BIRZEIT UNIVERSITY

**Faculty of Engineering and Technology**

**Computer Science Department**

COMP338 – ARTIFICIAL INTELLIGENCE



| | Students Name | Students Number |
|---|---|---|
| 1 | Yousef Sharbi | 1202057 |
| 2 | Anas karakra | 1200467 |



**Project 1: Solving Problems by Searching (Missionaries and Cannibals Problem)**

## What are BFS and DFS?

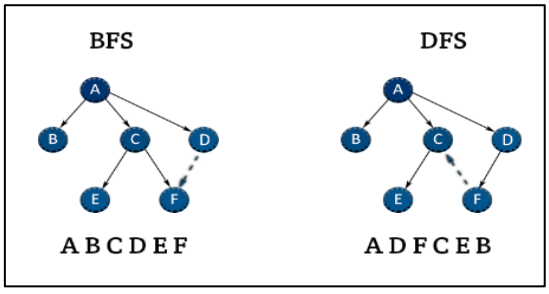
- **Breadth-First Search (BFS):**

  It's an algorithm that explores a graph level by level, starting from the root node and visiting all its neighbors before moving on to the next level. It ensures that all nodes at a given depth are visited before proceeding to deeper levels, and that make it useful for finding the shortest path in unweighted graphs. BFS search horizontally before vertical using queue (FIFO) structure.

- **Depth-First Search (DFS):**

  It's an algorithm that traverses a graph by exploring as far as possible along each branch before backtracking. It starts from the root node, visits a neighbor, and continues this process until it reaches the end of a branch before backtracking to explore other branches. DFS, in contrast to BFS, searches vertically before horizontal exploration, employing a stack (LIFO) structure for systematic traversal.

  Photo example:



## What are Missionaries and Cannibals problem?

Missionaries and Cannibals problem is very famous in Artificial Intelligence. The problem stated as follow: Three missionaries and three cannibals must cross a river using a boat which can carry at most two people, but the missionaries present on the either side of the bank cannot be outnumbered by cannibals. The boat cannot cross the river by itself with no people on board. And this problem can be solved by a graph search method using BFS and DFS.

## Problem Formulation (Missionaries and Cannibals):

- **States:** the number of missionaries and cannibals on each side of the river, as well as the boat's passengers and location.

- **Initial State:**
  - standard initial state: (3,3,1,0,0,0)
  - our initial state `(3, 3, new Boat("Empty", "Empty", "Left"), 0, 0)`

