EXPERIMENT-2

Name: Adwait Purao
UID: 2021300101
Div: B
Batch: B2

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:

1) **Missionaries and Cannibal Problem:** In the missionaries and cannibals' problem, three missionaries and three cannibals must cross a river using a boat which can carry at most two people, under the constraint that, for both banks, if there are missionaries present on the bank, they cannot be outnumbered by cannibals (if they were, the cannibals would eat the missionaries). The boat cannot cross the river by itself with no people on board.

   Missionary Cannibal Problem using Uninformed searching technique DFS:

   a. Start at the initial state and add it to the stack. This is the starting point for the search.

   b. While the stack is not empty:

      i. Pop the current state from the stack. This is the state that we will explore next.

      ii. If the current state is the goal state, then return the path from the initial state to the current state. We have found a solution!

      iii. Otherwise, generate all possible successor states of the current state. These are the states that we can reach from the current state.

      iv. Add all possible successor states to the stack. This ensures that we will eventually explore all possible paths from the initial state to the goal state.

   c. If the stack is empty, then no solution exists. We have explored all possible paths from the initial state, and none of them led to the goal state.

Code:

1. **Missionaries and Cannibal Problem:**

```
import time

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)]
```

```python
    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()
            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):
```

```python
    for i, state in enumerate(path):
        print(f"Step {i + 1} :")
        print(f"Left\t=>\t{state.missionaries_left}M\t{state.cannibals_left}C")
        print(f"Right\t=>\t{state.missionaries_right}M\t{state.cannibals_right}C")
        print(f"Boat\t=>\t{state.boat}\n")
        print("*****************************\n")

if __name__ == "__main__":
    initial_state = State(3, 3, "left", 0, 0)
    start=time.time()
    solution = dfs(initial_state)
    end=time.time()
    if solution:
        print("\nSolution found!")
        print("*****************************\n")
        print_solution(solution)
    else:
        print("No solution found.")
    print("The time of execution of above program is :", (end-start) * 10**3, "ms")
```

Output:

1. **Missionaries and Cannibal Problem:**

```
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 :
```

```
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

****************************

The time of execution of above program is : 0.12803077697753906 ms
```

Analysis of the Code:

- **Completeness:** A complete algorithm is one that is guaranteed to find a solution if one exists. **DFS is complete** because it explores all possible paths from the initial state to the goal state.
- **Optimality:** An optimal algorithm is one that finds the best solution among all possible solutions. **DFS is optimal** because it explores all possible paths from the initial state to the goal state and returns the first solution it finds.
- **Time complexity:** The time complexity of the DFS algorithm is **O(b^m)**, where b is the branching factor of the search tree and m is the maximum depth of the search tree. In the missionary and cannibal problem, the branching factor is 5 and the maximum depth of the search tree is 11. Therefore, the time complexity of the DFS algorithm for the missionary and cannibal problem is O(5^11).
- **Space complexity:** The space complexity of the DFS algorithm is **O(b^m)**, where b is the branching factor of the search tree and m is the maximum depth of the search tree. In the missionary and cannibal problem, the branching factor is 5 and the maximum depth of the search tree is 11. Therefore, the space complexity of the DFS algorithm for the missionary and cannibal problem is O(5^11).

Conclusion:

The missionary and cannibal problem can be solved using the depth-first search (DFS) algorithm. DFS is a complete and optimal algorithm, but it can be slow and memory-intensive for problems with large state spaces. To improve performance, heuristics, limited depth search, or parallel DFS can be used.