Missionaries and Cannibals

Name: Sumaiya Kashmin Zim

Student ID: 201814016

Section: A

Course Code: CSE-404

Date of submission: 5 April, 2021

Introduction: The missionaries and cannibals problem is a well-known problem in artificial intelligence. The problem can be 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. The above problem can be solved by a graph search method.

Methodology: I built the problem "Missionaries and Cannibals" using programming language Java and used the console window to show the output. The steps for solving the problem was to represent object State, implementing the search algorithm, generating the successors and to finalize the program and print the path.

I considered the missionaries, cannibals and the boat are initially on the left bank of the river, and they have to go to the right bank. So, the state look like this:

S = (CannibalLeft, MissionaryLeft, CannibalRight, MissionaryRight, BoatPosition)

Thus, Start State: (3, 3, 0, 0, LEFT)

Final State: (0, 0, 3,3, Right)

Then, I tried finding the cases where the states will not be valid and possible actions that can occur during the various states in the problem. Then used a graph search algorithm (BFS) to find the path to reach our expected node in fewest steps.

Before I jumped to the code section, I tried solving this using state space diagram. The diagram is shown below:

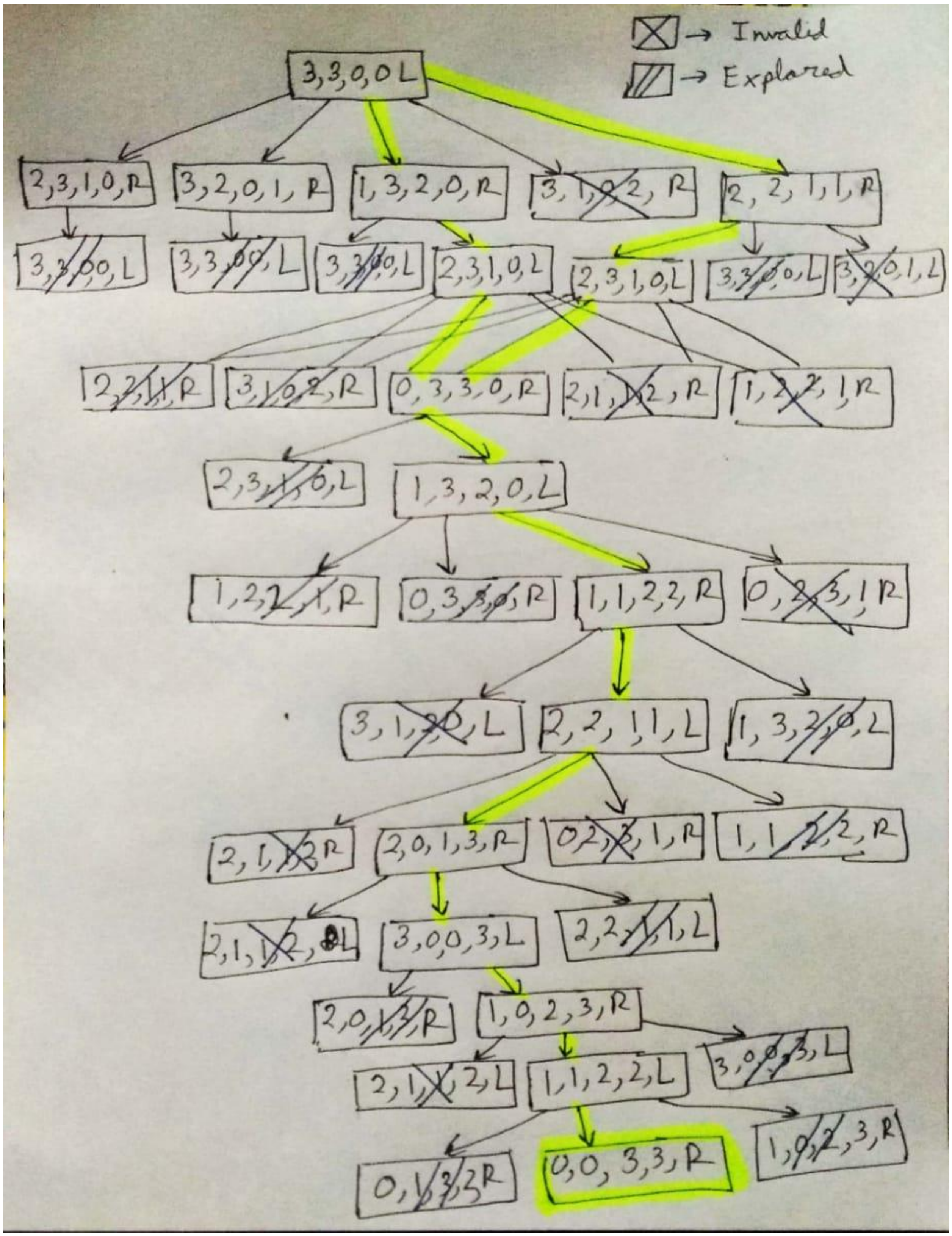


fig: state space diagram (the highlighted part is the path to reach the expected final node)

Code:

```
package assignment1;
import java.util.*;

public class Driver {
        public static void main(String[] args){
                state initial_State = new state (3, 3, 0, 0, Position.LEFT);
                bfs search = new bfs(initial_State);
                state result = search.bfs_execution();
                      List<state> path = new ArrayList<state>();
                      state state_ = result;
                      //System.out.print(state_.toString());
                while(null!=state_) {
                             path.add(state_);
                             state_ = state_.getParentState();
                      }

                      System.out.println("Missionaries and Cannibals Problem:");
                   System.out.println("(Cannibal Left, Missionary Left, Cannibal
Right, Missionary Right, Boat position)");

                      int depth = path.size() - 1;
                      for (int i = depth; i >= 0; i--) {
                             state_ = path.get(i);
                             if (state_.Final()) {
                                    System.out.print(state_.toString());
                             } else {
                                    System.out.println(state_.toString() + " -> ");
                             }
                      }
            }
}
```

Fig: Driver.java

```
package assignment1;
import java.util.*;

enum Position {RIGHT, LEFT}
public class state {
	private int cannibalLeft, cannibalRight, missionaryLeft, missionaryRight;
	private Position boat;
	private state parentState = null;

	public state(int cannibalLeft, int missionaryLeft, int cannibalRight, int 
missionaryRight,  Position boat) {
		this.cannibalLeft = cannibalLeft;
		this.missionaryLeft = missionaryLeft;
		this.cannibalRight = cannibalRight;
