Missionaries and Cannibals Solution

Mahesh Devalla

September 20, 2016

1 Introduction

Missionaries and cannibals is problem where missionaries and cannibals are to be moved to the other bank of the river safely. In this problem let us consider there are "m" number of missionaries and "c" number of cannibals and the boat "b" is on the one side of the river.The major issues involved with the problem are

- The number of missionaries has to be grater than cannibals on any side of the river.
- There should be a person to row the boat from one bank of the river to the other bank.

While performing the task we have many states in which some are legal and some are illegal. As stated earlier if we have "m"missionaries, "c" cannibals and "b" boat. It is represented as (m,c,b) state where the value of the boat "b" changes from one bank of the river to the other bank. By intuition we can guess that "m"(missionaries) should be greater than or equal to "c"(cannibals).

Let us consider a few states in the pictorial representation below where the a sample of valid and invalid states from a given state are represented.
(3,3,1)——>(1,3,1) (valid state)
(3,3,1)——>(1,3,0) (Invalid state)
(3,3,1)——>(2,2,0) (Invalid state) etc.

In the second level the states would be
(2,2,0)——>(0,2,1))(Invalid state)
(2,2,0)——>(1,1,1)(valid state)
(2,2,0)——>(1,2,1) (Invalid state) etc.

2 Implementation of the model

The model is implemented in `CannibalProblem.java`. Here's my code for `getSuccessors`:(The indentation may be a bit annoying at some places as I am new to Latex, I tried to an extent to keep the code clean but there might be a bit mess up in indentation :). I Will improve slowly, Thank you :)

```
public ArrayList<UUSearchNode> getSuccessors() {
 int boatDirection,totalMissionaries,totalCannibals;
 ArrayList<UUSearchNode> nextnode = new ArrayList<>();
 //The position of boat is changed if it moved from one bank to other bank
  boatDirection=(state[2]==0)?1:0;
 // Moving from one bank to the other bank of the river.
 for(int missionaries = 0; missionaries <= BOAT_SIZE; missionaries++)
 {
    for(int cannibals = 0; cannibals <= BOAT_SIZE - missionaries; cannibals++)
    {
```

```
      if(boatDirection==0)// reducing the missionaries and cannibals from first bank after they
          are moved to other bank
       {
         totalMissionaries = state[0] - missionaries;
         totalCannibals = state[1] - cannibals;
      }
      else
      {
         totalMissionaries = state[0] + missionaries;
         totalCannibals = state[1] + cannibals;
      }
CannibalNode nextreferencenode = new
    CannibalNode(totalMissionaries,totalCannibals,boatDirection,depth);
        if(cannibals+missionaries!= 0)
           if(nextreferencenode.areMissionariesSafe())
          nextnode.add(nextreferencenode);
          }
      }
       return nextnode;
}
```

The basic idea of `getSuccessors` is to produce all the possible states from a state,which is given randomly. For example if the given state is (3, 3, 1) then possible correct states would be (2,2,0), (3,1,0) etc. These states are mostly dependent upon the place of the boat either 1 or 0, because we can intuitively calculate the number of missionaries and cannibals. At any point of time the number of missionaries should be greater than cannibals regardless of the boat position is at 0 or 1. There is an exception that if there are no missionaries at a bank then there can be more cannibals because the missionaries would be zero.

I used a method `areMissionariesSafe()` that returns `true` if the number of missionaries are greater than cannibals irrespective of the position of the boat, whether it is at 1 or 0(either any one of the side of the bank). It would return `false` if in the other case where if more cannibals are present than missionaries. There is a special case here either the number of missionaries or the number of cannibals should be greater than or equal to zero.

```
  private boolean areMissionariesSafe() {
if( state[0] > 0 && state[0] < state[1]) // If number of missionaries are less than number of
    cannibals cannibals, then return from the function
return false;
else if( state[0] < 0 || state[1] < 0) //If provided with wrong input (Missionaries or cannibals
    less than zero)
return false;
else if((totalMissionaries-state[0]) > 0 && (totalMissionaries-state[0]) <
    (totalCannibals-state[1])) // If missionaries less than cannibals
return false;
else if(state[0] > totalMissionaries || state[1] > totalCannibals)// invalid number of
    missionaries and cannibals
return false;
else
return true;// if missionaries are greater than cannibals

  }
```

