Name: Satyam Jaiswal **UID:** 2021600028 **Batch**: B **Date:** 29/08/2023

EXP 2: Uninformed Search Strategy

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.

The missionaries and cannibals problem is usually stated as follows. Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place.

1- Implement it with simple If else

a. Formulate the problem precisely, making only those distinctions necessary to ensure a

valid solution. Draw a diagram of the complete state space.

b. Implement and solve the problem optimally using an appropriate search algorithm.

d. Analyze the performance against Time, Space complexity. Show the intermediate status of OPEN and Close

Program:

1. **DFS**

```
#include <iostream>

#include <vector>

#include <algorithm>


using namespace std;


struct State {

   int lc;

   int lm;

   int rc;

   int rm;

   int boat;
```

```
    bool operator==(const State& other) const {
        return lc == other.lc && lm == other.lm && rc == other.rc && rm == other.rm && boat == other.boat;
    }

};

bool isValid(State& state) {
    if ((state.lc >= 0 && state.rc >= 0 && state.lm >= 0 && state.rm >= 0 && state.rc <= 3 && state.lc <= 3 && state.rm <= 3 && state.lm <= 3) && (state.rm==0 || state.rm >= state.rc) && (state.lm==0 || state.lm >= state.lc)) {
        return true;
    }
    else {
        return false;
    }
}

bool isGoal(State& state) {
    if (state.lc == 0 && state.lm == 0 && state.rc == 3 && state.rm == 3 && state.boat == 1) {
        return true;
    }
    else {
        return false;
    }
}

void dfs(State& state, vector<State>& visited, vector<State>& path, int& open, int& closed) {
    if (isGoal(state)) {
        cout << "Goal State Reached" << endl;
        cout << "Path to the goal state:" << endl;
        for (const State& step : path) {
```

```
        cout << "Left Bank: " <<step.lm<< "M "<<step.lc<<"C "<<(step.boat==0 ?
"Boat":"\t")<<"\t\t\t\t";
        cout <<"Right Bank: " <<step.rm<<"M "<<step.rc<<"C "<<(step.boat==1 ? "Boat ":"")
<<endl;
      }
      cout << "Left Bank: " <<state.lm<< "M "<<state.lc<<"C "<<(state.boat==0 ?
"Boat":"\t")<<"\t\t\t\t";
      cout <<"Right Bank: " <<state.rm<<"M "<<state.rc<<"C "<<(state.boat==1 ? "Boat ":"")
<<endl;
      return;
   }
   if (isValid(state)) {
      if (find(visited.begin(), visited.end(), state) == visited.end()) {
         visited.push_back(state);
         path.push_back(state);
         open++;
         cout<<"Open: "<<open<<" "<<"Closed: "<<closed<<endl;
         State newState;
         if (state.boat == 1) {
            for (int i = 0; i <= 2; i++) {
               for (int j = 0; j <= 2; j++) {
                  if (i + j <= 2 && i + j > 0) {
                     newState = { state.lc + i, state.lm + j, state.rc - i, state.rm - j, 0 };
                     newState.boat = 0;
                     dfs(newState, visited, path, open, closed);
                  }
               }
            }
         }
         else {
            for (int i = 0; i <= 2; i++) {
               for (int j = 0; j <= 2; j++) {
```

```
                if (i + j <= 2 && i + j > 0) {
                    newState = { state.lc - i, state.lm - j, state.rc + i, state.rm + j, 1 };
                    newState.boat = 1;
                    dfs(newState, visited, path, open, closed);
                }
            }
        }
        path.pop_back();
        closed++;
        cout<<"Open: "<<open<<" "<<"Closed: "<<closed<<endl;
    }
    else {
        return;
    }
  }
}

int main() {
  #ifndef ONLINE_JUDGE
  freopen("input.txt", "r", stdin);
  freopen("output.txt", "w", stdout);
  #endif
  struct State initial = { 3, 3, 0, 0, 0 };
  vector<State> visited;
  vector<State> path;
  int open = 0;
  int closed = 0;
  dfs(initial, visited, path, open, closed);
}
```