		this.missionaryRight = missionaryRight;
		this.boat = boat;
	}

	public boolean invalid() {
		if (missionaryLeft < 0 || missionaryRight < 0 || cannibalLeft < 0 || 
cannibalRight < 0 || missionaryLeft + missionaryRight > 3 || cannibalLeft + 
cannibalRight > 3
		 || (missionaryLeft < cannibalLeft && missionaryLeft != 0) || 
(missionaryRight < cannibalRight && missionaryRight != 0))
		{
			return true;
		}
		return false;
	}

	public state getParentState() {
		return parentState;
	}

	private void Add_state(List<state> successors, state newState) {
		if (!newState.invalid()){
			newState.parentState=this;
			successors.add(newState);
		}
	}
	public List<state> generateSuccessors() {
		List<state> successors = new ArrayList<state>();
		if (boat == Position.LEFT)
		{
			// One missionary and one cannibal cross left to right
			Add_state(successors, new state(cannibalLeft - 1, 
missionaryLeft - 1, cannibalRight + 1, missionaryRight + 1, Position.RIGHT));
			// One missionary crosses left to right
			Add_state(successors, new state(cannibalLeft, missionaryLeft - 
1, cannibalRight, missionaryRight + 1, Position.RIGHT));
			// One cannibal crosses left to right
```

```
                    Add_state(successors, new state(cannibalLeft - 1,
missionaryLeft, cannibalRight + 1, missionaryRight, Position.RIGHT));
                    // Two missionaries cross left to right
                    Add_state(successors, new state(cannibalLeft, missionaryLeft -
2, cannibalRight, missionaryRight + 2, Position.RIGHT));
                    // Two cannibals cross left to right
                    Add_state(successors, new state(cannibalLeft - 2,
missionaryLeft, cannibalRight + 2, missionaryRight, Position.RIGHT));

             } else
             {
                    // One missionary and one cannibal cross right to left
                    Add_state(successors, new state(cannibalLeft + 1,
missionaryLeft + 1, cannibalRight - 1, missionaryRight - 1, Position.LEFT));
                    // One missionary crosses right to left
                    Add_state(successors, new state(cannibalLeft, missionaryLeft +
1, cannibalRight, missionaryRight - 1, Position.LEFT));
                    // One cannibal crosses right to left
                    Add_state(successors, new state(cannibalLeft + 1,
missionaryLeft, cannibalRight - 1, missionaryRight, Position.LEFT));
                    // Two missionaries cross right to left
                    Add_state(successors, new state(cannibalLeft, missionaryLeft +
2, cannibalRight, missionaryRight - 2, Position.LEFT));
                    // Two cannibals cross right to left.
                    Add_state(successors, new state(cannibalLeft + 2,
missionaryLeft, cannibalRight - 2, missionaryRight, Position.LEFT));

             }
             return successors;
      }
      public boolean Final() {
             return cannibalLeft == 0 && missionaryLeft == 0 && cannibalRight == 3
&& missionaryRight == 3;
      }

      public String toString() {
             if (boat == Position.LEFT) {
                    return "(" + cannibalLeft + "," + missionaryLeft + ","+
cannibalRight + "," + missionaryRight +  ",Left)";
             } else {
                    return "(" + cannibalLeft + "," + missionaryLeft+ ","+
cannibalRight + "," + missionaryRight +  ",Right)";
             }
     }

}
```

