Missionaries and Cannibals Solution

Boying Shi

January 14, 2014

1 Introduction

The mission is to solve the problem of `N` missionaries and `N` cannibals using one boat to across a river without any missionaries eaten. To solve this problem, the problem should be modeled as states and actions. States are represented as a tuple, for example if there are three missionaries, three cannibals and one boat are on the starting bank, the initial state is `(3,3,1)`. Actions links states, it is how to move the missionaries and cannibals from either side of the river.

To find the solution, states should be figured out as safe state or unsafe state. In order to find a path to the goal state `(0,0,0)`, a graph should be created which states represent the nodes and actions represent the links. Only the safe state can be added to the graph and then applying different graph searching method can find a path to goal state, therefore, the solution is found.

The model is implemented in `Java` and using `breadth-first search`, `memorizing depth-first search`, `path-checking depth-first search` and `iterative deepening search` to get solution of the model. The solution includes: the length of the path and each state on the path. In order to compare those different searching method, the nodes explored and the maximum memory usage in each search are recorded.

1.1 Question

1.1.1 Question one

For question "Give an upper bound on the number of states, without considering legality of states and Describe how you got this number."

My answer is that the upper bound on the number of states with an initial state (3, 3, 1) is 32. The way to get the number is to multiple the possible number of missionaries and cannibals and boat together. For example, the number of missionaries can be 0, 1, 2, 3 and the number of cannibals can be 0, 1, 2, 3 and the number of boat can be 0, 1, then 4*4*2 = 32.

1.1.2 Question two

For question "draw part of the graph of states, including at least the first state, all actions from that state, and all actions from those first-reached states. Show which of these states are legal and which aren't."

In order to demonstrate the model more clearly, I use a graph to show the path finding for problem of three missionaries, three cannibals and one boat.

As I mentioned before, the initial state is `(3,3,1)`. Since there are at most 2 people on the boat at one time, there are five possible actions (`(1,0,1)`, `(2,0,1)`, `(0,1,1)`, `(0,2,1)`, and `(1,1,1)`).

When the boat is at the starting bank, the current state should minus all these five possible actions to get states without considering legality.

When the boat is at the goal bank, then the current state should add all these five possible actions to get possible successors.

Firstly, the boat will carries people across the river, so the initial state should minus those five possible actions and then get 5 possible states including safe and unsafe. The procedure of this is shown as a graph

below, the initial state is (3,3,1) and five possible actions are verb‘(1,0,1)‘, `(2,0,1)`, `(0,1,1)`, `(0,2,1)`, and `(1,1,1)`. After the first move, the start node has five possible successors: verb‘(2,3,0)‘, `(1,3,0)`, `(3,2,0)`, `(3,1,0)`, and `(2,2,0)`. The first two nodes are unsafe nodes and I mark them as red and the last three nodes are safe nodes and I mark them as green.

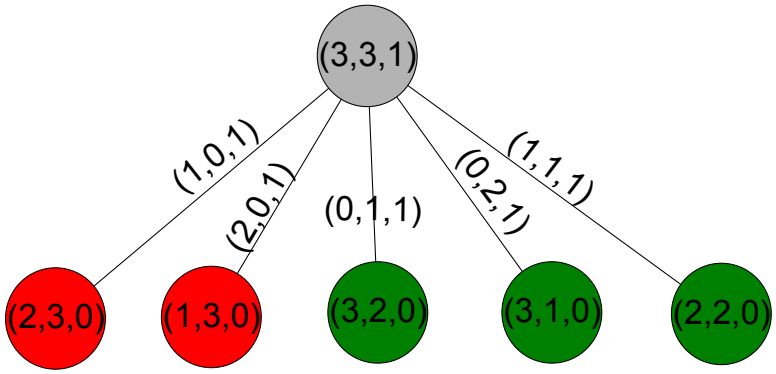

Figure 1: States

2 Implementation of the model

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

Listing 1: getSuccessors

```
    public ArrayList<UUSearchNode> getSuccessors() {

       ArrayList<UUSearchNode> successors = new ArrayList<UUSearchNode>();

        if (state[2] == 1){//the boat is at the starting side
           for(int i = 0; i < totalMissionaries; i++){
               for(int j = 0; j < totalCannibals; j++){
                CannibalNode c = new CannibalNode(state[0]-i,
                                                  state[1]-j,
                                                  state[2]-1,
                                                  depth+1);
                addSafeState(successors, c, i, j);
                    }
                }
           }else if(state[2] == 0){//when the boat is at the goal side
              for(int i = 0; i < totalMissionaries+1;i++){
                  for(int j = 0; j < totalCannibals+1; j++){
                CannibalNode c = new CannibalNode(state[0]+i,
                                                  state[1]+j,
                                                  state[2]+1,
                                                  depth+1);
                addSafeState(successors, c, i, j);

                                    }
                            }
                }
```

```
            return successors;
        }
```

