Missionaries and Cannibals Solution

Delos Chang

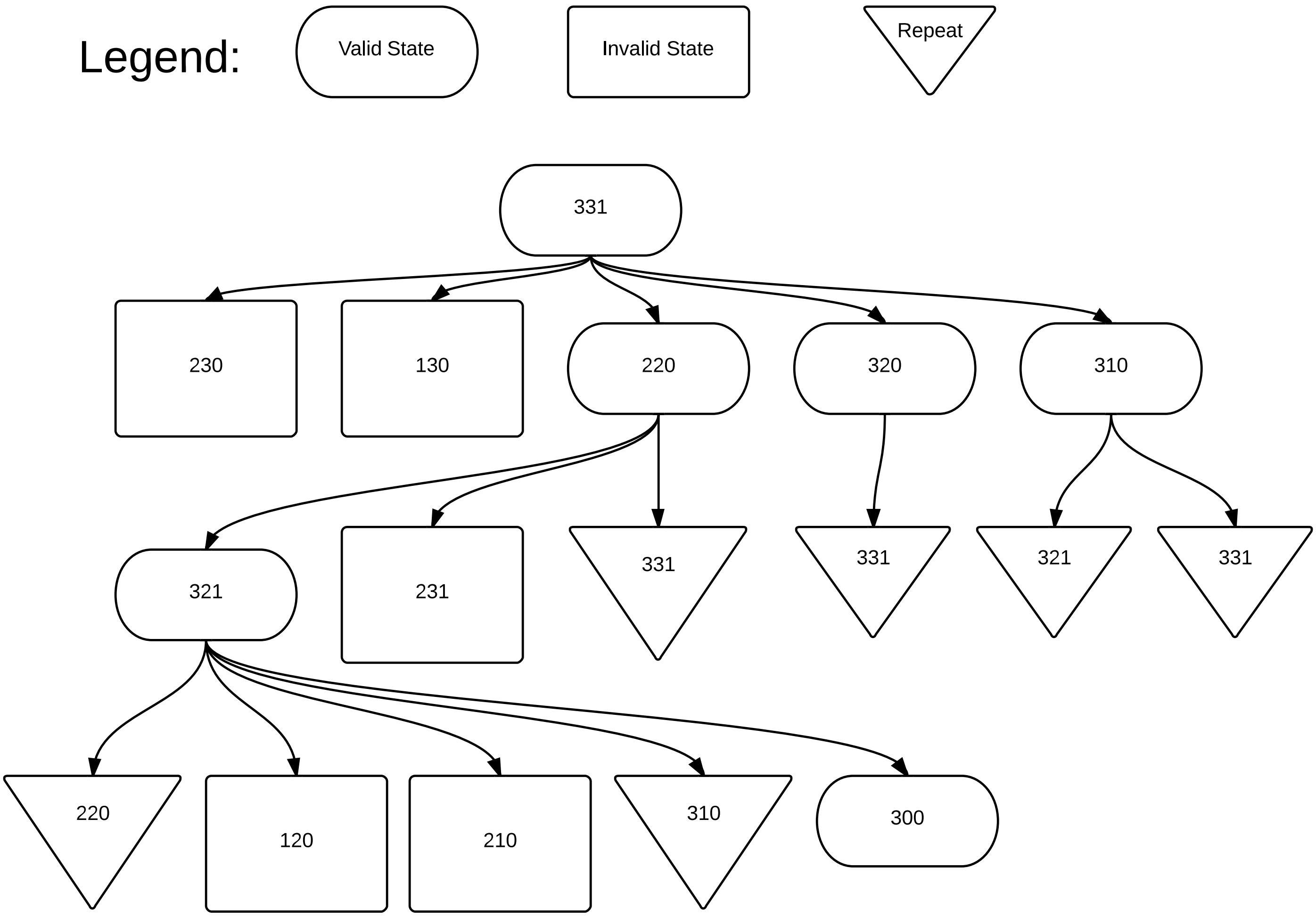
January 13, 2014

0.1 Introduction

In the Missionaries and Cannibals problem, we can represent the missionaries, cannibals and position of the boat using the notation (m, c, b). If boat has a value of 1, we assume it is on the starting side of the river. Thus, the initial problem state is represented as (3,3,1).

We can find an upper bound on the number of states by considering that for the number of missionaries and cannibals on the starting side of the river, this value must be between 0-4, inclusive. The boat must either be on the starting side or the other side (so only 2 options). We can multiply these options to arrive at $4 \cdot 4 \cdot 2 = 32$ total unique permutations of the number of states. This is the upper bound. More generally, give m missionaries and c cannibals, we can say the upper bound for the problem is $(m + 1) \cdot (c + 1) \cdot 2$ states.