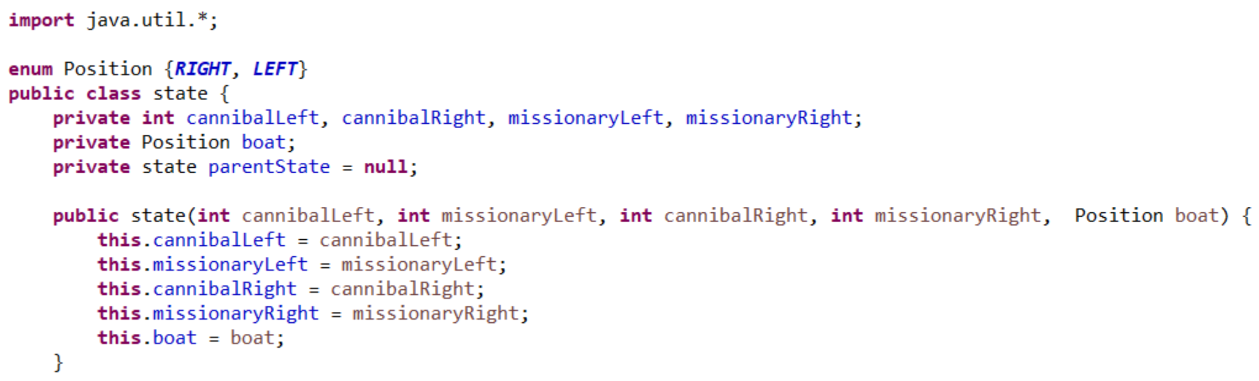
Fig: state.java

```
package assignment1;
import java.util.*;

public class bfs {
       state initialState;
       Queue<state> nodes;
       Set<state> explored;
       public bfs(state initialState_){
              initialState = initialState_;
              nodes = new LinkedList<state>();
              explored = new HashSet<state>();
       }
       public state bfs_execution() {
                     if (initialState.Final()) {
                            return initialState;
                     }
                     else {
              nodes.add(initialState);
              while (true) {
                     if (nodes.isEmpty()) {
                            return null;
                     }
                     state state1 = nodes.poll();
                     explored.add(state1);
                     List<state> successors = state1.generateSuccessors();
                     for (state child : successors) {
                                   if (child.Final()) {
                                          return child;
                                   }
                                   if (!explored.contains(child)||
!nodes.contains(child)){
                                   nodes.add(child);}
                            }
                     }
              }
       }
}
```

Fig: bfs.java

Explanation of Code:

State Class design: The state class is designed to store the current state of how many missionaries and cannibals are in each side of the river and the position of the boat. To find the boat position, I have declared an enum named ‘position’ which is a list of constants {Right, Left}. The arguments of states are consecutively cannibalLeft, missionaryLeft, cannibalRight, missionaryRight, Position boat. A parentState is also declared to find the parent of a child from the tree. Initially parent state is set to null.



