(cell, 'O')
                eval = self.minimax(depth + 1, True)
                self.make_move(cell, ' ')
                min_eval = min(min_eval, eval)
            return min_eval

    def get_best_move(self):
        best_move = -1
        best_eval = -math.inf
        for cell in self.get_empty_cells():
            self.make_move(cell, 'X')
            eval = self.minimax(0, False)
            self.make_move(cell, ' ')
            if eval > best_eval:
                best_eval = eval
                best_move = cell
        return best_move

    def play_game(self):
        print("Welcome to Tic Tac Toe!")
        print("Enter a number from 0-8 to make your move.")

        while True:
```

```
            self.display_board()

            if self.current_player == 'X':  # Human player's turn
                while True:
                    try:
                        user_move = int(input("Your turn (X), enter position: "))
                        if user_move not in self.get_empty_cells():
                            print("Invalid move. Try again.")
                        else:
                            self.make_move(user_move, 'X')
                            break
                    except ValueError:
                        print("Invalid input. Please enter a number from 0-8.")
            else:  # AI player's turn
                ai_move = self.get_best_move()
                print(f"AI's turn (O), position: {ai_move}")
                self.make_move(ai_move, 'O')

            # Check game status
            if self.is_winner('X'):
                self.display_board()
                print("Congratulations! You win!")
                break
            elif self.is_winner('O'):
                self.display_board()
                print("AI wins. Better luck next time!")
                break
            elif self.is_board_full():
                self.display_board()
                print("It's a tie!")
                break

            # Switch turns
            self.current_player = 'O' if self.current_player == 'X' else 'X'

if __name__ == "__main__":
    game = TicTacToe()
    game.play_game()
```

```
Welcome to Tic Tac Toe!
Enter a number from 0-8 to make your move.
-------------
|   |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
Your turn (X), enter position: 4
-------------
|   |   |   |
-------------
|   | X |   |
-------------
|   |   |   |
-------------
AI's turn (O), position: 0
-------------
| O |   |   |
-------------
|   | X |   |
-------------
|   |   |   |
-------------
Your turn (X), enter position: 1
-------------
| O | X |   |
-------------
|   | X |   |
-------------
|   |   |   |
-------------
AI's turn (O), position: 2
-------------
| O | X | O |
-------------
|   | X |   |
-------------
|   |   |   |
-------------
Your turn (X), enter position: 7
-------------
| O | X | O |
-------------
|   | X |   |
-------------
|   | X |   |
-------------
Congratulations! You win!
```