2. BFS

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <unordered_set>

using namespace std;

struct State {
    int lc;
    int lm;
    int rc;
    int rm;
    int boat;

    bool operator==(const State& other) const {
        return lc == other.lc && lm == other.lm && rc == other.rc && rm == other.rm && boat == other.boat;
    }
};

bool isValid(State& state) {
    if ((state.lc >= 0 && state.rc >= 0 && state.lm >= 0 && state.rm >= 0 && state.rc <= 3 && state.lc <= 3 && state.rm <= 3 && state.lm <= 3) && (state.rm==0 || state.rm >= state.rc) && (state.lm==0 || state.lm >= state.lc)) {
        return true;
    }
    else {
        return false;
    }
}
```

```
bool isGoal(State& state) {
    if (state.lc == 0 && state.lm == 0 && state.rc == 3 && state.rm == 3 && state.boat == 1) {
        return true;
    }
    else {
        return false;
    }
}

bool bfs(State& initial, int& open, int& closed) {
    queue<pair<State, vector<State>>> bfsQueue;
    vector<State> visited;

    vector<State> initialPath = {initial};
    bfsQueue.push({initial, initialPath});
    visited.push_back(initial);

    while (!bfsQueue.empty()) {
        pair<State, vector<State>> currentState = bfsQueue.front();
        State curr = currentState.first;
        vector<State> path = currentState.second;
        bfsQueue.pop();
        open--;
        closed++;
        if (isGoal(curr)) {
            cout << "Solution Found" << endl;
            cout << "Path to the goal state:" << endl;
            for (const State& step : path) {
                cout << "Left Bank: " << step.lm << "M " << step.lc << "C " << (step.boat == 0 ?
"Boat" : "\t") << "\t\t\t\t";
```

```
            cout << "Right Bank: " << step.rm << "M " << step.rc << "C " << (step.boat == 1 ?
"Boat" : "") << endl;
        }
        return true;
    }

    vector<State> children;

    if (curr.boat == 0) {
        for (int i = 0; i <= 2; i++) {
            for (int j = 0; j <= 2; j++) {
                if (i + j <= 2 && i + j > 0) {
                    State newState = {curr.lc - i, curr.lm - j, curr.rc + i, curr.rm + j, 1};
                    if (find(visited.begin(), visited.end(), newState) == visited.end() &&
isValid(newState)) {
                        children.push_back(newState);
                        visited.push_back(newState);
                        open++;
                    }
                }
            }
        }
    } else {
        for (int i = 0; i <= 2; i++) {
            for (int j = 0; j <= 2; j++) {
                if (i + j <= 2 && i + j > 0) {
                    State newState = {curr.lc + i, curr.lm + j, curr.rc - i, curr.rm - j, 0};
                    if (find(visited.begin(), visited.end(), newState) == visited.end() &&
isValid(newState)) {
                        children.push_back(newState);
                        visited.push_back(newState);
                        open++;
```

```
                    }
                }
            }
        }
    }

    for (State& child : children) {
        vector<State> newPath = path;
        newPath.push_back(child);
        bfsQueue.push({child, newPath});
    }
  }