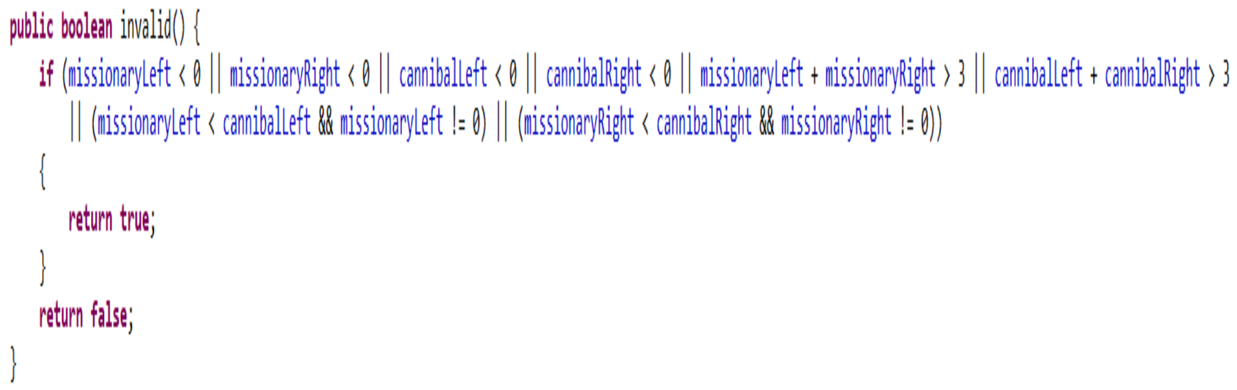
```
import java.util.*;

enum Position {RIGHT, LEFT}
public class state {
    private int cannibalLeft, cannibalRight, missionaryLeft, missionaryRight;
    private Position boat;
    private state parentState = null;

    public state(int cannibalLeft, int missionaryLeft, int cannibalRight, int missionaryRight,  Position boat) {
        this.cannibalLeft = cannibalLeft;
        this.missionaryLeft = missionaryLeft;
        this.cannibalRight = cannibalRight;
        this.missionaryRight = missionaryRight;
        this.boat = boat;
    }
```

fig: State Class design

Illegal State Handling: I found 8 illegal states:

1. Number of missionaries on the left side is negative
2. Number of missionaries on the right side is negative
3. Number of cannibals on the left side is negative
4. Number of cannibals on the right side is negative
5. Total number of missionaries on the left and right side is more than 3
6. Total number of cannibals on the left and right side is more than 3
7. Number of missionaries on the left side is less than number of cannibals on the same side unless number of missionaries is 0
8. Number of missionaries on the right side is less than number of cannibals on the same side unless number of missionaries is 0

So, inside of the boolean invalid function, I put all these above states inside an if condition. If any state satisfies any of these conditions, it will return true. Otherwise, it will return false.



```java
public boolean invalid() {
    if (missionaryLeft < 0 || missionaryRight < 0 || cannibalLeft < 0 || cannibalRight < 0 || missionaryLeft + missionaryRight > 3 || cannibalLeft + cannibalRight > 3
        || (missionaryLeft < cannibalLeft && missionaryLeft != 0) || (missionaryRight < cannibalRight && missionaryRight != 0))
    {
        return true;
    }
    return false;
}
```

