Missionaries and Cannibals Solution

Lexin Tang

September 20, 2016

1 Introduction

There is an upper bound on the number of states. For the (3 missionaries, 3 cannibals, 1 boat) problem, the total number of states is at most

$$4 * 4 * 2 = 32$$

To calculate the upper bound shown above, we examine all the possible numbers of missionaries, cannibals and boats in state representation. The number of missionaries on the starting bank could be 0, 1, 2, 3 (4 possible values) since there are 3 missionaries in total. Similarly, the number of cannibals could be 0, 1, 2, 3 (4 possible values) and the boat should be either 0 or 1 (2 possible values). Then we compute combinations of tuple (missionaries, cannibals, boats) by multiplying the number of possible values of each element.
In general, for a (x missionaries, y cannibals, z boat) problem, the upper bound on the number of states should be $(x + 1) * (y + 1) * (z + 1)$, regardless of legality of states.

Draw part of the graph of states in Figure 1.

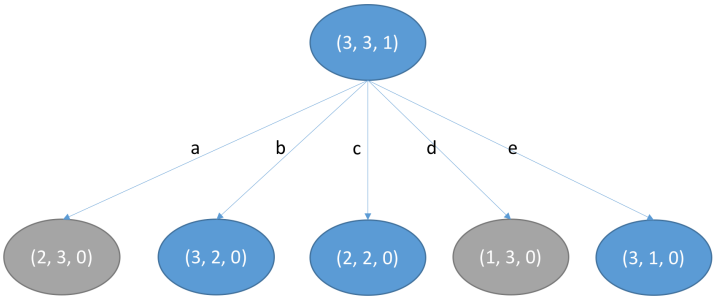

Figure 1: Part of the graph of states

In the graph above, the states in blue circle are legal and those in gray circle are illegal. We define a set of actions that transfer the initial state (3, 3, 1) to the first-reached states:
Action a: send one missionary across the river;
Action b: send one cannibal across the river;
Action c: send one missionary and one cannibal across the river;
Action d: send two missionaries across the river;
Action e: send two cannibals across the river.
Considering the state (2, 3, 0) and (1, 3, 0), the number of missionaries are smaller than that of cannibals on the starting bank, so the remaining missionaries on the starting bank will be eaten.

2 Implementation of the model

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

```
1   public ArrayList<UUSearchNode> getSuccessors() {
2     ArrayList<UUSearchNode> reachableStates = new ArrayList<UUSearchNode>();
3
4     // send i missionaries and j cannibals across the river
5     if (this.state[2] == 1){
6       for (int i = 0; i <= 2; i++)
7         for (int j = 0; j <= 2; j++){
8           if (i + j == 0 || i + j > 2){
9             continue;
10          }
11          if (this.state[0] - i >= 0 && this.state[1] - j >= 0 && (this.state[0] - i == 0
    || this.state[0] - i == totalMissionaries || (this.state[0] - i >= this.state[1] - j &&
    totalMissionaries - this.state[0] + i >= totalCannibals - this.state[1] + j))){
12            reachableStates.add(new CannibalNode(this.state[0] - i, this.state[1] - j, 0,
    depth + 1));
13          }
14      }
15    }
16    else {
17      // send i missionaries and j cannibals back to the starting shore
18      for (int i = 0; i <= 2; i++)
19        for (int j = 0; j <= 2; j++){
20          if (i + j == 0 || i + j > 2){
21            continue;
22          }
23          if (this.state[0] + i <= totalMissionaries && this.state[1] + j <=
    totalCannibals && (this.state[0] + i == 0 || this.state[0] + i == totalMissionaries || (
    this.state[0] + i >= this.state[1] + j && totalMissionaries - this.state[0] - i >=
    totalCannibals - this.state[1] - j))){
24            reachableStates.add(new CannibalNode(this.state[0] + i, this.state[1] + j, 1,
    depth + 1));
25          }
26      }
27    }
28    return reachableStates;
29  }
```

The basic idea of `getSuccessors` is to enumerate all legal actions and avoid illegal reachable states. In the 3 missionaries 3 cannibals problem, boat size is set to 2 so that we can send 0, 1 or 2 missionaries across the river. Similarly, we can also send 0, 1 or 2 cannibals across the river. But the total number of missionaries and cannibals sent across the river should not be greater than boat size. And if we send nobody to the boat, the state will keep unchanged so we should avoid this situation. That's why I use condition $(i + j == 0 || i + j > 2)$ to avoid illegal actions, where i missionaries and j cannibals are sent across the river.