  cout << "No solution found" << endl;
  return false;
}

int main() {
  #ifndef ONLINE_JUDGE
  freopen("input.txt", "r", stdin);
  freopen("output.txt", "w", stdout);
  #endif
  struct State initial = { 3, 3, 0, 0, 0 };
  vector<State> visited;
  vector<State> path;
  int open = 1;
  int closed = 0;
  bfs(initial, open, closed);
  cout<<"Open: "<<open<<" "<<"Closed: "<<closed<<endl;
}
```

Output:

1. **DFS**

Open: 1 Closed: 0

Open: 2 Closed: 0

Open: 2 Closed: 1

Open: 3 Closed: 1

Open: 4 Closed: 1

Open: 5 Closed: 1

Open: 5 Closed: 2

Open: 6 Closed: 2

Open: 7 Closed: 2

Open: 8 Closed: 2

Open: 9 Closed: 2

Open: 10 Closed: 2

Open: 11 Closed: 2

Open: 12 Closed: 2

Open: 13 Closed: 2

Goal State Reached

Path to the goal state:

Left Bank: 3M 3C Boat	Right Bank: 0M 0C
Left Bank: 2M 2C	Right Bank: 1M 1C Boat
Left Bank: 3M 2C Boat	Right Bank: 0M 1C
Left Bank: 3M 0C	Right Bank: 0M 3C Boat
Left Bank: 3M 1C Boat	Right Bank: 0M 2C
Left Bank: 1M 1C	Right Bank: 2M 2C Boat
Left Bank: 2M 2C Boat	Right Bank: 1M 1C
Left Bank: 0M 2C	Right Bank: 3M 1C Boat
Left Bank: 0M 3C Boat	Right Bank: 3M 0C
Left Bank: 0M 1C	Right Bank: 3M 2C Boat
Left Bank: 1M 1C Boat	Right Bank: 2M 2C
Left Bank: 0M 0C	Right Bank: 3M 3C Boat

Open: 13 Closed: 3

Open: 14 Closed: 3

Goal State Reached

Path to the goal state:

Left Bank: 3M 3C Boat

Left Bank: 2M 2C

Left Bank: 3M 2C Boat

Left Bank: 3M 0C

Left Bank: 3M 1C Boat

Left Bank: 1M 1C

Left Bank: 2M 2C Boat

Left Bank: 0M 2C

Left Bank: 0M 3C Boat

Left Bank: 0M 1C

Left Bank: 0M 2C Boat

Left Bank: 0M 0C

Right Bank: 0M 0C

Right Bank: 1M 1C Boat

Right Bank: 0M 1C

Right Bank: 0M 3C Boat

Right Bank: 0M 2C

Right Bank: 2M 2C Boat

Right Bank: 1M 1C

Right Bank: 3M 1C Boat

Right Bank: 3M 0C

Right Bank: 3M 2C Boat

Right Bank: 3M 1C

Right Bank: 3M 3C Boat

Open: 14 Closed: 4

Open: 14 Closed: 5

Open: 14 Closed: 6

Open: 14 Closed: 7

Open: 14 Closed: 8

Open: 14 Closed: 9

Open: 14 Closed: 10

Open: 14 Closed: 11

Open: 14 Closed: 12

Open: 14 Closed: 13

Open: 14 Closed: 14

2. **DFS**

Solution Found

Path to the goal state:

Left Bank: 3M 3C Boat	Right Bank: 0M 0C
Left Bank: 2M 2C	Right Bank: 1M 1C Boat
Left Bank: 3M 2C Boat	Right Bank: 0M 1C
Left Bank: 3M 0C	Right Bank: 0M 3C Boat
Left Bank: 3M 1C Boat	Right Bank: 0M 2C
Left Bank: 1M 1C	Right Bank: 2M 2C Boat
Left Bank: 2M 2C Boat	Right Bank: 1M 1C
Left Bank: 0M 2C	Right Bank: 3M 1C Boat
Left Bank: 0M 3C Boat	Right Bank: 3M 0C
Left Bank: 0M 1C	Right Bank: 3M 2C Boat
Left Bank: 1M 1C Boat	Right Bank: 2M 2C
Left Bank: 0M 0C	Right Bank: 3M 3C Boat

Open: 0 Closed: 15

Time & Space Complexity:

1. **DFS**

Time Complexity:

Let b be the maximum branching factor (the maximum number of child states generated from each state).

Let d be the depth of the search tree (the number of steps required to reach the goal state).

In the worst case, DFS explores all possible states up to depth d before finding a solution or concluding that no solution exists.

The time complexity of DFS can be expressed as: **O(b^d)**.

Space Complexity:

The space complexity of DFS is determined by the maximum depth of the search tree.

The maximum depth of the search tree can be expressed as: O(d).

Therefore, the space complexity of DFS can be expressed as: **O(d)**.

2. **BFS**

Time Complexity:

Let b be the maximum branching factor (the maximum number of child states generated from each state).

Let d be the depth of the search tree (the number of steps required to reach the goal state).

In the BFS approach, we explore all nodes at depth d before moving to nodes at depth d+1.

The time complexity of BFS can be expressed as: **O(b^d).**

Space Complexity:

In BFS, we need to maintain a queue to store nodes at each level of the search tree.

The maximum number of nodes that can be in the queue at any point corresponds to the number of nodes at the deepest level of the tree.

The maximum depth of the search tree can be expressed as: O(d).

Therefore, the space complexity of BFS can be expressed as: **O(b^d).**

Conclusion: 1) Learnt how to implement the missionary cannibal problem using both depth first search (DFS) and breadth first search (BFS).

2) Also learnt about the effects of bfs and dfs on the time and space complexity.