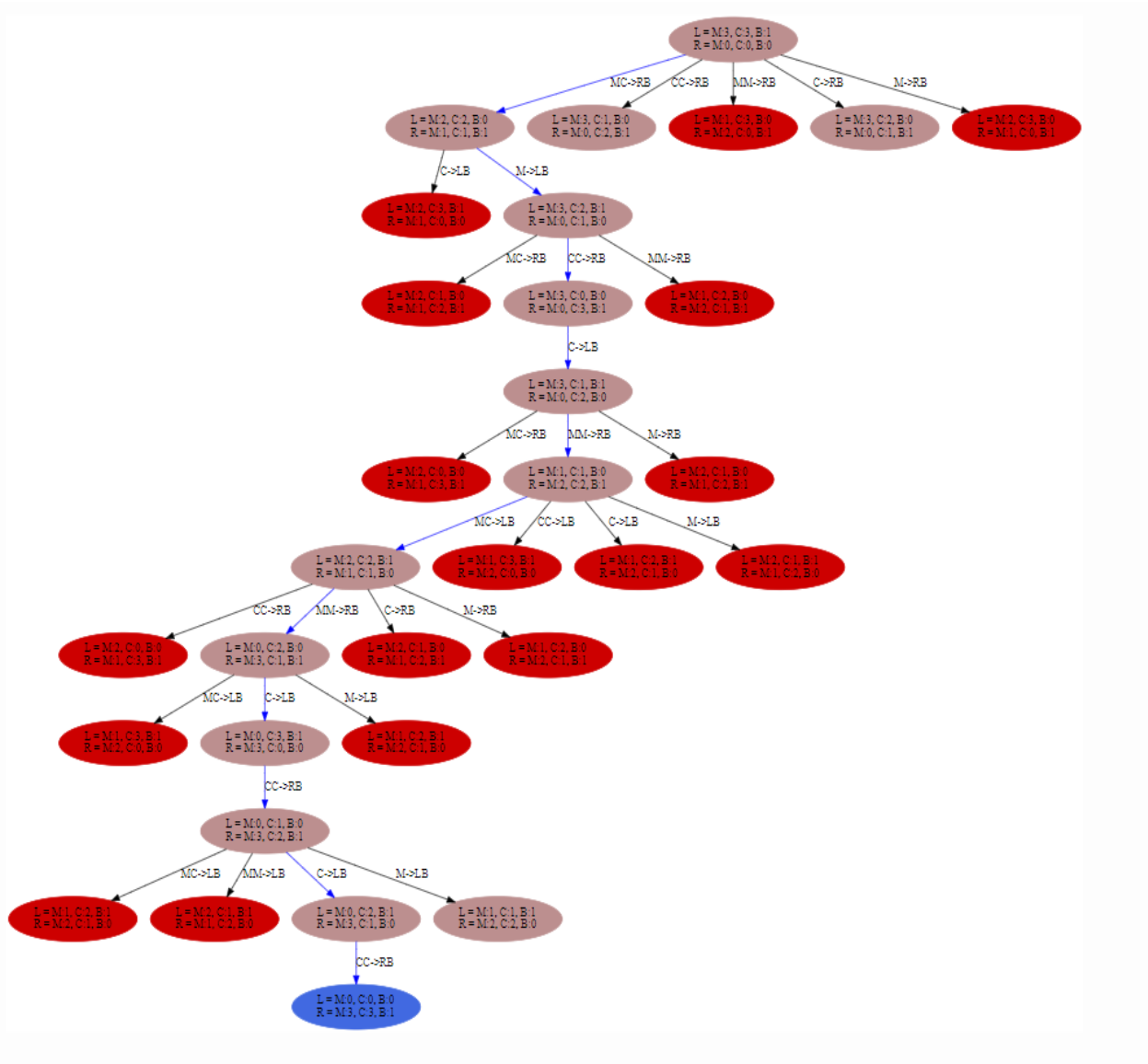
- **Successor Function:**
  - Possible actions involve moving missionaries and cannibals from one side to the other, and the boat as well.

- **Goal Test:**
  - have all missionaries and cannibals on the right side of the river.
  - (0,0, (“Empty”,”Empty”,”Right”),3,3)

- **Path Cost:**
  - each step costs 1.

- **Solution:**
  - sequence of actions leading from the initial state to a goal.
  - `[3, 3, (Empty, Empty, Left), 0, 0]` →`[0, 0, (Cannibal, Cannibal, Right), 3, 3]`

## Drawing the complete search space (State Space):



# The steps for optimal sol:

- **State (State.java):**

```
//=== Attributes ------------------------------------------------------------
private int left_cannibal;
private int left_missionary;
private Boat boat;
private int right_cannibal;
private int right_missionary;
private State prevState;


//=== Constructor -----------------------------------------------------------
public State(int left_cannibal, int left_missionary, Boat boat, int right_cannibal, int right_missionary) {
    this.left_cannibal = left_cannibal;
    this.left_missionary = left_missionary;
    this.boat = boat;
    this.right_cannibal = right_cannibal;
    this.right_missionary = right_missionary;
}
```

- 
```
//=== Getter and Setter -----------------------------------------------------
```

- **Method to check if the case valid to state:**

```
// check if the case valid to state
public boolean isValidCase() {
    if((left_cannibal>=0 && left_missionary>=0 && right_cannibal>=0 && right_missionary>=0) &&
       (left_missionary>=left_cannibal || left_missionary==0)&&
       (right_missionary>=right_cannibal) || right_missionary==0) {
        return true;
    }
    else {
        return false;
    }
}
```

