EXPERIMENT-2

Name: Adwait Purao
UID: 2021300101
Division: B
Batch: B

Problem Definition:

Implement the missionary and cannibal problem using the Uninformed searching technique DFS and analyze the algorithms with respect to Completeness, Optimality, time, and space complexity.

Theory:

The Missionary and Cannibal Problem is a classic puzzle in artificial intelligence and graph theory. In this challenge, three missionaries and three cannibals must cross a river using a boat that accommodates at most two people. The goal is to find a sequence of boat trips ensuring the safety of all individuals by following specific rules:

Rule 1: The number of cannibals on either side of the river should not exceed the number of missionaries, or the cannibals will eat the missionaries.

Rule 2: At least one person must be on the boat to steer it.

Depth-First Search (DFS) is an uninformed searching algorithm utilized to solve this problem. It operates as follows:

Initialization: Start with an initial state, such as having all missionaries and cannibals on one side of the river, with the boat on the other side.

Successor States: Generate possible successor states by moving one or two people (missionaries or cannibals) from one side to the other.

Validation: Apply the problem rules to check if each successor state is valid, ensuring it doesn't violate the rule of having more cannibals than missionaries on any side.

Exploration: If a valid successor state is found, add it to the stack and continue the search.

Backtracking: If there are no valid successors or a goal state (all missionaries and cannibals on the other side) is reached, backtrack by popping states from the stack until a new valid successor is found.

Repeat: Repeat steps 2-5 until a solution is found or all possible states are explored.

Analysis:

Completeness: DFS is not guaranteed to be complete in all scenarios. It can potentially get stuck in infinite loops if there is no solution or if the search space is too vast. However, for problems like the Missionary and Cannibal Problem with a finite search space, DFS is complete. Given a finite space, DFS will eventually find a solution if one exists.

• **Optimality:** DFS does not ensure finding the optimal solution. It might discover a solution that requires more steps (boat trips) than the shortest possible path. To guarantee optimality, modifications to DFS or the

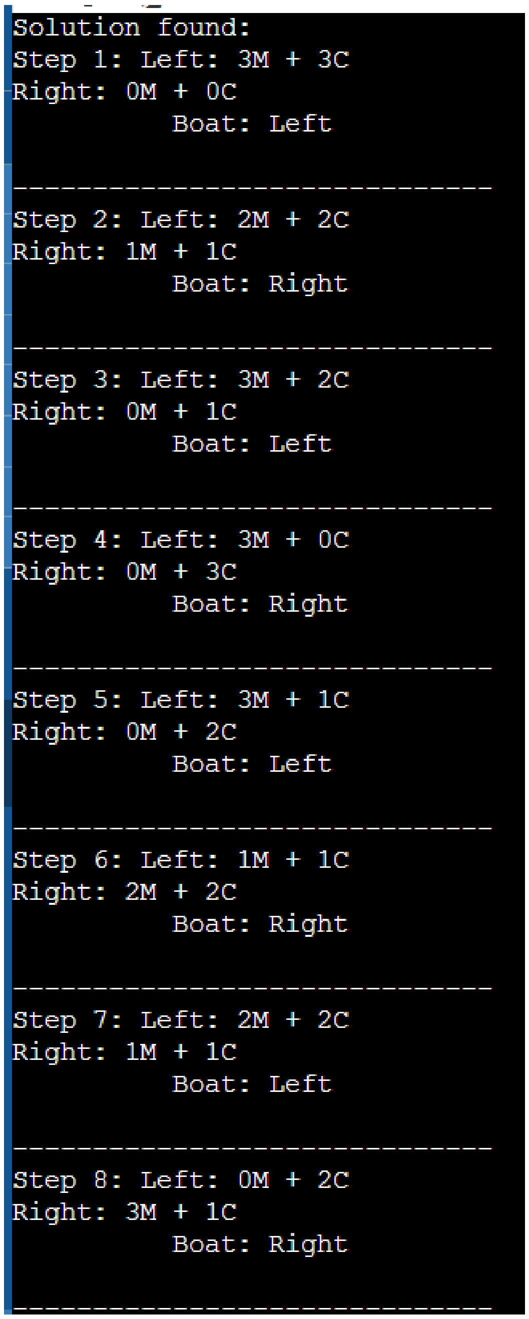
use of algorithms like Breadth-First Search (BFS) would be necessary.
• **Time Complexity:** In the worst case, DFS can have exponential time complexity. It explores all possible states in a depth-first manner. The time complexity is denoted as O(b^d), where "b" represents the branching factor (the number of possible actions at each state), and "d" represents the depth of the shallowest goal state. In the Missionary and Cannibal Problem, both "b" and "d" can vary based on the specific problem instance.
• **Space Complexity:** The space complexity of DFS relies on the maximum depth of the search tree. In the worst case, if the search tree is exceptionally deep, the space complexity can be O(d), where "d" represents the maximum depth of the search tree.
In summary, DFS is a straightforward and intuitive algorithm for solving the Missionary and Cannibal Problem. However, it doesn't always guarantee the optimal solution, and its time complexity can become exponential in scenarios where the search space is extensive.