The basic idea of `getSuccessors()` is to get all child nodes(successor states) of a node(current state). To implement this idea, I consider two conditions: one is the boat is at starting bank which `state[2]` equals to 1 and the other one is the boat is at goal bank which `state[2]` equals to 0.

When the boat is at the starting side, it need to carry people across the river to the goal side which causes the number of missionaries and cannibals to decrease in a safe way. In order to get all possible safe successor states of current state, I traverse all possible states and check them so as to add those safe states into an `ArrayList successors`.

When the boat is at the goal side, it only can take people back to the starting side which make the number of missionaries and cannibals to increase in a legal method. The implementation is also to traverse all possible states and choose those legal ones to store into `ArrayList successors`.

In my design, I use a method called `addSafeState()` to add legal states into `ArrayList successors`. Here's my code for `addSafeState()`:

Listing 2: addSafeState

```
    private void addSafeState(ArrayList<UUSearchNode> successors,
                              CannibalNode newState, int i, int j) {
      if (newState.isSafeState(i,j)) {
        successors.add(newState);
        }
      }
```

In `addSafeState()`, it firstly checks if a node is safe state or not by calling a method named `isSafeState()`. `isSafeState()` is basically designed to check if a state is legal or not.Then, if the node is safe, `addSafeState()` will add the node to the `ArrayList successors`. And my code for `isSafeState()` is shown below:

Listing 3: isSafeState

```
    private boolean isSafeState(int i, int j) {
      if( i+j >= 1 &&
          i+j <= BOAT_SIZE &&
          state[0] >= 0 &&
          state[1] >= 0 &&
          totalMissionaries - state[0] >= 0 &&
          totalCannibals - state[1] >=0){
          if(state[0] == 0 &&
             totalMissionaries - state[0] >= totalCannibals - state[1]){
            return true;
            }
          if(totalMissionaries - state[0] == 0 &&
             state[0] >= state[1]){
            return true;
            }
         if(state[0] >= state[1] &&
            (totalMissionaries - state[0]) >= (totalCannibals - state[1])){
            return true;
             }
            }
          return false;
          }
```

I use a method `isSafeState()`, it firstly checks the prerequisites for a safe node, there are six prerequisites and they are: (1) the number of people on the boat must greater than or equal to 1; (2) the number of people one the boat must smaller than or equal to the size of boat; (3) the number of missionaries on the starting side must greater or equal to 0; (4) the number of cannibals on the starting side must greater or equal to 0; (5) the number of missionaries on the goal side must greater or equal to 0; (6) the number of cannibals on the goal side must greater or equal to 0.

After all the six prerequisites are satisfied, `isSafeState` returns `true` if any one of the three conditions below is satisfied: (1) if the number of missionaries on the starting side is 0, then on the goal side, the number of missionaries is great than the number of cannibals; (2) if the number of missionaries on the goal side is 0, then on the starting side, the number of missionaries is great than the number of cannibals; (3) the number of missionaries must greater than or equal to the number of cannibals either on the starting side or on the goal side.

Moreover, since the solution path is stored as `List<UUSearchNode>`, method `ToString()`need to be rewrite in order to print out each state tuple on the solution path. The code of `ToString()` is as followed:

Listing 4: ToString

```
@Override
public String toString() {
  return Arrays.toString(state);
  }
```

Also, the mode needs to check condition to find if the problem reaches to goal state `(0,0,0)`. Therefore, I write method `goalTest()` to help my program to check if it has reach the goal state or not. If the program can reach the goal state, then it can return the solution path which means the solution path has been found. And if the program cannot reach the goal state, it means there is no solution path.

