Missionaries and Cannibals

This is my report for Homework 1. Please be gentle

1 Introduction

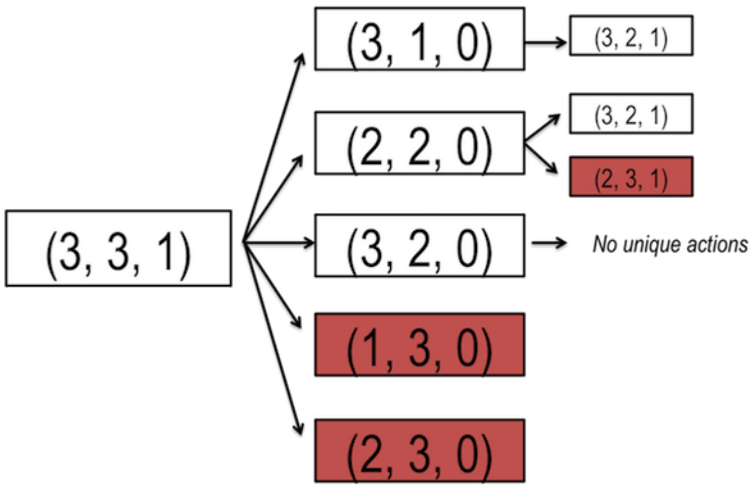
1.1 General Information on the Problem

The setting of the Missionaries and Cannibals problem is a river where there is a boat on one side and various missionaries and cannibals on either side. We represent the problem by (a,b,c,d,e,f), where a, b, and c are, respectively, the number of missionaries, the number of cannibals, and boat position (0 or 1) at the starting position. d, e, and f are similarly those of the goal position. The problem asks for a solution to the question: What sequence of steps do I need to take to get from the beginning state to the goal state? (Here state means the three integers taken as an ordered triple). From here on in, we abbreviate a certain problem as an (a,b,c) problem, although we actually are referring to (a, b, c, 0, 0, 0). We attempt to solve problems of this form with various graph search algorithms.

1.2 Bounds on Number of States

Without considering the legality of states, we can come up with an upper bound on the number of states with a counting argument. With m missionaries, c cannibals, and two boat positions, we have $2(m+1)(c+1)$ configurations. This is because we can have 0, ..., $(m-1)$, m missionaries on one side and 0, ..., $(c-1)$, c missionaries on the same side. The other side can be determined from the first side, and so this is a definite upper bound, although very naive in that legality is not considered at all. If we start from the state (3,3,1), this upper bound is $4 * 4 * 2 = 32$.

1.3 Diagram of States



The diagram of states above shows the illegal (red) actions and legal (white) actions starting with the state (3,3,1). Note that each of the three legal actions of (3,3,1) will themselves have an action changing them back to the (3,3,1) state. Thus, (3,3,1) is considered "non-unique" and not repeated. The actions available at illegal states are not displayed, for they are not reachable.

2 Code Structure

The code involves two classes: UUSearchProblem and CannibalProblem. The former aids in solving a generic search problem and contains the implementation of various search algorithms, which are all analyzed in the next section. The latter extends UUSearchProblem, and, by doing so, specifies exactly what the "Missionaries and Cannibals" problem entails. For instance, CannibalProblem contains important methods like goalTest and getSuccessors, which are vital in determining when the search is complete and finding valid configurations to pursue, respectively.

The next snippet is hard to format well in TeX.

```java
public ArrayList<UUSearchNode> getSuccessors() {
    ArrayList<UUSearchNode> list = new ArrayList<>();

    if(state[2] == 1){//boat on one side
    for(int c = 0; c <= (BOAT_SIZE < state[1] ? BOAT_SIZE : state[1]); c++)//# cannibals equal to the smaller of these
        for(int m = 0; m <= ((state[0] > (BOAT_SIZE - c) ? (BOAT_SIZE - c) : state[0])); m++){
            if((m + c) != 0 && //need someone in boat
            (m >= c || m == 0) //cannibals cannot outnumber missionaries
            && isSafeState(state[0] - m, state[1] - c))//Don't want more cannibals than missionaries on boat
                list.add(new CannibalNode(state[0] - m, state[1] - c, 0, depth));
                }
        } else {//boat on the other side, everything analagous
            for(int c = 0; c <= (BOAT_SIZE < (totalCannibals - state[1]) ? BOAT_SIZE : (totalCannibals - state[1])); c++)
                for(int m = 0;
                m <= ((totalMissionaries - state[0]) > (BOAT_SIZE - c) ? (BOAT_SIZE - c) : (totalMissionaries - state[0]));
                m++){
                        if((m + c) != 0 && (m >= c || m == 0) && isSafeState(state[0] + m, state[1] + c))
                            list.add(new CannibalNode(state[0] + m, state[1] + c, 1, depth));
                }
        }
        return list;
}
```

