Assignment 1
CP468
Group 12

Missionaries and Cannibals

Lyndon Rey
Tyler Farkas
Daniel Berezovski

October 6, 2017

0 Introduction

0.1 States

For the purposes of this assignment, the following notation will be used. Firstly, any given state s will be represented by a set of two column vectors in $\mathbb{R}^3$, both containing the elements m, c, b; these elements represent the number of missionaries, number of cannibals, and whether the boat is present, respectively. Each column vector in the set represents a shoreline given in the problem: the left vector represents the starting shore, and the right vector represents the finishing shore, and the subscripts L and R represent the left and right sides, respectively. The right shore is entirely dependent on the left, as the total number of missionaries and cannibals on both shores cannot exceed the initial number, notated as M and C respectively. There is always 1 boat ($B = 1$), so if the left shore has the boat (i.e. $b = 1$), then the right shore will not (i.e. $b = 0$). Thus, the abstract state can be defined as follows:

$$s = \left\{ \begin{bmatrix} m_L \\ c_L \\ b_L \end{bmatrix}, \begin{bmatrix} M - m_L \\ C - c_L \\ 1 - b_L \end{bmatrix} \right\} \tag{1}$$

Continuing to formalize, the following constraints must be put on the variables:

$$0 \leq m \leq M; \quad 0 \leq c \leq C; \quad b = \{0, 1\}; \quad m, c, b \in \mathbb{Z} \tag{2}$$

Another restriction considered when considering possible valid states is the safety of the missionaries. Due to the cannibalistic nature of cannibals, the number of missionaries on both the left shore and right shore can never be less than the number of cannibals; i.e. $m_L \geq c_L;\ m_L \geq 1$ and $m_R \geq c_R;\ m_R \geq 1$ for a state to be valid. To exemplify this, consider the state:

$$s = \left\{ \begin{bmatrix} 3 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix} \right\} M = 3;\ C = 3;\ B = 1 \tag{3}$$

In this state, the left (starting) shore has all 3 missionaries, 1 cannibal, and does not have the boat. The right (finishing) shore has no missionaries (as $m_R = M - m_L = 3 - 3 = 0$), 2 cannibals (as $c_R = C - c_L = 3 - 1 = 2$), and the boat (as $b_R = 1 - b_L = 1 - 0 = 1$). As the missionaries are never outnumbered, this is a valid state. However, if 1 missionary was moved from left to right, then the state would be invalid, as the missionaries would be outnumbered on the right shore.

0.2 Actions

Furthermore, all actions possible for any state will be also be represented as column vectors, with the same elements as the state column vectors (as seen in equation 1). The actions (described in section 1.2) will be of the form:

$$a = \begin{bmatrix} m \\ c \\ b \end{bmatrix} \tag{4}$$

It must be noted that each action on any given state only directly modifies the left shore vector, as the right is dependent on the value of the left (further described in section 0.1).

1 Problem Formulation

1.1 Initial State

The initial state is simply the state in which all missionaries and cannibals, as well as the boat, are on the left shore, thus there are no people or boats on the right shore. Given in the state notation described in section 0.1:

$$s = \left\{ \begin{bmatrix} M \\ C \\ B \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \right\} \tag{5}$$

For example, if the maximum number of missionaries and cannibals were both 3, then the initial state would look as such:

$$s = \left\{ \begin{bmatrix} 3 \\ 3 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \right\} \tag{6}$$

1.2 Actions on States

As stated in section 0.2, each action is represented as a column vector with the elements m, c, b. The boat in question can only hold 2 people, but must hold at least 1, so $1 \leq m + c \leq 2$. All actions require that a boat be present, so $b = 1$. An action vector can be either positive or negative, relative to the left shore: a negative action means that the boat is sailing away from the left shore, and a positive vector implies the opposite. Thus, there are

10 possible actions, with 5 negative/positive pairs. They are:

$$\pm c_2 = \begin{bmatrix}0\\2\\1\end{bmatrix}, \pm c_1 = \begin{bmatrix}0\\1\\1\end{bmatrix}, \pm c_1 m_1 = \begin{bmatrix}1\\1\\1\end{bmatrix}, \pm m_1 = \begin{bmatrix}1\\0\\1\end{bmatrix}, \pm m_2 = \begin{bmatrix}2\\0\\1\end{bmatrix} \tag{7}$$

The notation is simply indicates how many missionaries and/or cannibals are being added or subtracted. For example, if the boat was travelling with 1 missionary and 1 cannibal from the right shore to the left shore, then this action is notated as:

$$+c_1 m_1 = \begin{bmatrix}1\\1\\1\end{bmatrix} \tag{8}$$

The boat is tavelling left-to-right when $b_L = 1$, and from right-to-left when $b_L = 0$. A left-to-right action will always be negative, and a right-to-left action will always be positive (this is explained in more detail in section 1.3).