3 Breadth-first search

Breadth first search is similar to the breadth first search of an undirected graph but here in this case we are not sure of the states prior itself. So choosing data structures and keeping track of space complexity is a crucial factor, where we may visit unnecessary nodes and also several loops might occur if not handled correctly. The data structures we can use here depend on user to user where I used hashtable (even we can use array but I preferred hastable because of the easy use of access) and I used Queue implementation of a LinkedList(where I used a parent reference to store the child object so that many other methods of LinkedList can also be used).

The implementation of the breadth first search is as follows:

```
public List<UUSearchNode> breadthFirstSearch(){
  resetStats();

  // You will write this method
  HashMap<UUSearchNode,UUSearchNode> node = new HashMap<>();
  Queue<UUSearchNode> queue = new LinkedList<>(); // Queue implementation of LinkedList (parent
      reference is used to stored child object)
  HashSet<UUSearchNode> visitedNodes = new HashSet<>(); // HashSet to store the visited nodes

  queue.add(startNode);
  visitedNodes.add(startNode);
  incrementNodeCount();

  while(queue.isEmpty()==false)
  {
  UUSearchNode currentnode = queue.remove();
  if (currentnode.goalTest())
   return backchain(currentnode, node);

  ArrayList<UUSearchNode> al = currentnode.getSuccessors();
  for (UUSearchNode iterator : al)
  {
   if(!visitedNodes.contains(iterator))
   {
    node.put(iterator, currentnode);
    queue.add(iterator);
    visitedNodes.add(iterator);
    incrementNodeCount();
   }
  }
  updateMemory(visitedNodes.size());
  }
  return null;

   }
```

Additionally we need a method called backChain() to get the nodes or the states in this case.Below is the code of the backChain() method.

```
private List<UUSearchNode> backchain(UUSearchNode node,HashMap<UUSearchNode, UUSearchNode>
    visited) {
  // you will write this method
  if(visited.containsKey(node)==false)
   return null;
```

```
List<UUSearchNode> listOfNodes = new LinkedList<>();
listOfNodes.add(node);
while(visited.containsKey(node))
{
 UUSearchNode prev = visited.get(node);
 if(!prev.equals(startNode))
  listOfNodes.add(prev);
 node = prev;
}
return listOfNodes;}
```

The output i.e. states of the breadth first search are as follows when an input of (3, 3, 1, 0, 0, 0);(The output may be in reverse order which is due to the List instead of LinkedList (parent child relationship, which applies to other outputs' also.)

bfs path length: 11 [[0, 0, 0], [0, 2, 1], [0, 1, 0], [0, 3, 1], [0, 2, 0], [2, 2, 1], [1, 1, 0], [3, 1, 1], [3, 0, 0], [3, 2, 1], [3, 1, 0]]
Nodes explored during last search: 15
Maximum memory usage during last search 15

The output i.e. states of the breadth first search are as follows when an input of (8, 5, 1, 0, 0, 0);(The output may be in reverse order which is due to the List instead of LinkedList (parent child relationship, which applies to other outputs' also.)

bfs path length: 23 [[0, 0, 0], [1, 1, 1], [1, 0, 0], [2, 1, 1], [2, 0, 0], [3, 1, 1], [3, 0, 0], [3, 2, 1], [3, 1, 0], [4, 2, 1], [4, 1, 0], [4, 3, 1], [4, 2, 0], [5, 3, 1], [5, 2, 0], [5, 4, 1], [5, 3, 0], [6, 4, 1], [6, 3, 0], [8, 3, 1], [8, 2, 0], [8, 4, 1], [8, 3, 0]]
Nodes explored during last search: 62
Maximum memory usage during last search 62

