Missionaries and Cannibals Solution

Delos Chang

January 13, 2014

0.1 Introduction

0.2 Implementation of the model

The model is implemented in `CannibalProblem.java`. Here's my code for `getSuccessors`:

```
public ArrayList<UUSearchNode> getSuccessors() {
  // the final array to be returned
  ArrayList<UUSearchNode> retArr = new ArrayList<UUSearchNode>();

  int boatPlace = state[2];
  int candMissionaries = -1;
  int candCannibals = -1;
  int candBoat = -1;

  // Determine which missionaries and cannibals can travel
  if (boatPlace == 1){
    System.out.println("Boat @ starting side");
    // the candidate missionaries that can travel on the boat
    candMissionaries = state[0];
    candCannibals = state[1];
    candBoat = 0;
  } else if (boatPlace == 0){
    candMissionaries = totalMissionaries - state[0];
    candCannibals = totalCannibals - state[1];
    candBoat = 1;
  } else {
    // not valid input for boat
    System.out.println("Boat place not valid: " + boatPlace);
    System.exit(1);
  }

  for(int missCount=candMissionaries; missCount>=0; missCount--){
    for(int cannCount=candCannibals; cannCount>=0; cannCount--){
      System.out.println("Checking ("+missCount+","+cannCount+")");
      CannibalNode possNode;

      // Must fit in boat and have at least one missionary rowing
      if (missCount + cannCount <= BOAT_SIZE && (missCount > 0 || cannCount > 0)){
        // Must have something happen
        if (missCount == 0 && cannCount == 0){
          continue;
        }

        if (boatPlace == 1){
          // boat on starting side so starting side is subtracted
          possNode = new CannibalNode(candMissionaries - missCount,
              candCannibals - cannCount, candBoat, depth+1);
        } else {
```

```
            // boat on other side so starting side is added
            possNode = new CannibalNode( state[0] + missCount,
                state[1] + cannCount, candBoat, depth+1);
          }
          System.out.println("("+possNode.state[0]+","+possNode.state[1]+","+possNod
        } else {
          continue;
        }

        boolean isSafe = isSafeState(possNode);
        if (isSafe){
          // State is valid, add to the array
          System.out.println("Adding " + possNode);
          retArr.add(possNode);
        } else {
          // Node was not a valid state
          System.out.println("Not a safe state!");
        }
      }
    }
    return retArr;
}
```

The basic idea of `getSuccessors` is that it returns an array of valid states based off of the node that was passed in. So, given the first start node: (331), it will run through each combination of missionaries and cannibals. So it checks 3 Missionaries and 3 Cannibals on the boat, then 3 Missionaries 2 Cannibals, then 3 Missionaries 1 Cannibal, 3 Missionaries 0 Cannibals, 2 Missionaries 1 Cannibal and so on.

Then, it will check if that combination is valid for the boat size. If it is, it will add or subtract (depending on where the boat is) to create the new state. This new state is passed into the `isSafeState` to verify whether the missionaries are safe. If they are, return true. If not, return false.

I used a method `isSafeState` that returns `true` if the missionaries do not get eaten by the cannibals. We can check this by first validating that there are missionaries on either side of the river. Then we MUST make sure that the cannibals cannot outnumber missionaries on either side.

If there are no missionaries on one side, that implies that all the missionaries are on the other side. Thus, we just need to check whether the missionaries are outnumbered on the other side. This will cover the edge case where there are no missionaries but more than 0 cannibals on the same side. Without this edge case, the algorithm would not correctly process a node like (031).

```
// checks whether the humans get eaten :(
private boolean isSafeState(CannibalNode node){
  // miss + cannibals on starting side
  int startMissionaries = node.state[0];
  int startCannibals = node.state[1];
  // miss + cannibals on other side
  int otherMissionaries = totalMissionaries - startMissionaries;
  int otherCannibals = totalCannibals - startCannibals;
```

```
  if (startMissionaries != 0 && otherMissionaries != 0){
    // must have more missionaries than cannibals or else eaten :(
    if (startMissionaries >= startCannibals &&
        otherMissionaries >= otherCannibals){
      return true;
    }
  }

  // If no missionaries are on one side, must mean missionaries are on other side
  // Therefore, we check if the missionaries are outnumbered on the other side
  // Also starting with 1 Cannibal and 0 Missionaries on starting side is NOT
  // a valid state
  if (otherMissionaries == 0){
    return startMissionaries >= startCannibals;
  }

  if (startMissionaries == 0){
    return otherMissionaries >= otherCannibals;
  }

  // not a safe state
  return false;
}
```