To avoid illegal reachable state, we need to check both sides of river. If the number of missionaries are greater or equal to that of cannibals on the bank, missionaries are guaranteed to be safe. And if none of missionary are on the bank, nobody will be eaten, which is also a legal state.

Note that every action moves the boat to the other side of the river, b should flip between two connected states.

I write a piece of code to test my `getSuccessors()` in `CannibalDriver.java`.

```
1   // test getSuccessors() by showing the first-reached states
2   ArrayList<UUSearchNode> successors = mcProblem.startNode.getSuccessors();
3   for (int i = 0; i < successors.size(); i++){
```

```
        System.out.println(successors.get(i).toString());
    }
```

Output of the testing code is (3, 2, 0),(3, 1, 0),(2, 2, 0) for starting state (3,3,1), which agrees with those I drew in the introduction part.

3 Breadth-first search

The BFS is implemented in `UUSearchProblem.java`. Here's my code for `breadthFirstSearch`:

```
public List<UUSearchNode> breadthFirstSearch(){
  resetStats();
  HashMap<UUSearchNode, UUSearchNode> visited = new HashMap<UUSearchNode, UUSearchNode>();
  visited.put(startNode, null);

  Queue<UUSearchNode> queue = new LinkedList<UUSearchNode>();
  queue.offer(startNode);

  while (!queue.isEmpty()){
    int size = queue.size();
    for (int i = 0; i < size; i++){
      UUSearchNode node = queue.poll();
      incrementNodeCount();
      //System.out.println("polling " + node.toString());
      if (node.goalTest()){
        return backchain(node, visited);
      }

      ArrayList<UUSearchNode> nextNodeList = node.getSuccessors();
      for (int j = 0; j < nextNodeList.size(); j++){
        UUSearchNode nextNode = nextNodeList.get(j);
        //System.out.println("nextNode " + nextNode.toString());
        if (!visited.containsKey(nextNode)){
          queue.offer(nextNode);
          visited.put(nextNode, node);
          updateMemory(visited.size() + queue.size());
        }
      }
    }
  }

  return null;
}
```

The basic idea of `breadthFirstSearch` is to use a queue for storing all frontiers and use a hash table or hash set to mark the nodes which have been visited before.
My implementation of backchaining is shown as follows:

```
private List<UUSearchNode> backchain(UUSearchNode node,
  HashMap<UUSearchNode, UUSearchNode> visited) {
  List<UUSearchNode> result = new ArrayList<UUSearchNode>();
  UUSearchNode prevNode = visited.get(node);
  result.add(node);
  while (prevNode != null){
    node = prevNode;
    result.add(node);
    prevNode = visited.get(node);
  }

```

```
    return reverseList(result);
  }
```

Since backchaining outputs the exploring sequence of nodes reversely, I write a function `reverseList()` to get the actual sequence of searching.

```
  private List<UUSearchNode> reverseList(List<UUSearchNode> result){
    // reverse list of UUSearchNodes
    int i = 0;
    int j = result.size() - 1;
    while (i <= j){
      UUSearchNode temp = result.get(i);
      result.set(i, result.get(j));
      result.set(j, temp);
      i++;
      j--;
    }

    return result;
  }
```

The output for 3 missionaries, 3 cannibals problem:
bfs path length: 12 [(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: 15
Maximum memory usage during last search 17

4 Memorizing depth-first search

My implementation of memorizing depth-first search is shown as follows.

```
  private List<UUSearchNode> dfsrm(UUSearchNode currentNode, HashMap<UUSearchNode, Integer>
    visited, int depth, int maxDepth) {

    visited.put(currentNode, depth);

    // keep track of stats; these calls charge for the current node
    updateMemory(visited.size());
    incrementNodeCount();

    if (depth > maxDepth){  // base case
      return null;
    }

    if (currentNode.goalTest()){
      // base case
      List<UUSearchNode> result = new ArrayList<UUSearchNode>();
      result.add(currentNode);
      return result;
    }

    ArrayList<UUSearchNode> nextNodeList = currentNode.getSuccessors();
    for (int i = 0; i < nextNodeList.size(); i++){
      UUSearchNode nextNode = nextNodeList.get(i);
      if (!visited.containsKey(nextNode)){
        List<UUSearchNode> result = dfsrm(nextNode, visited, depth + 1, maxDepth);
        if (result != null){
          // recursive case
```

```
            result.add(currentNode);
            return result;
          }
        }
      }

      return null;
    }
```

The base case of the problem is in which goalTest() passes or depth exceeds maxDepth. Base case and recursive case are labeled in the code above.