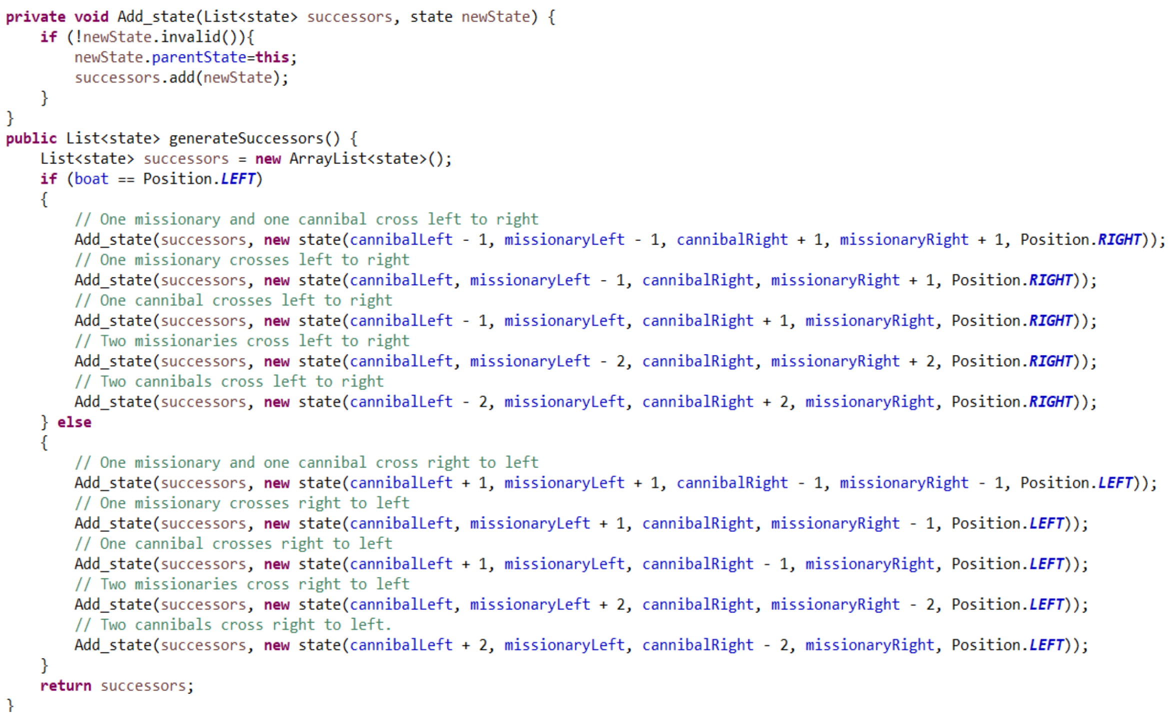
fig: Illegal state handling

Next State Generation: Next state will be generated after crossing the river by boat. As the boat can take highest 2 individuals, the way of crossing the boats are:

- **One missionary and one cannibal cross left to right:** In this case,
    1. number of missionary and cannibal on the left will be reduced by one.
    2. number of missionary and cannibal on the right will be increased by one.
    3. Boat position will be changed to right.
- **One missionary crosses left to right:** In this case,
    1. number of missionary on the left will be reduced by one.
    2. number of missionary on the right will be increased by one.
    3. Boat position will be changed to right.
- **One cannibal crosses left to right:** In this case,
    1. number of cannibals on the left will be reduced by one.
    2. number of cannibals on the right will be increased by one.
    3. Boat position will be changed to right.
- **Two missionaries cross left to right:** In this case,
    1. number of missionary on the left will be reduced by two.
    2. number of missionary on the right will be increased by two.
    3. Boat position will be changed to right.
- **Two cannibals cross left to right:** In this case,
    1. number of cannibals on the left will be reduced by two.
    2. number of cannibals on the right will be increased by two.
    3. Boat position will be changed to right.

- **One missionary and one cannibal cross right to left:** In this case,
    1. number of missionary and cannibal on the right will be reduced by one.
    2. number of missionary and cannibal on the left will be increased by one.
    3. Boat position will be changed to left.
- **One missionary crosses right to left:** In this case,
    1. number of missionary on the right will be reduced by one.
    2. number of missionary on the left will be increased by one.
    3. Boat position will be changed to left.
- **One cannibal crosses right to left:** In this case,
    1. number of cannibal on the left will be reduced by one.
    2. number of cannibal on the right will be increased by one.
    3. Boat position will be changed to left.
- **Two missionaries cross right to left:** In this case,
    1. number of missionaries on the right will be reduced by two.
    2. number of missionaries on the left will be increased by two.
    3. Boat position will be changed to left.
- **Two cannibals cross right to left:** In this case,
    1. number of cannibals on the right will be reduced by two.
    2. number of cannibals on the left will be increased by two.
    3. Boat position will be changed to left.