1.3 Transition Model

The transition model is simple, as an action performed on a state will generate a new state by adding the left shore vector with the action vector, and then calculating the right shore vector using the generated left shore vector. Formalized:

$$\text{Result}(s, a) = \vec{s_L} + \vec{a} \tag{9}$$

This equation only holds true when Result(s, a) is a valid state, using all the guidelines given in section 0.1. For example, if:

$$s = \left\{ \begin{bmatrix}3\\3\\1\end{bmatrix}, \begin{bmatrix}0\\0\\0\end{bmatrix} \right\} \text{ and } a = -c_1 = -\begin{bmatrix}0\\1\\1\end{bmatrix} \tag{10}$$

then:

$$\text{Result}(s, a) = \vec{s_L} + \vec{a} = \begin{bmatrix}3\\3\\1\end{bmatrix} - \begin{bmatrix}0\\1\\1\end{bmatrix} = \begin{bmatrix}3\\2\\0\end{bmatrix} \tag{11}$$

so the generated state would be:

$$s = \left\{ \begin{bmatrix}3\\2\\0\end{bmatrix}, \begin{bmatrix}0\\1\\1\end{bmatrix} \right\} \tag{12}$$

which is a valid state, given the terms in section 0.1. This means that the boat brought 1 cannibal from the left shore to the right shore.

1.4 Goal Test

The goal test is to check whether the left shore is empty in a given state. Formally, the goal is achieved when:

$$s = \left\{ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 3 \\ 3 \\ 1 \end{bmatrix} \right\} \tag{13}$$

From this, a Boolean function can be created:

$$\text{Goal}(s) = \begin{cases} \text{True}, & \text{if } s_L = \vec{0} \\ \text{False}, & \text{if } s_L \neq \vec{0} \end{cases} \tag{14}$$

1.5 Path Cost

The cost of each boat trip across the river is 1, so the path cost is the number of river crossings. The optimal path is the path in which the boat crosses the river the fewest times.

2 State Space Diagram

In the following state space diagram, it is assumed that each river crossing alternates addition and subraction, beginning with subtraction. However, each action is bidirectional, and if going from a frontier state back to a previously-seen state, then the sign of the action is reversed (this should never happen, as looping paths will never be more optimal than non-looping paths).

$$\begin{tikzcd}
 & & & \begin{bmatrix}1\\1\\0\end{bmatrix},\begin{bmatrix}2\\2\\1\end{bmatrix} \arrow[dd, leftrightarrow, "{+c_1 \atop +m_1}"] & & \\
 & & \begin{bmatrix}3\\1\\1\end{bmatrix},\begin{bmatrix}0\\2\\0\end{bmatrix} \arrow[ur, leftrightarrow, "+m_2"] & & & \\
 & & & \begin{bmatrix}2\\2\\1\end{bmatrix},\begin{bmatrix}1\\1\\0\end{bmatrix} \arrow[dd, leftrightarrow, "-m_2"] & & \\
 & \begin{bmatrix}3\\2\\0\end{bmatrix},\begin{bmatrix}0\\1\\1\end{bmatrix} & \begin{bmatrix}3\\0\\0\end{bmatrix},\begin{bmatrix}0\\3\\1\end{bmatrix} \arrow[uu, leftrightarrow, "+c_1"] & & & \\
\begin{bmatrix}3\\3\\1\end{bmatrix},\begin{bmatrix}0\\0\\0\end{bmatrix} \arrow[ur, leftrightarrow, "-c_1"] \arrow[r, leftrightarrow, "-c_2"] \arrow[ddr, leftrightarrow, "{-c_1 \atop -m_1}"'] & \begin{bmatrix}3\\1\\0\end{bmatrix},\begin{bmatrix}0\\2\\1\end{bmatrix} \arrow[dr, leftrightarrow, "+c_1"] & & \begin{bmatrix}0\\2\\0\end{bmatrix},\begin{bmatrix}3\\1\\1\end{bmatrix} \arrow[dd, leftrightarrow, "+c_1"] & & \\
 & & \begin{bmatrix}3\\2\\1\end{bmatrix},\begin{bmatrix}0\\1\\0\end{bmatrix} \arrow[uu, leftrightarrow, "-c_2"] & & \begin{bmatrix}0\\1\\1\end{bmatrix},\begin{bmatrix}3\\2\\0\end{bmatrix} \arrow[r, leftrightarrow, "+c_1"] & \begin{bmatrix}0\\0\\0\end{bmatrix},\begin{bmatrix}3\\3\\1\end{bmatrix} \arrow[dd, leftrightarrow, "-c_2"] \\
 & \begin{bmatrix}2\\2\\0\end{bmatrix},\begin{bmatrix}1\\1\\1\end{bmatrix} \arrow[ur, leftrightarrow, "+m_1"'] & & \begin{bmatrix}0\\3\\1\end{bmatrix},\begin{bmatrix}3\\0\\0\end{bmatrix} \arrow[d, leftrightarrow, "-c_2"] & & \\
 & & & \begin{bmatrix}0\\1\\0\end{bmatrix},\begin{bmatrix}3\\2\\1\end{bmatrix} \arrow[r, leftrightarrow, "+m_1"] \arrow[rr, leftrightarrow, bend right, "+c_1"'] & \begin{bmatrix}1\\1\\1\end{bmatrix},\begin{bmatrix}2\\2\\0\end{bmatrix} \arrow[uur, leftrightarrow, "{-c_1 \atop -m_1}"] & \begin{bmatrix}0\\2\\1\end{bmatrix},\begin{bmatrix}3\\1\\0\end{bmatrix}
\end{tikzcd}$$