Code:

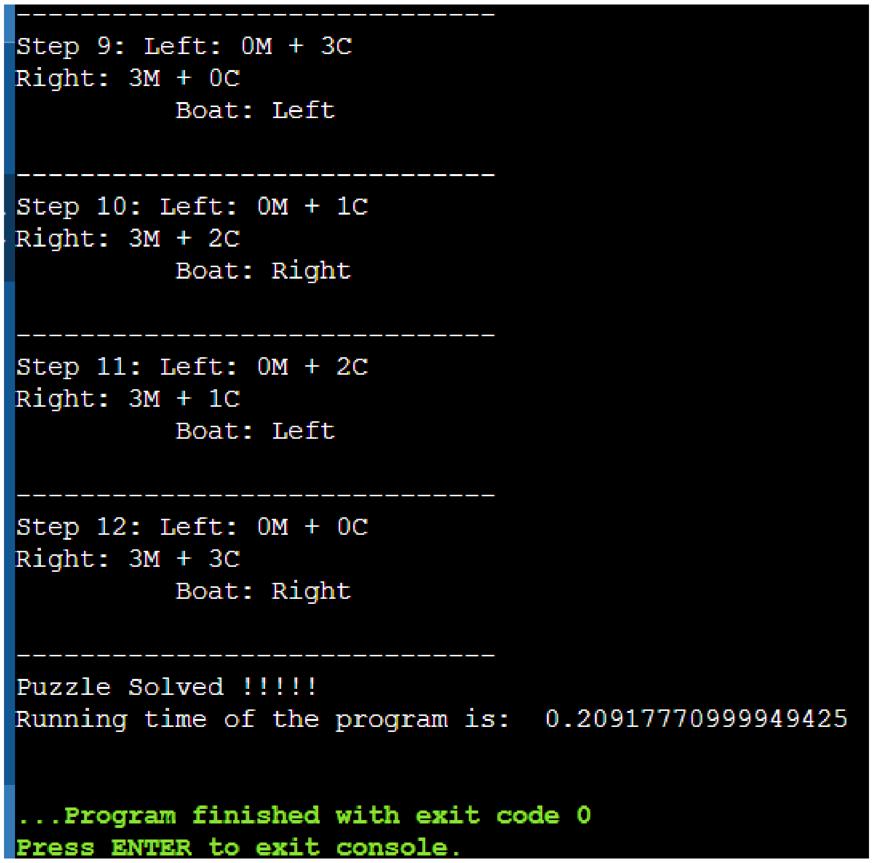
```
import timeit

class State:
    def __init__(self, missionaries_left, cannibals_left, boat, missionaries_right,
cannibals_right, parent=None):
        self.missionaries_left, self.cannibals_left, self.boat = missionaries_left,
cannibals_left, boat
        self.missionaries_right, self.cannibals_right, self.parent =
missionaries_right, cannibals_right, parent

    def is_valid(self):
        return (
            0 <= self.missionaries_left <= 3 and 0 <= self.cannibals_left <= 3
            and 0 <= self.missionaries_right <= 3 and 0 <= self.cannibals_right <=
3
            and (self.missionaries_left == 0 or self.missionaries_left >=
self.cannibals_left)
            and (self.missionaries_right == 0 or self.missionaries_right >=
self.cannibals_right)
        )

    def is_goal(self):
        return self.missionaries_left == 0 and self.cannibals_left == 0

    def __eq__(self, other):
        return (
            self.missionaries_left == other.missionaries_left
            and self.cannibals_left == other.cannibals_left
            and self.boat == other.boat
            and self.missionaries_right == other.missionaries_right
            and self.cannibals_right == other.cannibals_right
        )

    def __hash__(self):
        return hash((
            self.missionaries_left, self.cannibals_left, self.boat,
            self.missionaries_right, self.cannibals_right
        ))

def get_successors(current_state):
    successors = []
    moves = [(2, 0), (0, 2), (1, 1), (1, 0), (0, 1)]
    opposite = {"Left": "Right", "Right": "Left"}
    for m, c in moves:
        if current_state.boat == "Left":
            new_state = State(
                current_state.missionaries_left - m,
                current_state.cannibals_left - c,
                opposite[current_state.boat],
                current_state.missionaries_right + m,
                current_state.cannibals_right + c,
                parent=current_state,
            )
        else:
            new_state = State(
                current_state.missionaries_left + m,
                current_state.cannibals_left + c,
                opposite[current_state.boat],
                current_state.missionaries_right - m,
                current_state.cannibals_right - c,
                parent=current_state,
            )
        if new_state.is_valid():
            successors.append(new_state)
    return successors

def dfs(initial_state):
    visited = set()
    stack = [initial_state]
    while stack:
        current_state = stack.pop()
        if current_state.is_goal():
            path = []
            while current_state:
                path.append(current_state)
                current_state = current_state.parent
            path.reverse()
```

```python
            return path
        visited.add(current_state)
        successors = get_successors(current_state)
        stack.extend(s for s in successors if s not in visited)
    return None

def print_solution(path):
    for i, state in enumerate(path):
        print(f"Step {i + 1}:", end=' ')
        print(f"Left: {state.missionaries_left}M + {state.cannibals_left}C")
        print(f"Right: {state.missionaries_right}M + {state.cannibals_right}C")
        print(f"          Boat: {state.boat}\n")
        print("------------------------------")

if __name__ == "__main__":
    initial_state = State(3, 3, "Left", 0, 0)
    running_time = timeit.timeit("dfs(initial_state)", globals=globals(),
number=1000)
    solution = dfs(initial_state)
    if solution:
        print("Solution found:")
        print_solution(solution)
        print("Puzzle Solved !!!!!")
        print("Running time of the program is: ", running_time)
    else:
        print("No solution found.")
```