getSuccessors() uses the method isSafeState() to determines whether an acceptable number of missionaries are on each side of the river, with respect to the cannibal populations. Some of the convolution in the above method comes from the fact that this code is written to run on any boat size such that no missionaries will be eaten on the boat. See (5). Essentially, this method first checks which side the boat is on, iterates through all compatible combinations of cannibals and missionaries, and then adds a node to a list if the state is a legal one. In the most nested if-statements, the boat is checked to have no more cannibals than missionaries (unless there are only cannibals on board) and to have at least one person on board. In such a manner, every state is assigned a set of other states, each of which the state can legally transition to.

3 Implementation of Searches

3.1 Output of CannibalDriver on (8,5,1) Problem

```
bfs path length: 24 [0m, 0c, and 0b., 1m, 1c, and 1b., 0m, 1c, and 0b., 2m, 1c, and 1b., 1m, 1c, and 0b., 3m, 1c, and 1b.,
2m, 1c, and 0b., 3m, 2c, and 1b., 2m, 2c, and 0b., 4m, 2c, and 1b., 3m, 2c, and 0b., 4m, 3c, and 1b., 3m, 3c, and 0b.,
 5m, 3c, and 1b., 4m, 3c, and 0b., 5m, 4c, and 1b., 4m, 4c, and 0b., 6m, 4c, and 1b., 5m, 4c, and 0b.,6m, 5c, and 1b.,
 5m, 5c, and 0b., 7m, 5c, and 1b., 6m, 5c, and 0b., 8m, 5c, and 1b.]
Nodes explored during last search: 61
Maximum memory usage during last search 62
---------
dfs memoizing path length:34
Nodes explored during last search: 35
Maximum memory usage during last search 35
---------
dfs path checking path length:34
Nodes explored during last search: 35
Maximum memory usage during last search 34
---------
Iterative deepening (path checking) path length:24
Nodes explored during last search: 337752
Maximum memory usage during last search 24
```

3.2 Breadth-First Search

The BFS algorithm implemented here is unlike the typical BFS in that it involves a back-chaining mechanism. That is, a HashMap is created to map a state to an immediately preceding state. In this way, the path to the beginning configuration can iteratively be re-created once the goal is reached.

```java
public List<UUSearchNode> breadthFirstSearch() {
        Queue<UUSearchNode> queue = new LinkedList<>();
        HashMap<UUSearchNode, UUSearchNode> map = new HashMap<>();
        UUSearchNode goal = null;

        resetStats();
        queue.add(startNode);
        map.put(startNode, startNode);//just mapping startNode to some non-null value; not especially important

        bfsLoop:
        while(!queue.isEmpty()){
                UUSearchNode node = queue.remove();
                updateMemory(map.size());
                incrementNodeCount();

                if(node.goalTest()){//goal found
                        goal = node;
                        break bfsLoop;
                } else {
                        for(UUSearchNode n : node.getSuccessors()){
                                if(map.get(n) == null){//not visited
                                        map.put(n, node);
                                        queue.add(n);
                        }
                    }
                }
        }
        return backchain(goal, map);
}
```

```
private List<UUSearchNode> backchain(UUSearchNode node, HashMap<UUSearchNode, UUSearchNode> visited) {
        if(node == null) return null;

        ArrayList<UUSearchNode> list = new ArrayList<>();
        UUSearchNode curr = node;

        list.add(curr);

        while(!curr.equals(startNode)){
                curr = visited.get(curr);
                list.add(curr);
        }
        return list;
}
```

This implementation has the following output for the (3,3,1) case.

```
        bfs path length: 12 [0m, 0c, and 0b., 1m, 1c, and 1b., 0m, 1c, and 0b., 0m, 3c, and 1b., 0m, 2c, and 0b.,
 2m, 2c, and 1b., 1m, 1c, and 0b., 3m, 1c, and 1b., 3m, 0c, and 0b., 3m, 2c, and 1b., 2m, 2c, and 0b., 3m, 3c, and 1b.]
Nodes explored during last search: 15
Maximum memory usage during last search 15
```