My code of `goaltest()` is here:

Listing 5: goaltest

```
@Override
public boolean goalTest() {
return ((state[0] == 0 && state[1] == 0 && state[2] == 0) ? true: false);
              }
```

3 Breadth-first search

The main idea of `Breadth-first search` is to create an undirected graph which the nodes on the graph are all safe states and the edges are all actions and apply `BFS` method to search the graph and find the solution path and return the path.

3.1 Implementation

Specifically, my implementation of `Breadth-first search` need a `Queue` data structure to store the node to be explored and a `HashMap` data structure to record nodes that have been explored. Since `HashMap` has a feature of `key` and `value`, then each node that has just been explored can be store as a `key` while its parent explored node can be store as a `value`, in this way, node that is explored latter and be linked to node that is explored former. Then, by using a method called `backchain` could help to return the right order of the solution path, from start node to goal node.

At the same time, in order to have comparisons with the following three version of DFS about the total number of nodes that have been explored and the maximum memory used to explore node, method `incrementNodeCount()` is called each time a node has been visited (just follow each `visited.put()`) to record

the total number of explored nodes and method `updateMemory()` is called each time there is a memory change to record the maximum memory.

Here's my code for `Breadth-first search`:

Listing 6: breadthFirstSearch

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

   Queue<UUSearchNode> frontier = new LinkedList<UUSearchNode>();
   frontier.offer(startNode);//add initial state date structure to frontier
   updateMemory(frontier.size()) ;

   List<UUSearchNode>  solution = new ArrayList<UUSearchNode>();
   HashMap<UUSearchNode,UUSearchNode> visited =
        new HashMap<UUSearchNode,UUSearchNode>();
   visited.put(startNode, null);
   incrementNodeCount();
   updateMemory(visited.size()+frontier.size()) ;

   while (!frontier.isEmpty()) {//while frontier is not empty
      //get current_node from the frontier
      UUSearchNode current_node = frontier.poll();
      //get current_state from current_node
      ArrayList<UUSearchNode> current_state =
         current_node.getSuccessors();
      updateMemory(visited.size()+frontier.size()+current_state.size()) ;

         if(current_node.goalTest()){
           //backchain from current_node and return solution
           solution = backchain(current_node,visited);
           return solution;
             }

  for (UUSearchNode child : current_state) {
    if (!visited.containsKey(child)) {
        visited.put(child,current_node);
        incrementNodeCount();
        frontier.add(child);
        updateMemory(visited.size()+frontier.size());
         }
       }
     }
     return null; // failure
   }
```

Method `backchain` is used to find the right order of solution path. Here's my code for `backchain`:

Listing 7: backchain

```
private List<UUSearchNode> backchain(UUSearchNode node,
  HashMap<UUSearchNode, UUSearchNode> visited) {
    List<UUSearchNode> path = new ArrayList<UUSearchNode>();
    //path.add(0,node) means store node to position 0
```

```
        //firstly, store goal node to path
        path.add(0,node);
        //startNode has value of null
        //find the node has value of startNode can finish adding path
        while(!node.equals(startNode)){
          if(visited.containsKey(node)){
            //key records the latter state and value records the former state
            node = visited.get(node);
            path.add(0,node);
            }
          }
            System.out.println("number of nodes on path " + path.size());
            return path;
        }
```

3.2 Test Result

To test `Breadth-first search` , I used two cases: `(3,3,1)` and `(8,5,1)`. The "Nodes explored during last search" keeps track of the entire nodes that have been visited during the search. And the "Nodes explored during last search" records the maximum memory to store explored nodes plus the maximum memory of the `queue` data structure.

The result of `(3,3,1)` has following results:

Listing 8: BFS on (3,3,1)

```
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 18
```

The total number of nodes that have been explored during the entire BFS search is 15 and the number of nodes on the solution path is 12. The 12 states have been shown on my test result and the 3 more states which have been explored but not on the solution path during BFS and the maximum memory usage is 18. And the result of `(8,5,1)` is as shown below:

Listing 9: BFS on (8,5,1)