- **Method to list all valid cases:**

```
// valid cases list for status
public ArrayList<State> validCases(){
    ArrayList<State> listValidCases=new ArrayList<>();

    // Cases:


    // Cases when boat left position:

    // case 1: (one cannibal) go from left to right.
    State case1_boat_left=new State(left_cannibal-1 , left_missionary ,new Boat("Cannibal", "Empty", "Right") , right_cannibal+1 , right_missionary);

    // case 2: (one missionary) go from left to right.
    State case2_boat_left=new State(left_cannibal , left_missionary-1 ,new Boat("Missionary", "Empty", "Right") , right_cannibal , right_missionary+1);

    // case 3: (one missionary and one cannibal) go from left to right.
    State case3_boat_left=new State(left_cannibal-1 , left_missionary-1 ,new Boat("Cannibal", "Missionary", "Right") , right_cannibal+1 , right_missionary+1);

    // case 4: (two cannibals) go from left to right.
    State case4_boat_left=new State(left_cannibal-2 , left_missionary ,new Boat("Cannibal", "Cannibal", "Right") , right_cannibal+2 , right_missionary);

    // case 5: (two missionaries) go from left to right.
    State case5_boat_left=new State(left_cannibal , left_missionary-2 ,new Boat("Missionary", "Missionary", "Right") , right_cannibal , right_missionary+2);


    // array of state to store possible cases that when boat on left position.
    State listCases_boatLeft[]= {case1_boat_left,case2_boat_left,case3_boat_left,case4_boat_left,case5_boat_left};

    /* if the boat had left position then --> list all left position cases
     * then check if they are valid to that state and if they, then set this state(parent) the previous state for valid state(child).
     * finally store that valid state(child) on listValidCases.
     */
```

- 
```
    if(boat.isPositionLeft()) {
        for(int i=0;i<listCases_boatLeft.length;i++) {
            if(listCases_boatLeft[i].isValidCase()) {
                listCases_boatLeft[i].setPrevState(this);
                listValidCases.add(listCases_boatLeft[i]);
            }
        }
    }

    //-------------------------------------------------------------------------------------------------------------------------

    // Cases when boat right position:

    // case 1: (one cannibal) go from right to left.
    State case1_boat_right=new State(left_cannibal+1 , left_missionary ,new Boat("Cannibal", "Empty", "Left") , right_cannibal-1 , right_missionary);

    // case 2: (one missionary) go from right to left.
    State case2_boat_right=new State(left_cannibal , left_missionary+1 ,new Boat("Missionary", "Empty", "Left") , right_cannibal , right_missionary-1);

    // case 3: (one missionary and one cannibal) go from right to left.
    State case3_boat_right=new State(left_cannibal+1 , left_missionary+1 ,new Boat("Missionary", "Cannibal", "Left") , right_cannibal-1 , right_missionary-1);

    // case 4: (two cannibals) go from right to left.
    State case4_boat_right=new State(left_cannibal+2 , left_missionary ,new Boat("Cannibal", "Cannibal", "Left") , right_cannibal-2 , right_missionary);

    // case 5: (two missionaries) go from right to left.
    State case5_boat_right=new State(left_cannibal , left_missionary+2 ,new Boat("Missionary", "Missionary", "Left") , right_cannibal , right_missionary-2);


    // array of state to store possible cases that when boat on right position.
    State listCases_boatRight[]= {case1_boat_right,case2_boat_right,case3_boat_right,case4_boat_right,case5_boat_right};

    /* if the boat had right position then --> list all right position cases
     * then check if they are valid to that state and if they, then set this state(parent) the previous state for valid state(child).
     * finally store that valid state(child) on listValidCases.
     */
```

- 
```
/* if the boat had right position then --> list all right position cases
 * then check if they are valid to that state and if they, then set this state(parent) the previous state for valid state(child).
 * finally store that valid state(child) on listValidCases.
*/
if(boat.isPositionRight()) {
    for(int i=0;i<listCases_boatRight.length;i++) {
        if(listCases_boatRight[i].isValidCase()) {
            listCases_boatRight[i].setPrevState(this);
            listValidCases.add(listCases_boatRight[i]);
        }
    }
}
```

- 
```
return listValidCases;
```