One further test involves confirming that the initial diagram of states of (3,3,1) is correct. These legal states can be found by running the following code and receiving the following output.

```
public void testSuccessors(){
        System.out.println("Successors␣of␣" + this + ":␣" + getSuccessors());
        System.out.println("==============");
        for(UUSearchNode n : getSuccessors()){
                System.out.println("Successors␣of␣" + n + ":␣" + n.getSuccessors());
        }
}
```

Output:

```

Successors of 3m, 3c, and 1b.: [3m, 2c, and 0b., 2m, 2c, and 0b., 3m, 1c, and 0b.]
==============
Successors of 3m, 2c, and 0b.: [3m, 3c, and 1b.]
Successors of 2m, 2c, and 0b.: [3m, 2c, and 1b., 3m, 3c, and 1b.]
Successors of 3m, 1c, and 0b.: [3m, 2c, and 1b., 3m, 3c, and 1b.]
```

3.3 Memoizing Depth-First Search

By storing all visited states, DFS with memoization does not visit the same state twice. While this prevents the algorithm from unnecessarily repeating work that has already been done, it also means that a considerable number of states must be stored. As discussed in class, this algorithm is almost always inferior to BFS.

```
	public List<UUSearchNode> depthFirstMemoizingSearch(int maxDepth) {
		HashMap<UUSearchNode, Integer> visited = new HashMap<>();
		resetStats();

		return dfsrm(startNode, visited, 0, maxDepth);
	}

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

		if(currentNode.goalTest()){ // base case: goal found
			List<UUSearchNode> list = new ArrayList<>();
			list.add(currentNode);
			visited.put(currentNode, depth);
			updateMemory(visited.size());
			return list;
		} else if (depth < maxDepth){ //recursive case
			visited.put(currentNode, depth);
			updateMemory(visited.size());

			for(UUSearchNode n : currentNode.getSuccessors()){
				if(!visited.containsKey(n)){
					List<UUSearchNode> l = dfsrm(n, visited, depth + 1, maxDepth); //recurse furthur
					if(l != null){
						l.add(currentNode);
						return l;
					}
				}
			}
		}
		return null;
	}
```

(3.1) shows that DFS with memoizing has a maximum memory usage nearly half of BFS with memoizing. However, as shown in class, BFS is asymptotically better. Perhaps the di

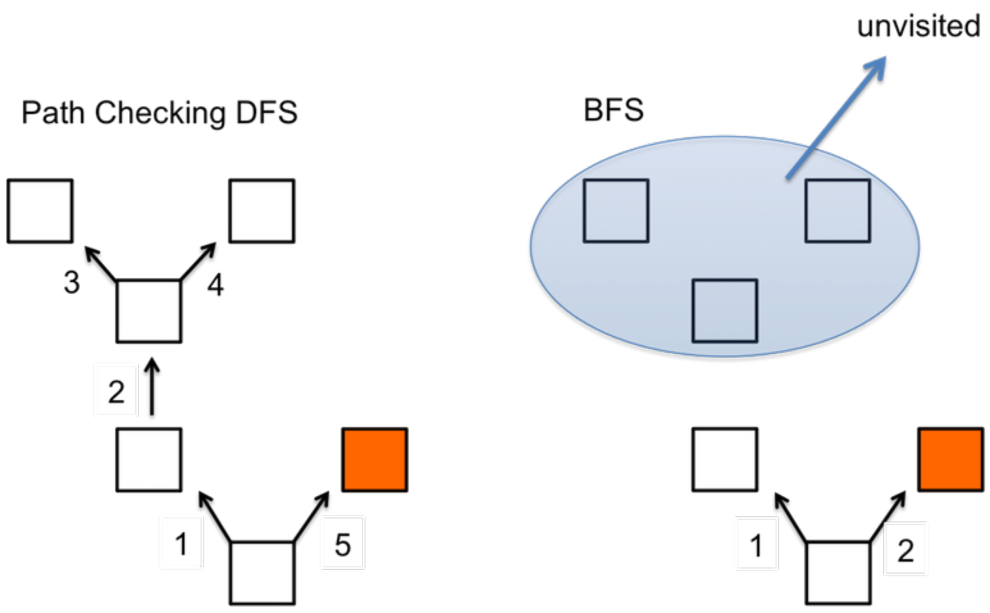
3.4 Path Checking Depth-First Search

In contrast to DFS with memoization, path checking DFS attempts to significantly cut down on what is stored in memory at any given time. It has an O(m) space complexity, where m is the maximal path length. A con, however, is that when a new path leads to a node that has already been visited via a former path, the node is visited again.

```
	public List<UUSearchNode> depthFirstPathCheckingSearch(int maxDepth) {
		resetStats();
		HashSet<UUSearchNode> currentPath = new HashSet<UUSearchNode>();

		return dfsrpc(startNode, currentPath, 0, maxDepth);

