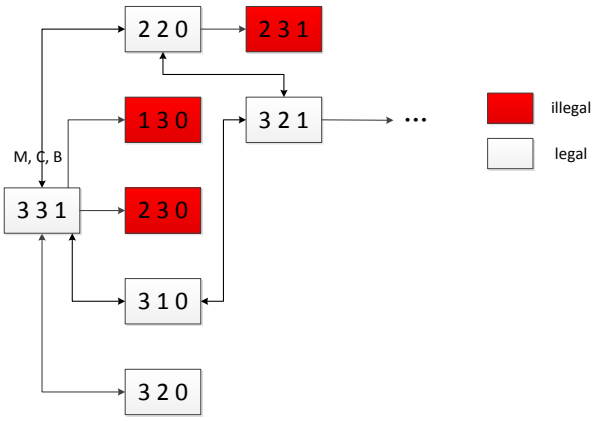
# Missionaries and Cannibals Solution

Xinqi Li

January 14, 2014

## 1 Introduction

Without considering the validity of each state, the upper bound of state for [3 3 1] problem is 4*4*2=32. Because there can be 0 to 3 missionaries or cannibals in state and there can have or have no boat on the start side. A part of the state graph like this:




There are 5 possible state after 331: 320, 310, 230, 130 and 220. since m should no less than c, 230 and 130 are illegal states which are red. When it comes to the successors of legal states, one common feature for them is that one of their successors is their father node. We can see that there is only one NEW legal state when it comes to depth 2(331 is on depth 0).

## 2 Implementation of the model

### 2.1 Implementation and How it Works

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

```
public ArrayList<UUSearchNode> getSuccessors() {
  // add actions (denoted by how many missionaries and cannibals to put
  // in the boat) to current state. ******legal states
  ArrayList<UUSearchNode> successors=new ArrayList<UUSearchProblem.UUSearchNode>();
```

```
    int m_current=state[0];
    int c_current=state[1];
    int b_current=state[2];
    int m_move;
    int c_move;
    int c_suc;
    int m_suc;
    int d_suc=depth+1;//successors depth is one more than the current state's.

    if(b_current==1)//minus
    {
      for(m_move=0;m_move<=BOAT_SIZE;m_move++)
        for(c_move=0;c_move<=BOAT_SIZE;c_move++)
        {
      if((m_move+c_move)!=0&&(m_move+c_move)<=BOAT_SIZE)
      {
        m_suc=m_current-m_move;
        c_suc=c_current-c_move;
        if(isSafeState(m_suc,c_suc))
        {
          CannibalNode suc= new CannibalNode(m_suc, c_suc, 0, d_suc);
          successors.add(suc);
        }
      }
        }
    }
    else if(b_current==0)//add
    {
      for(m_move=0;m_move<=BOAT_SIZE;m_move++)//change the cannibal's amount at first
        for(c_move=0;c_move<=BOAT_SIZE;c_move++)
        {
      if((m_move+c_move)!=0&&(m_move+c_move)<=BOAT_SIZE)
      {
        m_suc=m_current+m_move;
        c_suc=c_current+c_move;
        if(isSafeState(m_suc,c_suc))
        {
          CannibalNode suc= new CannibalNode(m_suc, c_suc, 1, d_suc);
          successors.add(suc);
        }
      }
        }
    }
    return successors;

  }
```

The basic idea of `getSuccessors` is that by going through all the possible number of change to cannibals and missionaries amount, we can have the number of c and m a successor state might have.For example, if the current state is [3,2,0](in [m,c,b]order), the possible moves (in [m-move,c-move] order) are [0,1],[0,2],[1,1],[1,2]and [2,2]. In this case, we will have 5 successor states(let's call it suc-state). Then I pass

the suc-state into `isSafeState` function to check for its legality.

I used a method `isSafeState` that returns `true` if

- Intuitively, the number of c and b is not negative and no more than the maximum.
- For the start side, m¿c or the exception is all the missionaries are on the other side
- For the other side, missionaries should also be more than cannibals or the exception is all the missionaries are on the start side

The code for `isSafeState`is here:

```
private boolean isSafeState(int m, int c) {
  if(m>=0&&c>=0&&m<=totalMissionaries&&c<=totalCannibals
   &&(m>=c||m==0)
   &&(totalMissionaries-m>=totalCannibals-c)||m==3)
    return true;
  else
    return false;
}
```