4 Memoizing depth-first search

Memoizing is almost we can consider as type of dynamic programming in one sense, in dynamic programming we store the previous computed result and here in Memoizing depth-first search we keep storing the previous states i.e nodes and also the depth. This is almost similar to the regular depth first search so there won't be any change in the data structures used. Below is the implementation of the Memoizing depth-first search.

```
public List<UUSearchNode> depthFirstMemoizingSearch(int maxDepth) {
 resetStats();
 // You will write this method
 HashMap<UUSearchNode,Integer> visitedNodes = new HashMap<>();
 visitedNodes.put(startNode,0);
 return dfsrm(startNode, visitedNodes, 0, maxDepth);

  }

  // recursive memoizing dfs. Private, because it has the extra
  // parameters needed for recursion.
  private List<UUSearchNode> dfsrm(UUSearchNode currentNode, HashMap<UUSearchNode, Integer>
      visited,
 int depth, int maxDepth) {
```

```
//  "base case" and "recursive case" that any recursive function has.
updateMemory(visited.size());
incrementNodeCount();
// you write this method. Comments *must* clearly show the
//  "base case" and "recursive case" that any recursive function has.

if(currentNode.goalTest())// Testing and returning nodes(sates)
{
List<UUSearchNode> listOfNodes = new LinkedList<>();
return listOfNodes;
}
else if(depth>=maxDepth) // If depth exceeded beyond the limit then return
{
return null;
}
for(UUSearchNode uusnloop: currentNode.getSuccessors())
{
if(!visited.containsKey(uusnloop) || visited.get(uusnloop) >= depth)
{
visited.put(uusnloop,depth+1);
List<UUSearchNode> states = (LinkedList<UUSearchNode>) dfsrm(uusnloop, visited,depth+1,maxDepth);
if(states!=null)
{
 states.add(uusnloop);
 return states;
}
}
}
return null;
 }
```

The output i.e. states of the Memoizing depth first search are as follows when an input of (3, 3, 1, 0, 0, 0);

dfs memoizing path length: 11 [[0, 0, 0], [0, 2, 1], [0, 1, 0], [0, 3, 1], [0, 2, 0], [2, 2, 1], [1, 1, 0], [3, 1, 1], [3, 0, 0], [3, 2, 1], [3, 1, 0]]
Nodes explored during last search: 13
Maximum memory usage during last search 13

The output i.e. states of the Memoizing depth first search are as follows when an input of (8, 5, 1, 0, 0, 0);

dfs memoizing path length: 33 [[0, 0, 0], [0, 2, 1], [0, 1, 0], [1, 1, 1], [1, 0, 0], [2, 1, 1], [2, 0, 0], [2, 2, 1], [2, 1, 0], [3, 1, 1], [3, 0, 0], [3, 2, 1], [3, 1, 0], [3, 3, 1], [3, 2, 0], [4, 2, 1], [4, 1, 0], [4, 3, 1], [4, 2, 0], [4, 4, 1], [4, 3, 0], [5, 3, 1], [5, 2, 0], [5, 4, 1], [5, 3, 0], [5, 5, 1], [5, 4, 0], [6, 4, 1], [6, 3, 0], [8, 3, 1], [8, 2, 0], [8, 4, 1], [8, 3, 0]] Nodes explored during last search: 40
Maximum memory usage during last search 40

5 Path-checking depth-first search

The name itself suggest that we are more concerned about path, even though all the depth first searches do concern about the same, here we don't consider the invalid nodes(states in this problem). We just skip the

nodes that are invalid and almost similar to the Memoizing depth-first search but uses recursion functions. Below is the code of my implementation of Path-checking depth-first search.

```
private List<UUSearchNode> dfsrpc(UUSearchNode currentNode, HashSet<UUSearchNode> currentPath,
        int depth, int maxDepth) {

     // you write this method
     incrementNodeCount();
     updateMemory(currentPath.size());
     if(currentNode.goalTest())
     {
        List<UUSearchNode> list = new LinkedList<>();
        return list;
     }
     else if(depth>=maxDepth) // IIf Depth increased more than given depth
     {
        return null;
     }

     /*
      * Recursion Case
      */
     for(UUSearchNode UUSNloop: currentNode.getSuccessors()){
        if(!currentPath.contains(UUSNloop))
        {
  currentPath.add(UUSNloop);
  List<UUSearchNode> listOfNodes = dfsrpc(UUSNloop, currentPath, depth + 1, maxDepth);

  if(listOfNodes!=null) // Add node to loop if function gives not null
  {
   listOfNodes.add(UUSNloop);
   return listOfNodes;
  }
  else
  {
   currentPath.remove(UUSNloop);
  }
        }
     }
     return null;
   }
```