```
bfs path length:  24 [[8, 5, 1], [8, 3, 0], [8, 4, 1], [8, 2, 0], [8, 3, 1],
    [6, 3, 0], [6, 4, 1], [5, 3, 0], [5, 4, 1], [5, 2, 0], [5, 3, 1], [4, 2,
    0], [4, 3, 1], [4, 1, 0], [4, 2, 1], [3, 1, 0], [3, 2, 1], [3, 0, 0], [3,
    1, 1], [2, 0, 0], [2, 1, 1], [1, 0, 0], [1, 1, 1], [0, 0, 0]]
Nodes explored during last search:  62
Maximum memory usage during last search 68
```

The total nodes explored during the search is 62 and the maximum memory usage is 68. The number of nodes on the solution path is 24.

3.3 Question

For question "Using a linked list to keep track of which states have been visited would be a poor choice. Why?"

My answer to this question is that using linked list is inefficient to do searching from the list while HashMap is much more efficient to search wanted node due to its feature of key and value.

4 Memorizing depth-first search

The main idea of `Memorizing depth-first search` is to keep track of all nodes that have been explored and make sure every node that has been already explored will not be explored for a second time and to apply `DFS` on the graph to return solution path.

4.1 Imeplementation

Although it can be implement by `stack`, here I use `recursive` method to implement Memorizing depth-first search.

Here's my code for `Memorizing depth-frost search`, it includes two function, one is main function named `depthFirstMemoizingSearch()`, which is to input start state and initial variables. The other one is recursive function named `dfsrm()`, which is to recursively call itself if it cannot reach the base case. My code is shown below:

Listing 10: Memorizing depth-first search

```
public List<UUSearchNode> depthFirstMemoizingSearch(int maxDepth) {
  resetStats();
  HashMap<UUSearchNode,Integer> visited = new
      HashMap<UUSearchNode,Integer>();
  return dfsrm(startNode,visited,0,maxDepth);
  }

private List<UUSearchNode> dfsrm(UUSearchNode currentNode,
 HashMap<UUSearchNode, Integer> visited, int depth, int maxDepth) {
// keep track of stats; these calls charge for the current node
List<UUSearchNode> path = new ArrayList<UUSearchNode>();
if(depth>maxDepth){
  return null;
  }else{
     if(depth==0){
       visited.put(startNode, 0);
       incrementNodeCount();
       }

       //base case
       if(currentNode.goalTest()){
         path.add(currentNode);
         incrementNodeCount();
         return path;
         }
      List<UUSearchNode> successor = currentNode.getSuccessors();

      //maxsuccessor is a global variable to record the maxmemory to store
          successors
      if(successor.size()>maxsuccessor){
        maxsuccessor = successor.size();
        }

      //recursive case
       if(!successor.isEmpty()){
         for(UUSearchNode child: successor){
```

```
            if(!visited.containsKey(child)){
              visited.put(child,depth+1);
              //after visited.put, it means a new node has been explored
              incrementNodeCount();
              //the maxmemory is the new max memory of visited node
              //plus the max memory to store successors
              updateMemory(visited.size()+maxsuccessor);
              List<UUSearchNode> next_path =
                            dfsrm(child,visited,depth+1,maxDepth);
              if(next_path!=null){
                path = next_path;
                path.add(0,currentNode);
                break;
                }
               }
              }
               return (path.isEmpty()) ? null:path;
             }
            }
           return null;
          }
```

In my implementation of Memorizing depth-first search, I firstly write a function called `depthFirstMemoizingSearch()`, it is a main function to put the `startNode` and other initial values into recursive method `dfsrm()`. I think it is a "driver" to begin the recursive function. Secondly, I write the recursive function `dfsrm()`. It will at first check if `depth > maxDepth` which means the depth should not be too deep and should now greater than the maximum depth I set. After checking the depth, it will enter to the main body of this method.

To begin, it will put the `startNode` into a HashMap named `visited`, which means the start node has been explored. Then I write the base case: if there is a node satisfy my method `goalTest()`, which means it reach the goal state and finds the right solution path, then it will return the solution path. After the base case, I write the recursive case. If the current node has successors (after applying `getSuccessor()`), it will check each successor recursively. More specifically, it will begin from the first successor among all successors of current node, if it has not been visited yet, then add it to `visited` HashMap. And then start from this node, to call `dfsrm()` recursively with `depth+1`. When the current node reach to goal state and satisfy base case, it will go back to the former layer and add the current node to the solution path recursively. Finally, the method will return the right solution path of the problem.

Also, in order to track the number of nodes that have been explored during the search as well as the maximum memory used to find the solution path, I call `incrementNodeCount()` method every time there is call for `visited.put()`. And I record the maximum memory to store successors and plus the maximum memory to store explored nodes to act as the maximum memory used to find the solution path.

4.2 Test Result

To test `Memorizing depth-first search` , I used two cases: `(3,3,1)` and `(8,5,1)`. The result of `(3,3,1)` has following results:

Listing 11: MDFS on (3,3,1)