### 2.2 Testing the code:getSuccessors

I write a test code to output the successors of one state in `CannibalProblem.java`.

```
public static void main(String args[]) {
  CannibalProblem mcProblem = new CannibalProblem(3, 3, 1, 0, 0, 0);
  CannibalNode currentnode=mcProblem.new CannibalNode(2,2, 1, 6);
  ArrayList<UUSearchNode> successors= new ArrayList<UUSearchNode>();
  successors=currentnode.getSuccessors();
  System.out.println("successors of "+currentnode+"are"+successors);
}
```

Here I test the state 221 in 331 problem. 6 is the depth of state 221. The result is:
successors of [221]are[[110], [020]]
2 m and 2 c are on the start side which means the other side has 1 m and 1 c. Now to move people from start side we have 5 possibilities: 1m1c, 1m, 1c, 2m and 2c. Considering the validity, moving 1m1c or 2m is safe, hence the safe successors are 110 and 020 which verifies the result.

## 3 Breadth-first search

### 3.1 Implementation and How it Works

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

```
public List<UUSearchNode> breadthFirstSearch(){
  resetStats();
  int i;
  HashMap<UUSearchNode,UUSearchNode> explored=new HashMap<UUSearchNode,UUSearchNode>()
  List<UUSearchNode> path=new ArrayList<UUSearchNode>();//the path to goal
  Queue<UUSearchNode> queue = new LinkedList<UUSearchProblem.UUSearchNode>();

  queue.add(this.startNode);//initializing, add start node in queue
```

```
    explored.put(startNode,null);
    incrementNodeCount();//update the node it explored
    updateMemory(queue.size()+explored.size());

    while(!queue.isEmpty()) {
      //dequeue the head of queue to be the current node
      UUSearchNode current_node = (UUSearchNode)queue.remove();

      if(current_node.goalTest()==true){
          path=backchain(current_node, explored);
      }
      else{
          //add unexplored successors to fringe
          ArrayList<UUSearchNode> successors= new ArrayList<UUSearchNode>();
          successors=current_node.getSuccessors();
          for(i=0;i<successors.size();i++) {
          //update the node it explored before check for explored ones
            incrementNodeCount();
            if(explored.containsKey(successors.get(i))==false){
                explored.put(successors.get(i),current_node);
                queue.add(successors.get(i));
                updateMemory(queue.size()+explored.size());

            }//if
          }//for
      }//else
    }//while
    return path;
  }
```

The basic idea of `breadthFirstSearch` is using a queue structure to determine the visit order of each state. My code follows the following steps:

- Initialization: a hashMap to store the explored states, a list to store the valid states in path and a queue to determine the visit order of successors. At the beginning, add start state to queue.
- Dequeue the state to be the current state. If this state is the goal than call `backchain` to get the path
- Otherwise enqueue current state's unexplored successors one by one, keep track of them in hashMap.
- Repeating the former 2 steps until the queue is empty.

Heres my code for `backchain`:

```
private List<UUSearchNode> backchain(UUSearchNode node,
        HashMap<UUSearchNode, UUSearchNode> visited)
    {
    List<UUSearchNode> path=new ArrayList<UUSearchNode>();

    while(node!=null){
      path.add(node);
      node=visited.get(node);
    }
```

```
    Collections.reverse(path);
    return path;
  }
```

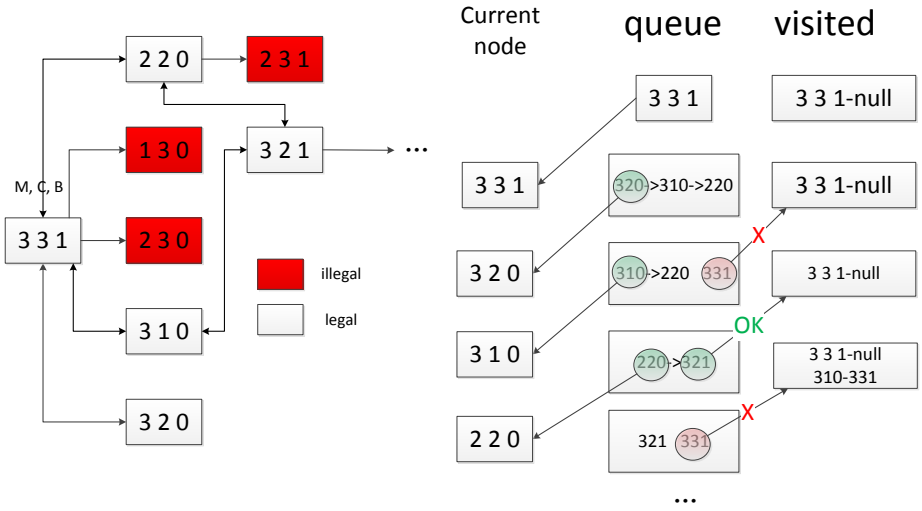
The goal state and the visited hashMap will be passed in the function. In the while loop I add state in list backward, then I use a reverse function in java to make it forward. It's very convenient.