- **For BFS (BFS.java):**
  - **We have queue list(queue) and visited list (array List)**
    - ```
      Queue<State> queue_list=new Queue<>();
      ArrayList<State> visited_list=new ArrayList<>();
      ```
  - At first enqueue the state inside queue list.
    - if the queue empty then there is no solution.
    - Else dequeue state from queue list and store it in variable s to add it to visited list.
  - Then store all valid cases to that state on listValidCases.
  - check if this valid case not on visited list and not on queue list
  - if it achieve that cannibal and missionary are all cross the river then it success, else it will enqueue the valid cases till the search success or fail.

```
// At first enqueue the state inside queue list.
queue_list.enqueue(state);
for(;;) {
    if(queue_list.isEmpty()) { // if the queue empty then there is no solution.
        return null;
    }
    else {
        State s=queue_list.dequeue(); // dequeue state from queue list and store it in variable s to add it to visited list.

        visited_list.add(s);
        ArrayList<State> listValidCases=s.validCases(); // store all valid cases to that state on listValidCases.

        for(int i=0;i<listValidCases.size();i++) {
            // check if this valid case not on visited list and not on queue list
            if( (!visited_list.contains(listValidCases.get(i))) || (!queue_list.isContains(listValidCases.get(i))) ) {

                /* if it achieve that cannibal and missionary are all cross the river then it success, else
                 * it will enqueue the valid cases till the search success or fail.
                */
                if(listValidCases.get(i).getLeft_cannibal()==0 && listValidCases.get(i).getLeft_missionary()==0) {
                    return listValidCases.get(i);
                }
                else {
                    queue_list.enqueue(listValidCases.get(i));
                }
            }
        }
    }
}
```
```

- **For DFS (DFS.java):**
  - **We have stack list(stack) and visited list (array List)**
    - ```
      Stack<State> stack_list = new Stack<>();
      ArrayList<State> visited_list = new ArrayList<>();
      ```
  - At first push the state inside stack list.
  - pop state from stack list and store it in variable s to add it to visited list.
  - store all valid cases to that state on listValidCases.
  - check if this valid case not on visited list and not on stack list
  - if it achieve that cannibal and missionary are all cross the river then it success, else it will push the valid cases till the search success or fail.
  - if the stack empty then there is no solution.

```
// At first push the state inside stack list.
stack_list.push(state);
while (!stack_list.isEmpty()) {
    State s = stack_list.pop(); // pop state from stack list and store it in variable s to add it to visited list.

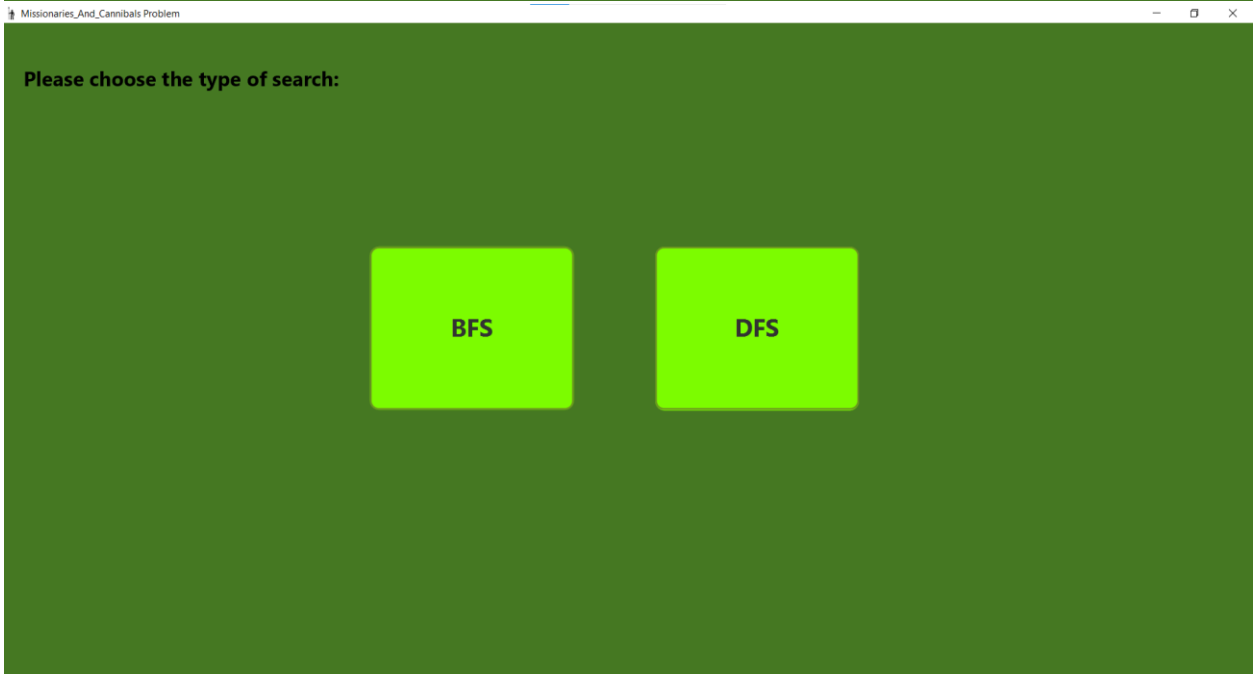
    visited_list.add(s);
    ArrayList<State> listValidCases = s.validCases(); // store all valid cases to that state on listValidCases.