```
dfs memoizing 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:  14
```

```
Maximum memory usage during last search 16
--------MDFS
```

The result of (8,5,1) has following results:

Listing 12: MDFS on (8,5,1)

```
dfs memoizing path length: 34[[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 45
--------DFMS
```

4.3 Question

For the question about "Does memorizing DFS save significant memory with respect to breadth-first search? Why or why not?", my answer is that Memorizing DFS do save memory with respect to breadth-first search. But as for significant or not, I think it depends.

Consider there are two extreme situation. If the goal state is just at left corner and each nodes except leaf nodes have thousands of successors, then memorizing DFS will save significant memory than BFS because BFS need to traverse every successors in each layer and then get to the goal node while Memorizing DFS just get successors of each node on the left for each layer and go deep to the goal state. However, there is an opposite situation to it. If the goal state is just at right corner and also each nodes except leaf nodes have thousands of successors, then Memorizing DFS will do the same thing like BFS, so it cannot save much memory than BFS.

Moreover, there is also a possible situation that each node has only one successor, then BFS will do the same work with Memorizing DFS, therefore, Memorizing DFS also cannot save much memory than BFS. And, these extreme situations I shown here are just part of them, therefore, just saying Memorizing DFS save significant memory with respect to breadth-first search is not very precise.

While except those extreme situations, since memorizing DFS do not need to traverse nodes at each layer, Memorizing DFS can use less memory than BFS in average situations.

Therefore, I think Memorizing DFS can save memory than BFS can except some extreme situations.

5 Path-checking depth-first search

Path-check depth-first search is another type of DFS. The different between Path-checking DFS and memorizing DFS is that memorizing DFS will record every node that has been visited whatever it is on the solution path or not while Path-checking DFS will only record nodes that is on the solution path and also those nodes that have been visited before will not be visited for a second time even though they are not recorded.

5.1 Implementation

The implementation of path-checking depth-first search is very similar to the implementation of memorizing depth-first search, they all use the idea of recursive to do depth first search. However, there are still some difference between them. Rather than using `HashMap` to keep track of visited node, here I use a `HashSet` to store visited node. Also, each time if the current node has no successor and has not reach the goal state, then it will come back to the former layer and remover current node from visited set. Therefore, it can

implement that the HashSet currentPath only store nodes that are on the solution path. Also, it can ensure that a visited node can only be visited for once.

Here is my code of Path-checking depth-first search:

Listing 13: Path-checking depth-first search

```
public List<UUSearchNode> depthFirstPathCheckingSearch(int maxDepth) {
  resetStats();
  HashSet<UUSearchNode> currentPath = new HashSet<UUSearchNode>();
  return dfsrpc(startNode, currentPath, 0, maxDepth);
  }

  // recursive path-checking dfs. Private, because it has the extra
  // parameters needed for recursion.
  private List<UUSearchNode> dfsrpc(UUSearchNode currentNode,
     HashSet<UUSearchNode> currentPath,int depth, int maxDepth) {
  List<UUSearchNode> solutionpath = new ArrayList<UUSearchNode>();

  if(depth>maxDepth){
   return null;
   }else{
    if(depth == 0){
      //add initial start node to visited set
      currentPath.add(startNode);
      incrementNodeCount();
     }

    //Base Case
    if(currentNode.goalTest()){
      solutionpath.add(currentNode);
      //update max memory with max memory used to store successors
      //plus max memory used to store visited node
      updateMemory(maxsuccessor+maxvisited);
      return solutionpath;
      }

    List<UUSearchNode> successors = currentNode.getSuccessors();
    if(successors.size()>maxsuccessor){
        //record max memory of storing successors
        maxsuccessor = successors.size();
        }