All of these possibilities are generated through generateSuccessors() function. Before generating next state like this, I checked if these states are actually valid or not. This checking was done by using the invalid() function I described in the previous section and adding was done by a function Add_State(). I have also set this state to parentState of the next generated state.

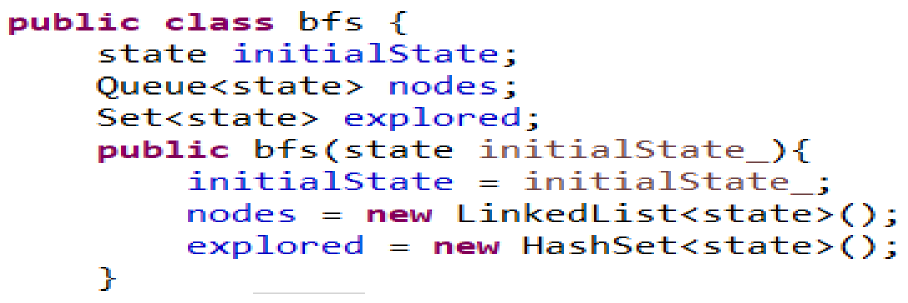


```
private void Add_state(List<state> successors, state newState) {
    if (!newState.invalid()){
        newState.parentState=this;
        successors.add(newState);
    }
}
public List<state> generateSuccessors() {
    List<state> successors = new ArrayList<state>();
    if (boat == Position.LEFT)
    {
        // One missionary and one cannibal cross left to right
        Add_state(successors, new state(cannibalLeft - 1, missionaryLeft - 1, cannibalRight + 1, missionaryRight + 1, Position.RIGHT));
        // One missionary crosses left to right
        Add_state(successors, new state(cannibalLeft, missionaryLeft - 1, cannibalRight, missionaryRight + 1, Position.RIGHT));
        // One cannibal crosses left to right
        Add_state(successors, new state(cannibalLeft - 1, missionaryLeft, cannibalRight + 1, missionaryRight, Position.RIGHT));
        // Two missionaries cross left to right
        Add_state(successors, new state(cannibalLeft, missionaryLeft - 2, cannibalRight, missionaryRight + 2, Position.RIGHT));
        // Two cannibals cross left to right
        Add_state(successors, new state(cannibalLeft - 2, missionaryLeft, cannibalRight + 2, missionaryRight, Position.RIGHT));
    } else
    {
        // One missionary and one cannibal cross right to left
        Add_state(successors, new state(cannibalLeft + 1, missionaryLeft + 1, cannibalRight - 1, missionaryRight - 1, Position.LEFT));
        // One missionary crosses right to left
        Add_state(successors, new state(cannibalLeft, missionaryLeft + 1, cannibalRight, missionaryRight - 1, Position.LEFT));
        // One cannibal crosses right to left
        Add_state(successors, new state(cannibalLeft + 1, missionaryLeft, cannibalRight - 1, missionaryRight, Position.LEFT));
        // Two missionaries cross right to left
        Add_state(successors, new state(cannibalLeft, missionaryLeft + 2, cannibalRight, missionaryRight - 2, Position.LEFT));
        // Two cannibals cross right to left.
        Add_state(successors, new state(cannibalLeft + 2, missionaryLeft, cannibalRight - 2, missionaryRight, Position.LEFT));
    }
    return successors;
}
```

fig: Next State Generation

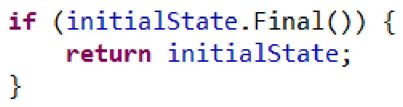
BFS: The final stage is graph traversal. I used BFS (Breadth-first search) here. In BFS, all immediate children of nodes are explored before any of the children's children are considered.

- At first, I have created a constructor which will set the initial value of initial state. I declared a queue named "nodes" and a HashSet named "explored" which will save the visited nodes(states).