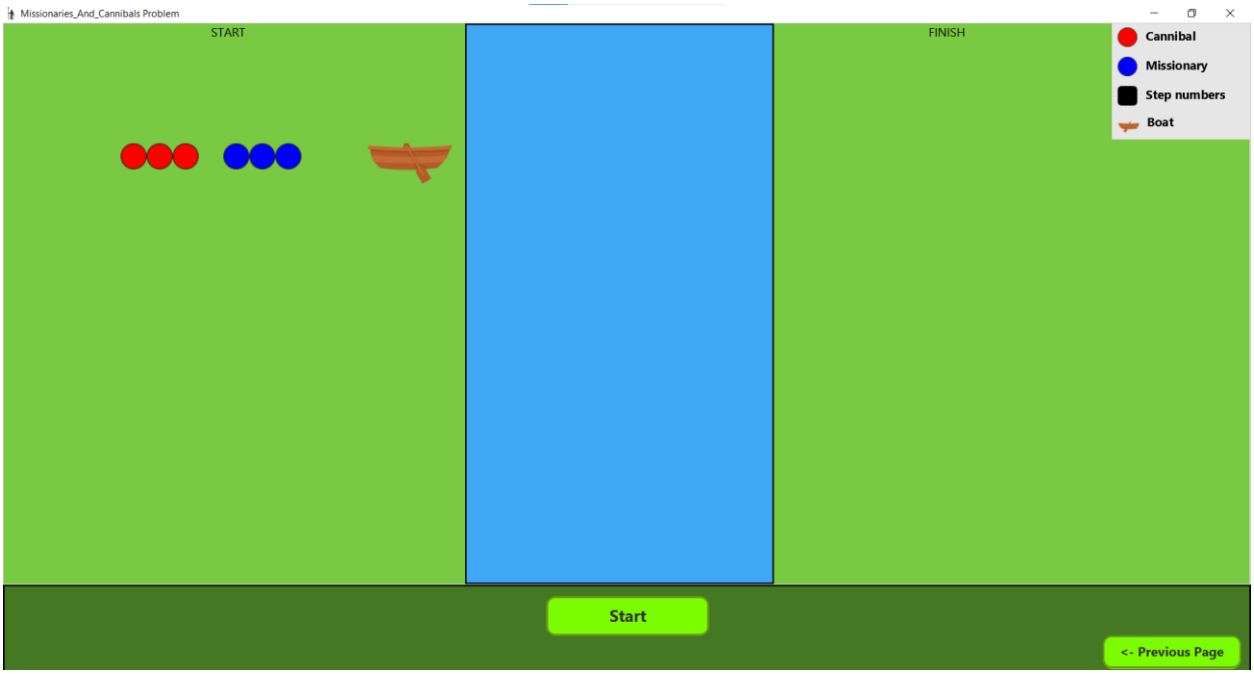
    for(int i=0;i<listValidCases.size();i++) {
        // check if this valid case not on visited list and not on stack list
        if (!visited_list.contains(listValidCases.get(i)) && !stack_list.isContains(listValidCases.get(i))) {