### 3.2 Testing and Result of the code:breadthFirstSearch

The result is:

```
bfs path length:  12 [[331], [310], [321], [300], [311], [110],
[221], [020], [031], [010], [021], [000]]
Nodes explored during last search:  31
Maximum memory usage during last search 17
```

To verify the result we can see the process in following graph:




At the beginning, 331 is in queue. Then we dequeue 331 to be current node. Since it's not the goal, we put [331, null] into visited hashMap, examine its successors and enqueue them one by one. Now we have 320, 310, 220 in queue. They are also in the visited hashMap. Then dequeue 320, examine its successor 331. Since 331 is in visited hashMap and 320 has no more successors, we dequeue 310. Again we find 331 has been visited so we enqueue 321. Now the queue has220, 321. The process will keep going until it find the goal or queue is empty. And we can see the path shows state order is 331, 310, 321 just as I implied formerly.

### 3.3 Q&A

Q: Why using linked list to keep track of visited states id a bad idea?
A: To find a state in linked list we should, at worst case, go through all the states in list. In BFS, when we go deeper, the number of visited states growth exponentially since BFS will go through all states in one depth. In practice, I use HashMap to store the visited state and its source state.

# 4 Memoizing depth-first search

The model is implemented in `UUSearchProblem.java`. Here's my code for `depthFirstMemoizingSearch` This code only provides an entrance to the recursive code

```
public List<UUSearchNode> depthFirstMemoizingSearch(int maxDepth) {
  resetStats();
  UUSearchNode current_node=this.startNode;
      //explored:tracking the explored nodes
  HashMap<UUSearchNode, Integer> explored=new HashMap<UUSearchNode,Integer>();
  explored.put(current_node, 0);
  updateMemory(explored.size());
  incrementNodeCount();
      //the path to goal
  List<UUSearchNode> path=dfsrm(current_node, explored, 0, maxDepth);
  return path;
}
```

The code for the recursive method `dfsrm` is:

```
private List<UUSearchNode> dfsrm(UUSearchNode currentNode, HashMap<UUSearchNode, Integer>
    int depth, int maxDepth) {
  List<UUSearchNode> current_path=new ArrayList<UUSearchNode>();
  // keep track of stats; these calls charge for the current node
  updateMemory(visited.size());
  int i;//count for the successors

  //base case
  if(currentNode.goalTest()==true)
  {
    current_path.add(currentNode);
    return current_path;
  }
//    recursive case
  else {
    ArrayList<UUSearchNode> successors= new ArrayList<UUSearchNode>();
    successors=currentNode.getSuccessors();

      for(i=0;i<successors.size();i++) {
        incrementNodeCount();
        if( successors.get(i).getDepth()<=maxDepth //depth must smaller than maxDepth
            &&( visited.containsKey(successors.get(i))==false//not be visited
            ||successors.get(i).getDepth()<visited.get(successors.get(i))))
//or if the depth is smaller than the visited one
        { //go ahead to search the successor(i)

          visited.put(successors.get(i),successors.get(i).getDepth());
          List<UUSearchNode> suc_path=new ArrayList<UUSearchNode>();
          suc_path=dfsrm(successors.get(i), visited,
                  successors.get(i).getDepth(), maxDepth);
          if(!suc_path.isEmpty()){
            current_path.clear();
            current_path.add(currentNode);
```

```
                current_path.addAll(suc_path);//the path to goal
                return current_path;
              }
            }
          }
          return current_path;
      }
  }
```

The function pass the current node, current path, depth for current node and the maxDepth to limit the dfs.
The base case is when the current node is the goal. In this case I add current node to path and return path.
The recursive case is when current node is not the goal. I examine one of current node's successors and find the successive path that start from the successor. If the successive path is empty I will try another successor.