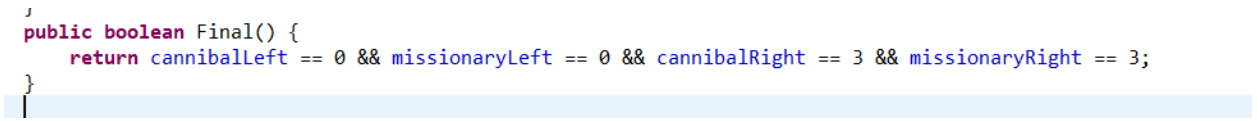
```
public class bfs {
    state initialState;
    Queue<state> nodes;
    Set<state> explored;
    public bfs(state initialState_){
        initialState = initialState_;
        nodes = new LinkedList<state>();
        explored = new HashSet<state>();
    }
```

- In the bfs_excution function, I've first checked if the initialState is actually the final state we expected. If so, then there is no need to traverse the graph. It will just return the state.



```
if (initialState.Final()) {
    return initialState;
}
```

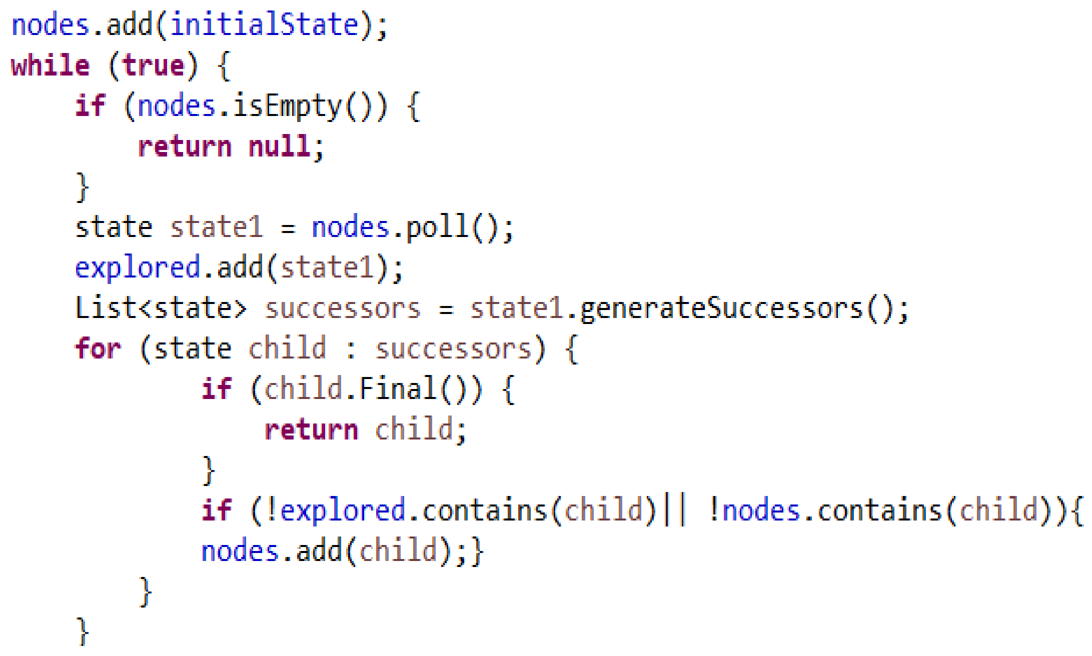
To check if it’s final state or not, a function named Final() is written in state class:



```
public boolean Final() {
    return cannibalLeft == 0 && missionaryLeft == 0 && cannibalRight == 3 && missionaryRight == 3;
}
```