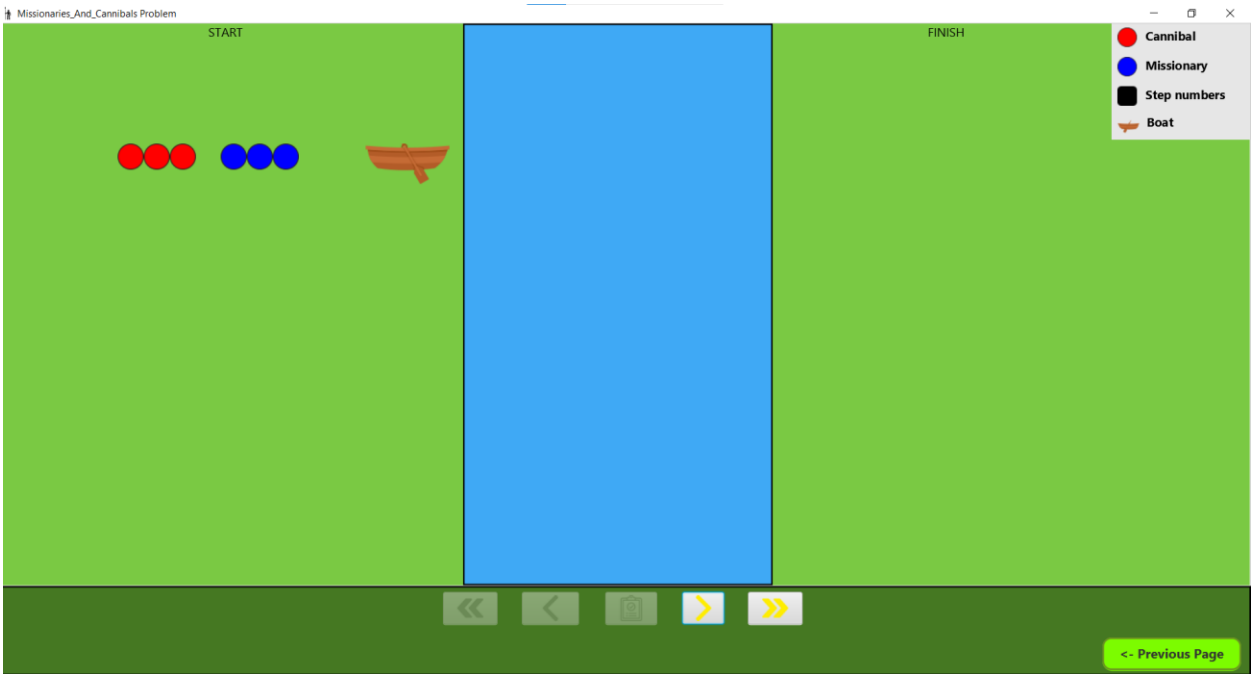
            /* if it achieve that cannibal and missionary are all cross the river then it success, else
             * it will push the valid cases till the search success or fail.
            */
            if(listValidCases.get(i).getLeft_cannibal()==0 && listValidCases.get(i).getLeft_missionary()==0) {
                return listValidCases.get(i);
            }
            else {
                stack_list.push(listValidCases.get(i));
            }
        }
    }
}