      //recursive case
    if(!successors.isEmpty()){
          for(UUSearchNode child: successors){
            if(!currentPath.contains(child)){
              currentPath.add(child);
              incrementNodeCount();
              if(currentPath.size()>maxvisited){
                maxvisited = currentPath.size();
                }
                List<UUSearchNode> next_path =
                    dfsrpc(child,currentPath,depth+1,maxDepth);
```

```
                if(next_path!=null){
                  solutionpath = next_path;
                  solutionpath.add(0,currentNode);
                  break;
                  }else{
                    currentPath.remove(child);
                    }
                  }
                 }
                    return (solutionpath.isEmpty()?null:solutionpath);
               }
              return null;
             }
         }
```

In my implmentation, the `Path-checking DFS` is implemented by recursive method. Initially, it will call function `depthFirstPathCheckingSearch` to input initial variables into recursive method `dfsrpc()`. Then, it will come into the recursive method. At the beginning, the program will check if the depth is greater than maximum depth or not. If depth is not greater than maximum depth, then it will continue the program and put the start node into `HashSet currentPath`, which means the start node has already been explored.

Then, it comes to the base case, if the program reach the final goal state, it will return the solution path and update the maximum memory used during this search. If the program have not reached the goal state, it will come into the recursive case and to traverse the successors of current node recursively. Just as I said before, if a `currentNode` has no successors and has not reach the goal state, it will be removed from `currentPath` which help ensure that `currentPath` only keep track of those nodes on the solution path. That is the difference from memorizing DFS. Then, the program will finally return the solution path and print it out through `main` program.

In order to track nodes explored during the search, I call function `incrementNodeCount()` just after `currentPath.add()` every time. For tracking maximum memory used, I use a global variable `maxsuccessor` to record the maximum memory usage for getting successors and also I use a global variable `maxvisited` to record the maximum memory usage for storing visited node. And in the base case, I call method `updateMemory` to update the maximum memory usage during the search by adding the value of `maxsuccessor` and `maxvisited`.

5.2 Test Result

The result of `(3,3,1)` has following results:

Listing 14: Path-checking DFS on (3,3,1)

```
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 15
```

The result of `(8,5,1)` has following results:

Listing 15: Path-checking DFS on (8,5,1)