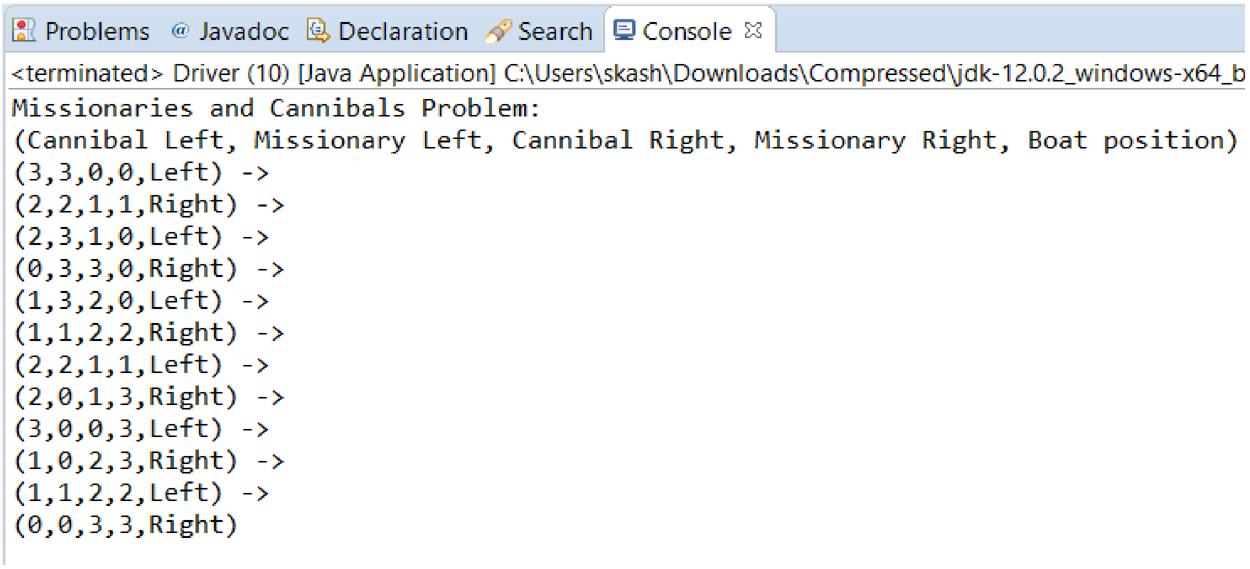
fig: Final() function

- If the initial state is not Final state, then we have to execute bfs. The traversal starts with the first element of nodes which is the root of the tree of states which is actually the initial state where all the missionaries and cannibals are on the left side. So, at first initial state is marked as nodes and added to the queue.
- I removed the first state in a queue and save it in a state and mark it by using explored. Then, generated the next states of this state. For each child, in case it’s not visited, I added it to the nodes queue.
- The graph will traverse through the neighborhood states until it satisfies the Final() function which will return the shortest path to reach the final node. If it never satisfies the Final(), then it will continue until the queue becomes empty and in this case it will return null.



```
nodes.add(initialState);
while (true) {
    if (nodes.isEmpty()) {
        return null;
    }
    state state1 = nodes.poll();
    explored.add(state1);
    List<state> successors = state1.generateSuccessors();
    for (state child : successors) {
            if (child.Final()) {
                return child;
            }
            if (!explored.contains(child)|| !nodes.contains(child)){
            nodes.add(child);}
        }
    }
```

Result:



```
<terminated> Driver (10) [Java Application] C:\Users\skash\Downloads\Compressed\jdk-12.0.2_windows-x64_b
Missionaries and Cannibals Problem:
(Cannibal Left, Missionary Left, Cannibal Right, Missionary Right, Boat position)
(3,3,0,0,Left) ->
(2,2,1,1,Right) ->
(2,3,1,0,Left) ->
(0,3,3,0,Right) ->
(1,3,2,0,Left) ->
(1,1,2,2,Right) ->
(2,2,1,1,Left) ->
(2,0,1,3,Right) ->
(3,0,0,3,Left) ->
(1,0,2,3,Right) ->
(1,1,2,2,Left) ->
(0,0,3,3,Right)
```

fig: Result (Shortest path to cross the river safely)

Discussion: Basically, I divided the whole problem into 3 classes: Driver class, state class, bfs class. Driver class is the main class. State class was mainly used to store the current position of the missionaries, cannibals and the boat. This class also generates the next valid state to solve to problem. Then the bfs class was used. This class contains mainly the graph traversal algorithm breadth first search (BFS) to get the solution of the problem. Depth first search (DFS) could also be used in here instead of BFS. But I preferred to use BFS because it is guaranteed to find the shortest path from initial state to final state. DFS might not guarantee the shortest path and there is a risk of getting stuck in an infinite loop in this problem by using DFS.

Conclusion: The Missionary and Cannibal problem is a famous graph search-based problem. It was a challenge to implement the states and to decide which search based algorithm to choose. This problem can help us to implement AI search techniques to solve real world problem in future.