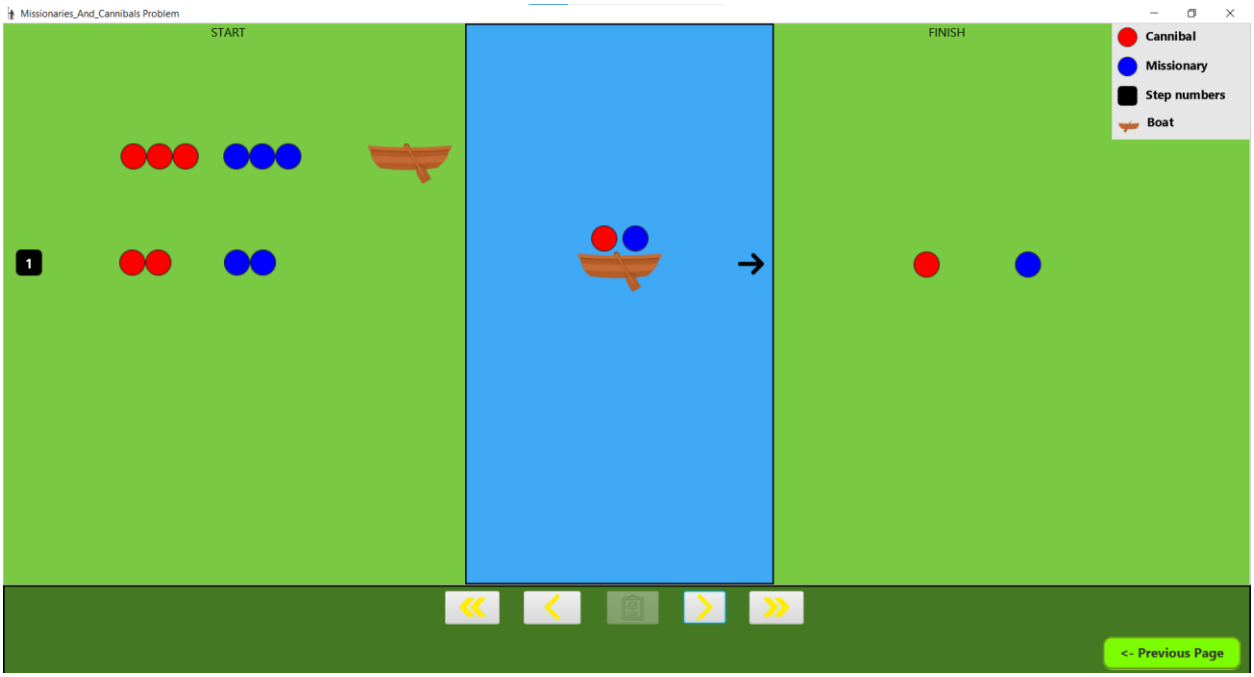

// if the stack empty then there is no solution.
return null;
```
```

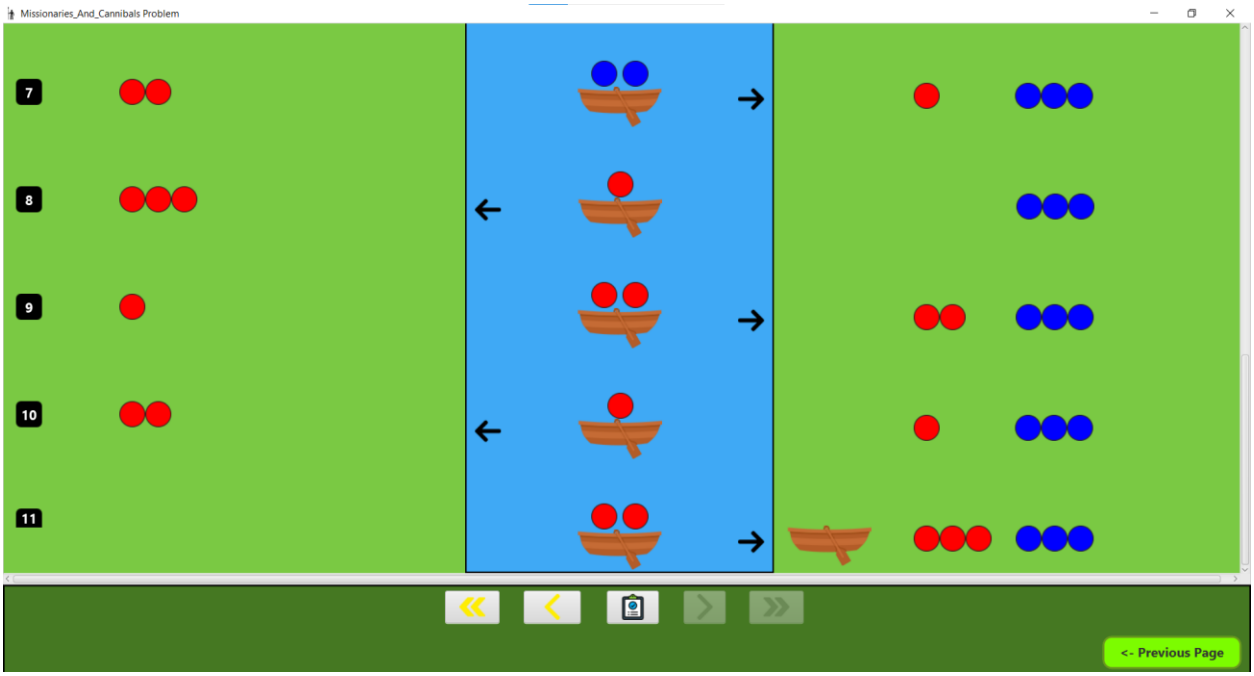
## JavaFX(UI):

- 
Missionaries_And_Cannibals Problem
MISSIONARIES
AND
CANNIBALS
Let's GO
- 
Missionaries_And_Cannibals Problem
Please choose the type of search:
BFS
DFS