The output i.e. states of the Path-checking depth-first search search are as follows when an input of (3, 3, 1, 0, 0, 0);
dfs path checking path length: 13 [[3, 2, 0], [3, 3, 1], [3, 1, 0], [3, 2, 1], [3, 0, 0], [3, 1, 1], [1, 1, 0], [2, 2, 1], [0, 2, 0], [0, 3, 1], [0, 1, 0], [0, 2, 1], [0, 0, 0]]
Nodes explored during last search: 14
Maximum memory usage during last search 13

The output i.e. states of the Path-checking depth-first search search are as follows when an input of (8, 5, 1, 0, 0, 0);
dfs path checking path length: 35 [[8, 4, 0], [8, 5, 1], [8, 3, 0], [8, 4, 1], [8, 2, 0], [8, 3, 1], [6, 3, 0], [6, 4, 1], [5, 4, 0], [5, 5, 1], [5, 3, 0], [5, 4, 1], [5, 2, 0], [5, 3, 1], [4, 3, 0], [4, 4, 1], [4, 2, 0], [4, 3, 1], [4, 1, 0], [4, 2, 1],

[3, 2, 0], [3, 3, 1], [3, 1, 0], [3, 2, 1], [3, 0, 0], [3, 1, 1], [2, 1, 0], [2, 2, 1], [2, 0, 0], [2, 1, 1], [1, 0, 0], [1, 1, 1], [0, 1, 0], [0, 2, 1], [0, 0, 0]] Nodes explored during last search: 41
Maximum memory usage during last search 35

6 Iterative deepening search

Iterative Deepening search depends on depth of the graph(tree). If a state is found valid and we increase the depth but before exploring the next level we first check that if this level and previous level is completely traversed or not, this is a bit faster in the initial cases and also gives an optimal solution when the depth is small. We use path checking depth first search in this algorithm and also increase the depth as we traverse from one state to another state. In all the above algorithms I used the same data structures, Hashset to search the path and a LinkedLits implementation of queue.

```
public List<UUSearchNode> IDSearch(int maxDepth) {
    resetStats();
    for(int i = 0; i < maxDepth; i++)
    {
       HashSet<UUSearchNode> path = new HashSet<>();
       path.add(startNode);
       List<UUSearchNode> states = dfsrpc(startNode, path, 0, i);

       if(states!=null)
          return states;
    }
    return null;
  }
```

The output i.e. states of the Iterative deepening search are as follows when an input of (3, 3, 1, 0, 0, 0);
Iterative deepening (path checking) path length:11 Nodes explored during last search: 162
Maximum memory usage during last search 12

The output i.e. states of the Iterative deepening search search are as follows when an input of (8, 5, 1, 0, 0, 0);
Iterative deepening (path checking) path length:23
Nodes explored during last search: 19425295
Maximum memory usage during last search 24

7 Lossy missionaries and cannibals

If a case like no more than "x" missionaries are to be eaten like in the example of input (8,5,1,0,0,0), which by intuition we can say that number of states increase as the method areMissionariesSafe() would result in a good number of states when compared with the input (3,3,1,0,0,0), where we have a no restriction about the number of missionaries eaten by cannibals. We have to make a few changes to the code which areMissionariesSafe() method return true in more cases, we can get the value by just subtracting the number of missionaries from total missionaries, which is surely a subset of the total missionaries. There would be changes in the getSuccessors() method to accommodate the new states in the graph.

8 Extra Information

This algorithm(code) would work even for different boat sizes, please see the output section at end of the each method above.