A graphical representation of the graph of first few level of states can be found as Figure 1:


Legend:
Valid State
Invalid State
Repeat
331
230
130
220
320
310
321
231
331
331
321
331
220
120
210
310
300

0.2 Implementation of the model

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

```java
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]+","+poss
      } 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 si
  // 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

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

  // Goal node that completes the search
  UUSearchNode goalNode = null;
  UUSearchNode grandparent = null;

  // Initialize queue with start node
```

```
Queue<UUSearchNode> queue = new LinkedList<UUSearchNode>();
queue.add(startNode);
HashMap<Integer, UUSearchNode> visited = new HashMap<Integer, UUSearchNode>(

while(!queue.isEmpty()){
  updateMemory(queue.size() + visited.size());
  System.out.println("Queue: " + queue);
  UUSearchNode parentNode = (UUSearchNode) queue.remove();
  System.out.println("Using " + parentNode);

  incrementNodeCount();

  // check if the goal has been reached
  if (parentNode.goalTest()){
    goalNode = parentNode;
    int key = parentNode.hashCode();
    visited.put(key, grandparent);
    break;
  }

  ArrayList<UUSearchNode> children = parentNode.getSuccessors();
  for(int i=0; i < children.size(); i++){
    // if haven't seen before
    UUSearchNode childNode = children.get(i);
    int nodeKey = childNode.hashCode();
    System.out.println(nodeKey);
    if (!visited.containsKey(nodeKey)){
      // mark previous node
      if (!queue.contains(childNode)){
        queue.add(childNode);
      }
    } else {
      System.out.println("Already seen " + childNode);
    }
  }
  // visited 'key' will contain the hashCode of the node
  // visited 'value' will contain the node's predecessor node
  // startNode has no predecessor
  int key = parentNode.hashCode();
  visited.put(key, grandparent);

  // once all children have been added, mark this node visited
  grandparent = parentNode;

}


System.out.println(goalNode);

// Check if goalNode has been found
```

```
    if (goalNode == null){
      return null;
    } else {
      // return the backchain link
      return backchain(goalNode, visited);
    }
  }

  // backchain should only be used by bfs, not the recursive dfs
  private List<UUSearchNode> backchain(UUSearchNode goalNode,
      HashMap<Integer, UUSearchNode> visited) {
    List<UUSearchNode> retArr = new ArrayList<UUSearchNode>();
    while (goalNode != null){
      retArr.add(goalNode);
      goalNode = visited.get(goalNode.hashCode());
    }
    return retArr;
  }
```

We implement BFS with a queue and a HashMap. The queue is FIFO and allows us to dequeue the startNode and enqueue its successors, then dequeue the first level of the startNode's successors and enqueue *their* successors etc. By nature of BFS, we successively check each depth level from the startNode. We also avoid duplicates by checking if the queue already contains some node.

The HashMap visited will keep track of which nodes have already been visited by using the provided *hashCode* function. The key of the HashMap is the hashcode of the node. The value of the HashMap is the predecessor node that links to it in the graph. We track the memory by considering the size of the queue and the size of the *visited* HashMap at any point. Thus, when we find the goalnode, we can simply use a *while* loop to backchain from goalnode using the *visited* HashMap to find its predecessor.

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){
```

```java
            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 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
          // 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;
  }

}
```

We start by initializing the array which will be returned as the solution. We also initialize the HashMap which will mark whether the node has been visited and will have the key as the depth. We add the depth because, in some cases, if one sets a MAX_DEPTH for DFS, DFS may explore one path and exceed this max depth. Then, when DFS checks shorter paths, it may run into a duplicate. We want DFS to continue if this duplicate's depth is smaller than the original node's depth. Hence, we add the *visited.get*(*child*) > *depth* method in the recursive DFS method.

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.

We loop through each one of the $startNode$'s successors and then run the recursive DFS method on them. In the recursive DFS function, if the node has already been seen, we return *null*. If the node has not been seen, we run recursive DFS again on each of the node's successors.

When the goal node is finally reached, as said before, we add it to a list and return that list. Then in the recursive DFS function, if the returned list is NOT *null*, then we add the currently examined node to that list and return the expanded list. In this way, we build the solution from the ground-up, starting with the goal node and then progressively adding nodes that link to one another.

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

In Path-checking DFS, we ensure that the current path does not contain duplicates by using a set. As a result, we avoid infinite loops. Each time we're about to call *getSucessors*() on a child node, we check the set to ensure that it has not already been added.

This only ensures that there are no duplicates in the *current* path, but we could run into redundant nodes in other paths. The trade-off, however, is that we use less memory than simply memoizing everything.

0.6 Iterative deepening search

0.7 Lossy missionaries and cannibals