```

```
    }

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

        incrementNodeCount();

        if(currentNode.goalTest()){//base case: goal found
            List<UUSearchNode> list = new ArrayList<>();
            currentPath.add(currentNode);
            updateMemory(currentPath.size()); //memory updated whenever path changes
            list.add(currentNode);


            return list;
        } else if(depth < maxDepth){//recursive case
            currentPath.add(currentNode);
            updateMemory(currentPath.size());

            for(UUSearchNode n : currentNode.getSuccessors()){
                if(!currentPath.contains(n)){
                    List<UUSearchNode> list = dfsrpc(n, currentPath, depth + 1, maxDepth);//recurse

                    if(list != null){
                        list.add(currentNode);
                        return list;
                    }
                }
            }
            currentPath.remove(currentNode); //remove node when goal not found on current path
        }
        return null;
    }
```

(3.1) suggests path checking DFS is more memory efficient than BFS. However, in terms of time, the figure below may help to elucidate why BFS is often better at finding the goal (red box).



Like all DFS algorithms, path checking DFS might come up short, even when the goal is relatively close by.

3.5 Iterative Deepening Search

IDS combines the ability of BFS to quickly find nearby goals with the memory efficiency of path checking DFS. From (3.1), it is clear that IDS explores many more states than the other search algorithms, but it is also most memory efficient, and thus optimal. The simple code for IDS calls upon the path checking DFS for incrementally increasing depths.

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

        for(int i = 0; i <= maxDepth; i++){
            List<UUSearchNode> l = depthFirstPathCheckingSearch(i);

            if(l != null)
                return l;
        }

        return null;
    }
```

In (3.1), IDS achieves the shortest path, as it is identical to the result of the BFS algorithm. As discussed in class, DFS with memoizing always falls short of BFS in both time and space complexity. DFS with path checking has a much lower space complexity, requiring just O(m) space, where m is the maximal path length. The latter beats BFS in terms of space complexity, although not necessarily for time complexity. Nonetheless, it is better that DFS with path checking be used in IDS, rather than DFS with memoization.

4 Discussion of Lossy Missionaries and Cannibals

The state for this new problem would be similar to our problem in that we would need to keep track of the number of missionaries and cannibals on a given side of the river and where the boat is located. However, we additionally need to know how many missionaries have been eaten. Otherwise, we will not be able to deterministically figure out how many of whom are on the other side. In order to implement this, we would need to change our getSuccessors() method to include states where missionaries get eaten. We would, of course, need to make sure that we do not include a state where more than E missionaries are eaten. Then, the problem can be tackled through the same method of graph searches. To construct an upper bound, it must first be noticed that there are still (c+1) arrangements of cannibals on either side of the river. Secondly, it is seen that when 0 missionaries are eaten, there are (m+1) arrangements of missionaries, and when 1 missionary is eaten, there are m arrangements, and so on. Finally, there are the two choices of where the boat is.

Thus, an upper bound (that does not even consider the legality of states) would be $2(c+1)\sum_{n=m+1-E}^{m+1} n = 2(c+1)(m+1)(E+1) - E(E+1)(c+1) = (E+1)(c+1)(2m+2-E)$. Notice that for E = 0, the upper bound for the original problem is obtained.

5 Extension of Assignment

The Missionaries and Cannibals problem can be generalized/extended in many different ways and what follows is a description of just one. In the original problem, there is no need to worry about whether cannibals will eat missionaries in the boat, as there will always be at most one cannibal per missionary, when missionaries are present. This is a basic fact for a boat of size 2. To take care of the cases when the boat size is larger than 2, getSuccessors() must be generalized. Firstly, the different numbers of cannibals which can be placed in the boat are determined. These numbers are bounded above by the smaller of the boat size and the number of cannibals, hence the presence of the ternary operator. Then, all numbers of missionaries are considered. Clearly, this number must not be less than the number of cannibals present, unless zero missionaries are on board, which is one of the conditions in the if-statement. For code, see (2).