3 Implementation

For the implementation of the above algorithm, we used a depth-first search strategy written in Java. The main method contains the ability to change the starting number of missionaries and cannibals, as well as the boat capacity; with these parameters, the program outputs the final DFS-generated optimal path as a terminal output, with each line representing a node passed through.

3.1 Classes

3.1.1 Main

The main class contains the DFS implementation, using the node and state classes (sections 3.1.2 and 3.1.3 respectively).

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Stack;

public class Main {

  public static void main(String[] args) throws Exception {
    dfs(3, 3);
  }

  private static Node dfs(int numMissionaries, int numCannibals) {

    State initialState;

    try {

      initialState = State.createState(new int[]
          {numMissionaries, numCannibals, 1}, new int[] {0, 0, 0});
    } catch (Exception e) {

      return null;
    }

    Node initialNode = new Node(initialState);

    ArrayList<State> existingStates = new ArrayList<State>();
    existingStates.add(initialState);

    Stack<Node> path = new Stack<Node>();
    path.push(initialNode);

    dfs_traverse(initialNode, existingStates, path);

    for (Node node : path) {

      System.out.println(node.getState());
    }
    return initialNode;
  }

  private static void dfs_traverse(Node node, ArrayList<State>
      existingStates, Stack<Node> path) {

    final int BOAT_CAPACITY = 2;

    for (int i = 0; i <= BOAT_CAPACITY; i++) {

      for (int j = 0; j <= BOAT_CAPACITY; j++) {

        if (i == 0 && j == 0 || i + j > BOAT_CAPACITY) continue;

        int[] start = Arrays.copyOf(node.getState().getStart(),
            node.getState().getStart().length);
        int[] end = Arrays.copyOf(node.getState().getEnd(),
            node.getState().getEnd().length);

        if (start[State.BOAT_LOCATION] == 1) {

          start[State.MISSIONARIES] -= i;
          end[State.MISSIONARIES] += i;

          start[State.CANNIBALS] -= j;
          end[State.CANNIBALS] += j;

          start[State.BOAT_LOCATION] = 0;
          end[State.BOAT_LOCATION] = 1;
        } else {

          start[State.MISSIONARIES] += i;
          end[State.MISSIONARIES] -= i;

          start[State.CANNIBALS] += j;
          end[State.CANNIBALS] -= j;

          start[State.BOAT_LOCATION] = 1;
          end[State.BOAT_LOCATION] = 0;
        }

        try {

          State state = State.createState(start, end);
          if (!existingStates.contains(state)) {

            existingStates.add(state);

            node.addNode(new Node(state));
          }
        } catch (Exception e) {}
      }
    }

    if (node.getNodes().isEmpty())
      path.pop();

    for (Node newNode : node.getNodes()) {

      path.push(newNode);

      if (!newNode.getState().isGoal())
        dfs_traverse(newNode, existingStates, path);
      else break;
    }
  }
}
```

3.1.2 Node

The Node class contains a State (section 3.1.2), as well as an array of all nodes above it.