Output:



```
Solution found:
Step 1: Left: 3M + 3C
Right: 0M + 0C
          Boat: Left

------------------------------
Step 2: Left: 2M + 2C
Right: 1M + 1C
          Boat: Right

------------------------------
Step 3: Left: 3M + 2C
Right: 0M + 1C
          Boat: Left

------------------------------
Step 4: Left: 3M + 0C
Right: 0M + 3C
          Boat: Right

------------------------------
Step 5: Left: 3M + 1C
Right: 0M + 2C
          Boat: Left

------------------------------
Step 6: Left: 1M + 1C
Right: 2M + 2C
          Boat: Right

------------------------------
Step 7: Left: 2M + 2C
Right: 1M + 1C
          Boat: Left

------------------------------
Step 8: Left: 0M + 2C
Right: 3M + 1C
          Boat: Right

------------------------------
```



```
------------------------------
Step 9: Left: 0M + 3C
Right: 3M + 0C
          Boat: Left

------------------------------
Step 10: Left: 0M + 1C
Right: 3M + 2C
          Boat: Right

------------------------------
Step 11: Left: 0M + 2C
Right: 3M + 1C
          Boat: Left

------------------------------
Step 12: Left: 0M + 0C
Right: 3M + 3C
          Boat: Right

------------------------------
Puzzle Solved !!!!!
Running time of the program is:  0.20917770999949425


...Program finished with exit code 0
Press ENTER to exit console.
```

Running Time: 0.2091 seconds

Time Complexity: O(b^d)

Space Complexity: O(d)

Completeness: For this puzzle, DFS is complete.

Conclusion:

In this experiment, we successfully applied the Depth-First Search (DFS) algorithm in Python to solve the Missionary and Cannibal Problem. While DFS proved to be a complete approach, it may not always yield the most optimal solution due to its depth-first nature. Additionally, its time complexity can be exponential in worst-case scenarios, making it essential to assess problem characteristics. This experiment highlights the need for considering alternative search algorithms to address efficiency and optimality concerns in problem-solving. Overall, it showcases the versatility and challenges of using AI algorithms to tackle complex puzzles like the Missionary and Cannibal Problem.