```
dfs path checking path length:34[[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:  40
Maximum memory usage during last search 39
```

5.3 Question

5.3.1 Question one

For the question "Does path-checking depth-first search save significant memory with respect to breadth-first search?"

My answer is that path-checking depth-first search do save memory with respect to breadth-first search and in somehow save more space with comparison with memorizing depth first search. But for significant or not, it depends on the position of the goal state on the state graph and how deep the graph is and how breath of each layer of the graph. Like the explanation I have on the Question section of Memorizing DFS, there exist some situations that path-checking depth-first search have just slightly save memory than BFS or have not save memory with respect to BFS or even use more memory compare to BFS. Then, just saying it saves significant memory with respect to breadth-first is not rigorous.

However, consider that path-checking depth-first search do save memory with respect to BFS in a lot of situations and it reduces some memory to store those visited nodes but not on the solution path, path-checking depth-first search can also save some memory compare to memorizing DFS.

Therefore, I think path-checking depth-first search can save memory than BFS and Memorizing DFS but I cannot agree with the saying it saves significant memory.

5.3.2 Question two

Draw an example of a graph where `path-checking DFS` takes much more run-time than breadth-first search; include in your report and discuss.

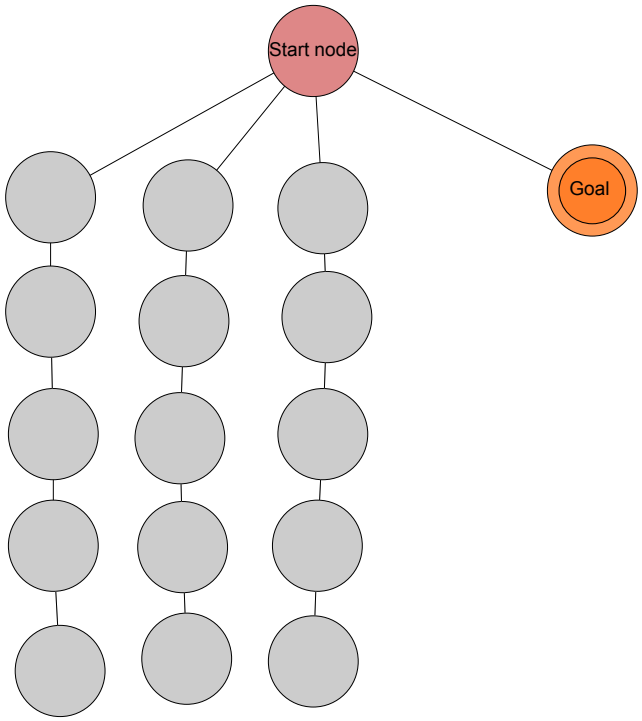

Figure 2: Example

From my example of graph, running a BFS on it to reach goal node only need to explore 5 nodes (including start node and goal node). However, since the depth is very deep, running a path-checking DFS will go down to the deepest node for three time and then reach to the goal node. So the total nodes that running a path-checking DFS on the graph need to explore 17 nodes (including start node and goal node). Therefore, situation like this will make path-checking DFS takes much more run-time than BFS.

6 Iterative deepening search

Iterative deepening search is to iteratively find the shortest solution path from depth 0 to the depth where goal node is on. Although BFS can find the shortest path, it needs too much memory during the search. As I discuss before, the two version of DFS can save memory than BFS. Then, applying DFS on each iteration on traverse each layer of the graph can help to save memory to find the shortest solution path.

6.1 Implementation

The implementation of IDSearch is very simple. It just need to add a `for` loop to help the program do DFS on each depth from depth = 0 to depth = the depth of where goal state is on. By calling `dfsrpc` iteratively, it applies recursive version of DFS on each iteration by adding depth+1 each time. If the solution path is not null, which means the solution path is found, it can jump out of the iteration and return the solution path. At beginning of each iteration, the `currentPath` should be empty, which means that each iteration begin a totally new DFS and nodes that have been visited in former iterations should be set as non-visited.

Here is my code for `IDSearch`. Since method `dfsrpc` is the same code with path-checking DFS, I do not show the code here.

Listing 16: Iterative deepening search

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

    //iteratively find solution path by adding depth by DFS
    for(int i = 0; i < maxDepth+1; i++){
      currentPath.clear();
      solutionPath = dfsrpc(startNode, currentPath, 0, i);
      //If the solution path is found, then jump out of the iteration
        if(solutionPath != null){
          break;
          }
        }
          return solutionPath;
       }
```

6.2 Test Result

The result of `(3,3,1)` has following results:

Listing 17: Iterative deepening search on (3,3,1)

```
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:  161
Maximum memory usage during last search 15
```

The result of `(8,5,1)` has following results:

Listing 18: Iterative deepening search on (8,5,1)

```
Iterative deepening (path checking) path length:24[[8, 5, 1], [8, 3, 0], [8,
    4, 1], [8, 2, 0], [8, 3, 1], [6, 3, 0], [6, 4, 1], [5, 3, 0], [5, 4, 1],
    [5, 2, 0], [5, 3, 1], [4, 2, 0], [4, 3, 1], [4, 1, 0], [4, 2, 1], [3, 1,
    0], [3, 2, 1], [3, 0, 0], [3, 1, 1], [2, 0, 0], [2, 1, 1], [1, 0, 0], [1,
    1, 1], [0, 0, 0]]
Nodes explored during last search:  19425294
Maximum memory usage during last search 39
```

Since IDSearch finds the shortest solution path as the same with BFS do, the result of path length should equal to the result of BFS. From my test result of IDSearch, I have the same path length with BFS on both two test cases. Therefore, my implementation of IDSearch is correct.

6.3 Question

For the question "On a graph, would it make sense to use Path-checking DFS, or would you prefer memorizing dfs in your iterative deepening search?"

My answer is that there are note so much different to use Path-checking DFS or Memorizing DFS. If I have to choose one from them, I prefer Path-checking DFS since it can save more space than using Memorizing DFS (record only nodes on the solution path).

However, Although IDSearch takes advantage of DFS to save space and has the same function of BFS to find the shortest solution path, it takes too much running time on exploring nodes. Take a look at my test case (8, 5, 1), the nodes explored during IDSearch reach as much as 19425294. Although it uses the fewest maximum memory among BFS, Memorizing DFS and Path-checking DFS, comparing to the running time on exploring nodes, it wastes a lot of time. Also, it is very clear when I run test case `(8, 5, 1)` I need to wait for 5 or 10 seconds to have its result shown.