```
import java.util.ArrayList;

public class Node {

  private State state;
  private ArrayList<Node> nodes = new ArrayList<Node>();

  Node(State state) {

    this.state = state;
  }

  public State getState() {
    return state;
  }
  public ArrayList<Node> getNodes() {
    return nodes;
  }

  public void addNode(Node node) {
    nodes.add(node);
  }
}
```

3.1.3 State

The State class contains all the information required to create a valid state, including the left shore and right shore vectors, and maximum values for missionaries, cannibals, and boat capacity. Also, it contains the method for the transition model ('move'), goal state check ('isGoal'), and validity check ('validState').

```
import java.util.Arrays;

public class State {

  public static final int MISSIONARIES = 0;
  public static final int CANNIBALS = 1;
  public static final int BOAT_LOCATION = 2;

  private static final int VECTOR_LENGTH = 3;

  private int[] start = new int[VECTOR_LENGTH];
  private int[] end = new int[VECTOR_LENGTH];

  private int maxMissionaries;
  private int maxCannibals;

  private State(int[] start, int[] end) throws Exception {

    this.start = start;
    this.end = end;

    maxMissionaries = start[MISSIONARIES] + end[MISSIONARIES];
    maxCannibals = start[CANNIBALS] + end[CANNIBALS];
  }
```

```
public static State createState(int[] start, int[] end) throws
    Exception {

   if (start.length > VECTOR_LENGTH || end.length > VECTOR_LENGTH)
      throw new Exception("Make sure both of the arrays have a
          length of 3");

   State newState = new State(start, end);

   if (!validState(newState))
      throw new Exception("State is invalid");

   return newState;
}

public static boolean validState(State state) {

   if (state.getStart()[MISSIONARIES] != 0 &&
       state.getStart()[MISSIONARIES] <
       state.getStart()[CANNIBALS])
      return false;
   else if (state.getEnd()[MISSIONARIES] != 0 &&
       state.getEnd()[MISSIONARIES] < state.getEnd()[CANNIBALS])
      return false;
   else if (state.getStart()[MISSIONARIES] +
       state.getEnd()[MISSIONARIES] > state.maxMissionaries)
      return false;
   else if (state.getStart()[CANNIBALS] +
       state.getEnd()[CANNIBALS] > state.maxCannibals)
      return false;
   else if (state.getStart()[MISSIONARIES] < 0 ||
       state.getEnd()[MISSIONARIES] < 0)
      return false;
   else if (state.getStart()[CANNIBALS] < 0 ||
       state.getEnd()[CANNIBALS] < 0)
      return false;

   return true;
}

public State move(int numMissionaries, int numCannibals) throws
    Exception {

   int[] newStart = this.start;
   newStart[MISSIONARIES] += numMissionaries;
```

```
    newStart[CANNIBALS] += numCannibals;
    newStart[BOAT_LOCATION] = start[BOAT_LOCATION] == 1 ? 0 : 1;

    int[] newEnd = this.end;
    newEnd[MISSIONARIES] -= numMissionaries;
    newEnd[CANNIBALS] -= numCannibals;
    newEnd[BOAT_LOCATION] = start[BOAT_LOCATION] == 1 ? 1 : 0;

    return createState(newStart, newEnd);
  }

  public boolean isGoal() {

    return start[MISSIONARIES] == 0 && start[CANNIBALS] == 0 &&
        end[MISSIONARIES] == maxMissionaries && end[CANNIBALS] ==
        maxCannibals;
  }

  public int[] getStart() {
    return start;
  }

  public int[] getEnd() {
    return end;
  }

  public boolean boatAtStart() {

    return start[BOAT_LOCATION] == 1;
  }

  public int getMaxMissionaries() {
    return maxMissionaries;
  }

  public int getMaxCannibals() {
    return maxCannibals;
  }

  public String toString() {

    return "Start: " + Arrays.toString(start) + " End: " +
        Arrays.toString(end);
  }
```

```
   @Override
    public boolean equals(Object object) {

      State state = (State) object;

      return (Arrays.equals(start, state.getStart()) &&
          Arrays.equals(end, state.getEnd()));
    }
}
```