Overall, I tested both methods by manually expanding the states graph from the initial problem setup of (331) and then cross-checking with the results that the `getSuccessors` method yields. This verifies the method for the first level. We could test the second and third levels in the states graph via grabbing the array returned by the first `getSuccessors` method and calling `getSuccessors` again on one of the valid states.

```
// Test levels 2 and 3
ArrayList<UUSearchNode> retArr = mcProblem.startNode.getSuccessors();
System.out.println(retArr.get(0).getSuccessors().get(1).getSuccessors());
```

Since the code works for levels 1, 2 and 3 and matches the actual valid states drawn manually, it covers the the cases for the boat going from one side to another. In other words, given an arbitrary state at level x, we get the valid successor states for state x+1 level. Thus, by induction, we prove the correctness of the code.

0.3 Breadth-first search

0.4 Memoizing depth-first search

```
public List<UUSearchNode> depthFirstMemoizingSearch(int maxDepth) {
  resetStats();

  // You will write this method
```

```
    List<UUSearchNode> retArr = new ArrayList<UUSearchNode>();
    HashMap<UUSearchNode, Integer>visited = new HashMap<UUSearchNode, Integer>();

    List<UUSearchNode> startChildren = startNode.getSuccessors();

    // mark startNode as visited
    visited.put(startNode, startNode.getDepth());

    for(int i=0; i < startChildren.size(); i++){
      UUSearchNode child = startChildren.get(i);
      if (!visited.containsKey(child)){
        retArr = dfs(child, visited, child.getDepth(), CannibalDriver.MAXDEPTH);
        if (retArr != null){
          return retArr;
        }
      }

    }
    return null;
  }

  // recursive memoizing dfs. Private, because it has the extra
  // parameters needed for recursion.
  private List<UUSearchNode> dfs(UUSearchNode currentNode, HashMap<UUSearchNode, Int
      int depth, int maxDepth) {
    System.out.println("Following " + currentNode);

    // keep track of stats; these calls charge for the current node
    updateMemory(visited.size());
    incrementNodeCount();
    List<UUSearchNode> retArr= new ArrayList<UUSearchNode>();

    // you write this method.  Comments *must* clearly show the
    //  "base case" and "recursive case" that any recursive function has.

    // BASE CASE: currentNode is the goal Node
    if (currentNode.goalTest()){
      retArr.add(currentNode);
      return retArr;
    } else {
      // RECURSIVE CASE: not the goalNode, continue recursing down successor line
      List<UUSearchNode> currentChildren = currentNode.getSuccessors();
      visited.put(currentNode, currentNode.getDepth());

      // stop if the depth is exceeded
      if (depth > maxDepth){
        System.out.println("Depth exceeded! Try a shorter route");
        return null;
      }
```

```
    for(int i=0; i < currentChildren.size(); i++){
      UUSearchNode child = currentChildren.get(i);

      // if it is depth limited, we need to make sure that the DFS doesn't stop at
      // a duplicate (compare depths)
      if (!visited.containsKey(child) || visited.get(child) > depth){
        retArr = dfs(child, visited, child.getDepth(), maxDepth);
        if (retArr != null){
          retArr.add(child);
          return retArr;
        }
      }
    }
  System.out.println("No path found!");
    return null;
  }

}
```

The base case for memoizing DFS is if the node being examined is the goal node. In that case, we do not enter the recursive case and we can simply add this goal node into a list and return it.

The recursive case is if the node being examined is NOT the goal node.

However, memoizing dfs does NOT save significant memory compared to BFS. First, we consider whether the state space is finite or infinite. Given a finite state space of size n, the HashMap implemented in both DFS and BFS cannot exceed n nodes at any given time. Thus, both BFS and DFS implementations in the finite search space have $O(n)$ memory usage.

However, by nature of BFS, because it ripples out from the start node and checks each successive depth from the start node, in an infinite state space, BFS will terminate with time complexity $O(b^d)$, where d is the depth of the goal from start node and b is the upper bound of the graph's branching factor.

Conversely, in an infinite search space, DFS may not even terminate! Hence, in some cases, DFS may have substantially more exhaustive memory usage.

0.5 Path-checking depth-first search

0.6 Iterative deepening search

0.7 Lossy missionaries and cannibals