Memorizing dfs output for 3 missionaries, 3 cannibals problem:
dfs memorizing path length:12 [(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: 13
Maximum memory usage during last search 13

Memoizing dfs doesn't save significant memory with respect to breadth-first search because both of them maintain a hash table of hash set to store visited nodes.

5 Path-checking depth-first search

My implementation of Path-checking depth-first search is shown as follows.

```
1   private List<UUSearchNode> dfsrpc(UUSearchNode currentNode, HashSet<UUSearchNode>
      currentPath, int depth, int maxDepth) {
2     if (depth > maxDepth){
3       return null;
4     }
5
6     currentPath.add(currentNode);
7     // keep track of stats; these calls charge for the current node
8     updateMemory(currentPath.size());
9     incrementNodeCount();
10
11    if (currentNode.goalTest()){
12      List<UUSearchNode> result = new ArrayList<UUSearchNode>();
13      result.add(currentNode);
14      return result;
15    }
16
17    ArrayList<UUSearchNode> nextNodeList = currentNode.getSuccessors();
18    for (int i = 0; i < nextNodeList.size(); i++){
19      UUSearchNode nextNode = nextNodeList.get(i);
20      if (!currentPath.contains(nextNode)){
21        List<UUSearchNode> result = dfsrpc(nextNode, currentPath, depth + 1, maxDepth);
22        if (result != null){
23          result.add(currentNode);
24          return result;
25        }
26      }
27    }
28
29    currentPath.remove(currentNode);
30    return null;
31  }
```

Path-checking dfs output for 3 missionaries, 3 cannibals problem:
dfs path checking path length:12 [(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: 13
Maximum memory usage during last search 12

Path-checking depth-first search saves significant memory with respect to breadth-first search because path-checking dfs only save current path, which is at most the length of the longest path to goal, while bfs stores all visited nodes which could be the total number of existing nodes.

Here is an example of a graph where path-checking dfs takes much more run-time than breadth-first search.

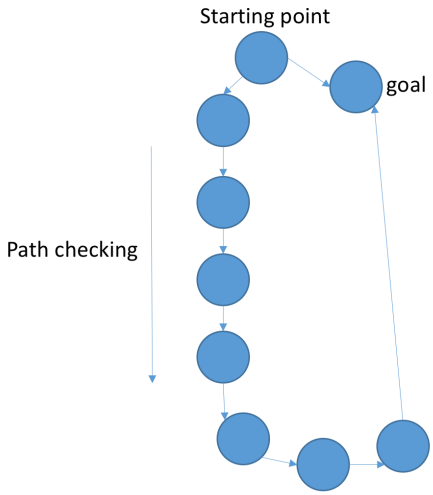

Figure 2: path-checking dfs takes much more run-time than breadth-first search

6 Iterative deepening search

My implementation of Iterative deepening search is shown as follows.

```
1   public List<UUSearchNode> IDSearch(int maxDepth) {
2     resetStats();
3     List<UUSearchNode> result = new ArrayList<UUSearchNode>();
4     HashSet<UUSearchNode> currentPath = new HashSet<UUSearchNode>();
5     for (int i = 0; i <= maxDepth; i++){
6       result = dfsrpc(startNode, currentPath, 0, i);
7       if (result != null){
8         return reverseList(result);
9       }
10    }
11    return null;
12  }
```

Iterative deepening output for 3 missionaries, 3 cannibals problem:
Iterative deepening (path checking) path length:12 [(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: 162
Maximum memory usage during last search 12

Discussion:
For better memory-efficiency, I would prefer path-checking dfs because path checking uses less memory with respect to memorizing dfs.
For better runtime-efficiency, I would prefer memorizing dfs because path checking dfs takes much longer time to find a goal if certain node is searched more than once, which could be very common when searching on a graph.

7 Lossy missionaries and cannibals

The state should contain the number of missionaries on the starting bank, the number of cannibals on the starting bank, where the boat is, and the number of missionaries eaten by cannibals.

I would change my `getSuccessors()` because some illegal states in the original problem now becomes legal if at most E missionaries could be eaten. And I also want to modify `goalTest()` since the goal of the problem is changed. `UUSearchProblem.java` can remain unchanged in this case.

In general, for a (x missionaries, y cannibals, z boat) problem, at most x missionaries could be eaten. So the upper bound of possible states for this problem should be $(x+1)*(y+1)*(z+1)*(x+1)$, regardless of legality of states.