Therefore, consider both time and space complexity together, I prefer to use BFS instead.

7 Lossy missionaries and cannibals

7.1 Question

7.1.1 Question one

For the question "What if, in the service of their faith, some missionaries were willing to be made into lunch? Let us design a problem where no more than E missionaries could be eaten, where E is some constant. What would the state for this problem be? What changes would you have to make to your code to implement a solution?"

To solve this problem, I think I will add more safe state situations into my `isSafe()` function.

After checking the 6 prerequisites for a safe node ((1) the number of people on the boat must greater than or equal to 1; (2) the number of people one the boat must smaller than or equal to the size of boat; (3) the number of missionaries on the starting side must greater or equal to 0; (4) the number of cannibals on the starting side must greater or equal to 0; (5) the number of missionaries on the goal side must greater or equal to 0; (6) the number of cannibals on the goal side must greater or equal to 0.). The `isSafe()` function will need to add the following checking situations:

(1) If the boat is on the starting bank side and there is `R` missionaries and `U` cannibals on the goal bank side, `M` missionaries and `C` cannibals on the starting bank side, and the boat is going to carry `T` people to the goal bank side then:

Listing 19: pseudocode of this situation

```
if(T + U > R && R<= E && R>= U){
  if(C - T<= M){
    TotalMissinaries = TotalMissinaries - R;
    E = E -R;
    return true;
    }else if( C - T > M ){
       if ( M < E - R ){
         TotalMissinaries = TotalMissinaries - R;
         E = E - R - M;
         return true;
         }else{
            return false;
            }
          }
        }
```

At the beginning, it should check if the number of cannibals sending to the goal side + the number of cannibals already on the goal side is greater than the number of missionaries on the goal side and if the number of missionaries on the goal side is smaller than or equal to E (the number of missionaires who want to be killed) as well as if the number of missionaries on the goal side is greater or equal to the number of canals on the goal side. If all these conditions are satisfied, then continue.

If the number of cannibals on the starting side is smaller than or equal to the number of missionaries on the starting side, then it is a safe state.The total number of missionaries should minus those missionaries who have been eaten. And the number of missionaries who want to be killed should minus those who have been eaten.

If the number of cannibals on the starting side is greater than the number of missionaries on the starting side, then it need to check whether the number of missionaries on the starting side is smaller than the rest number of missionaries who want to be killed. If it is smaller, then it is a safe state and update the number of total missionaries and the number of E. If it is not, then return false, it is not a safe state.

If the boat is on the goal side and there is only `R` missionaries where `R <= E` and 0 cannibals on the starting bank side, then it is ok to carry `T` cannibals back to the starting bank side where `T > R`, and also need to ensure that after sending those `T` cannibals, the number of missionaries on the goal side should still greater than or equal to the number of cannibals on the goal side. Then, it is a safe state.

(2) If the boat is on the goal bank side and there are still `R` missionaries and `U` cannibals on the goal bank side, `M` missionaries and `C` cannibals on the starting bank side, and the boat is going to carry `T` people to the starting bank side then:

Listing 20: pseudocode of this situation

```
if(T + C > M && M <= E && M >= C){
  if(U - T <= R){
    TotalMissinaries = TotalMissinaries - M;
    E = E - M;
    return true;
    }else if( U - T > R ){
       if ( R < E - M ){
         TotalMissinaries = TotalMissinaries - M;
         E = E - R - M;
         return true;
         }else{
            return false;
            }
```

```
            }
        }
```

The idea of this situation is very alike to the situation that the boat is on the starting side. Since I have explained it in situation (1), I will not explain it again here.

7.1.2 Question two

For the question "Give an upper bound on the number of possible states for this problem."

Assuming there are totally `tM` missionaries and `tC` cannibals in this problem. Then the upper bound `upperBound` on the number of possible states should be:

$upperBound = (tM + 1) * (tC + 1) * 2$

Although there will be some missionaries to be killed during the program, the number of missionaries will not exceed the totoal number of missionaries. By calculating `(tM+1)*(tC+1)*2`, it can ensure all possible states have already be calculated. Therefore, the upperBound is equal to `(tM+1)*(tC+1)*2`.