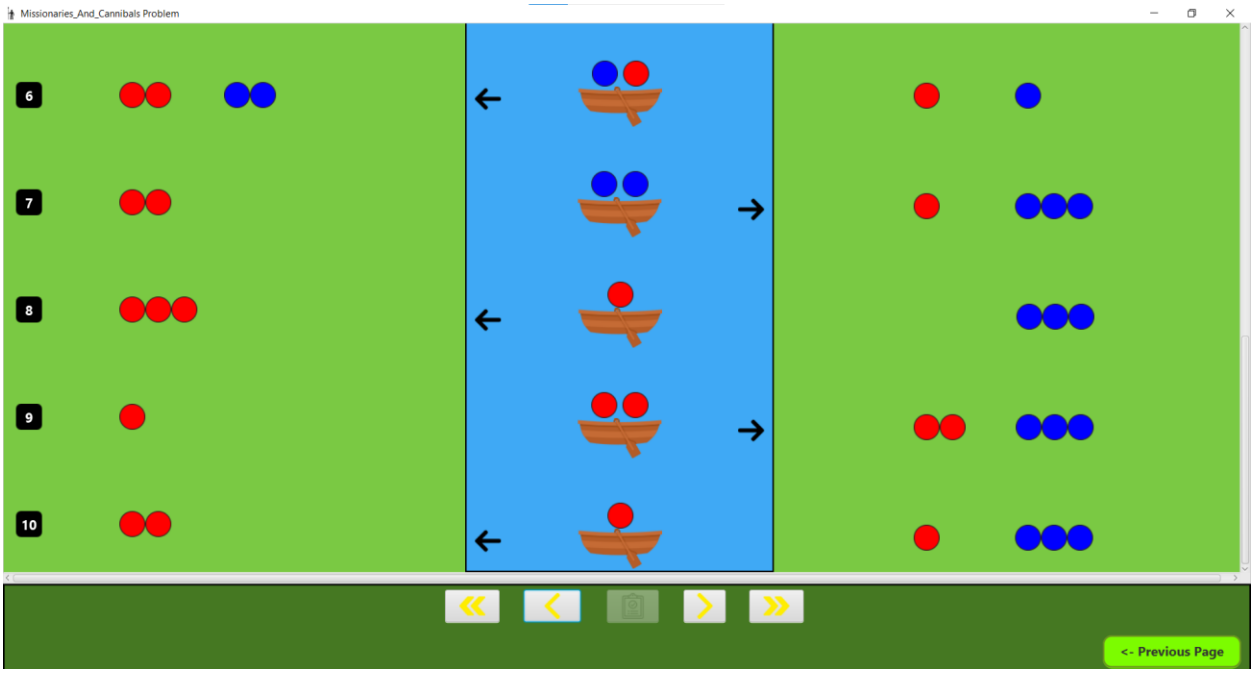

Missionaries_And_Cannibals Problem
START
FINISH
Cannibal
Missionary
Step numbers
Boat
Start
<- Previous Page


Missionaries_And_Cannibals Problem
START
FINISH
Cannibal
Missionary
Step numbers
Boat
<- Previous Page


Missionaries_And_Cannibals Problem
START
FINISH
Cannibal
Missionary
Step numbers
Boat
1
<- Previous Page


Missionaries_And_Cannibals Problem
7
8
9
10
11
<- Previous Page


Missionaries_And_Cannibals Problem
Show Result
Initial State : [3, 3, (Empty, Empty, Left), 0, 0]
V
Step number (1): [2, 2, (Cannibal, Missionary, Right), 1, 1]
V
Step number (2): [2, 3, (Missionary, Empty, Left), 1, 0]
V
Step number (3): [0, 3, (Cannibal, Cannibal, Right), 3, 0]
V
Step number (4): [1, 3, (Cannibal, Empty, Left), 2, 0]
V
Step number (5): [1, 1, (Missionary, Missionary, Right), 2, 2]
V
Step number (6): [2, 2, (Missionary, Cannibal, Left), 1, 1]
V
Step number (7): [2, 0, (Missionary, Missionary, Right), 1, 3]
V
Step number (8): [3, 0, (Cannibal, Empty, Left), 0, 3]
V
Step number (9): [1, 0, (Cannibal, Cannibal, Right), 2, 3]
7
8
9
10
11
Show Result
<- Previous Page


Missionaries_And_Cannibals Problem
6
7
8
9
10
<- Previous Page