### 4.1 Some Details

Memoizing method like bfs, keeps track of the visited states. After getting a successor, I first exam whether it is explored or whether is too deep that exceeds max depth limit. But the exception is when visited depth is larger than the current depth. Keep exploring this can help us find a shorter way or maybe can avoid not finding a way shorter than the depth limit. This is every useful when we apply Memoizing dfs in IDS which can find out the shortest way.
Every time before I add currentNode to path I will clear the path and then add the current node and the successive path from it. This guarantees every time I return the right path.

### 4.2 Q&A

Q:Does memoizing dfs save significant memory with respect to BFS?
A:Not really. Basically, the result of 851 as start state shows that bfs costs 53 units memory(1 unit is for one UUSearch node), however dfs costs 37 units. The reason is that BFS explores almost all valid states however, DFS at best only searches the states on its path and the one brother state of it. However, in the worst case dfs will cost as much memory as bfs does because it might choose the wrong successor to go on. In this case, dfs might search all states to find out goal or never find goal if the graph is infinite.

## 5 Path-checking depth-first search

The model is implemented in `UUSearchProblem.java`. Here's my code for `depthFirstPathCheckingSearch`

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

}
```

The code for the recursive method `dfsrpc` is:

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

  // you write this method
```

```
    currentPath.add(currentNode);
    // keep track of stats; these calls charge for the current node
    updateMemory(currentPath.size());

    int i;//count for the successors
    List<UUSearchNode> path=new ArrayList<UUSearchProblem.UUSearchNode>();
    //base case
    if(currentNode.goalTest()==true)
    {
      path.add(currentNode);
      return path;
    }
//    recursive case
    else {
      ArrayList<UUSearchNode> successors= new ArrayList<UUSearchNode>();
      successors=currentNode.getSuccessors();

        for(i=0;i<successors.size();i++) {
          incrementNodeCount();
          if( successors.get(i).getDepth()<=maxDepth //depth must smaller than maxDepth
              && currentPath.contains(successors.get(i))==false)//not be visited
          { //go ahead to search the successor(i)

            List<UUSearchNode> suc_path=new ArrayList<UUSearchNode>();
            suc_path=dfsrpc(successors.get(i),
                currentPath, successors.get(i).getDepth(), maxDepth);
            if(!suc_path.isEmpty()){
              //currentPath.clear();
              path.add(currentNode);
              path.addAll(suc_path);//the path to goal
              return path;
            }
            else
              currentPath.remove(successors.get(i));
                //becareful to the change of the currentpath,
                //I add current node in every begin point of a fuction
                //nomatter whether it has sub_path
          }
        }
        return path;
    }
  }
```

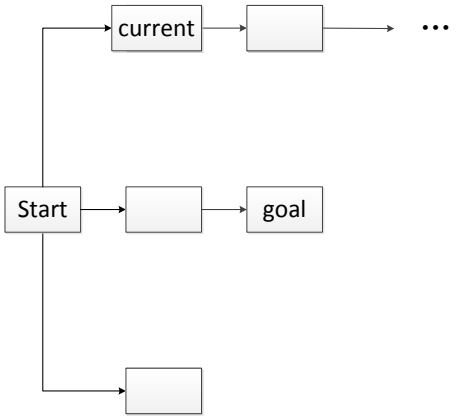
The function pass the current node, current path which is a hashset, depth for current node and the maxDepth to limit the dfs.
The base case is when the current node is the goal. In this case I add current node to path and return path. The recursive case is when current node is not the goal. I examine one of current node's successors. First see if it is not on the path. Then find the successive path that start from the successor. If the successive path is empty I will try another successor.

### 5.1 Some Details

Every time I call the dfsrpc function I will add current node to path for the future path checking. So I should remove the successor from path if I cant find a successive path from the successor.
The current path is a hashset not linked list is because hashset find exist node easily however we need to at worst go through the list to find a exist node.

### 5.2 Q&A

Q:Does path checking dfs save significant memory with respect to BFS?
A:Not really. DFS doesn't guarantee to find a path since it choose the wrong successor to go on. As the graph showed below, in this case, dfs might find goal since it choose to go to a successor that have infinite successive states.




## 6 Iterative deepening search

The model is implemented in`UUSearchProblem.java`. Here's my code for `IDSearch`

```
public List<UUSearchNode> IDSearch(int maxDepth) {
  resetStats();
  // you write this method
  int md=0;

  //HashSet<UUSearchNode> explored=new HashSet<UUSearchNode>();//explored:tracking the e
  HashMap<UUSearchNode, Integer> explored=new HashMap<UUSearchNode, Integer>();
  explored.put(startNode, null);
  List<UUSearchNode> path=new ArrayList<UUSearchProblem.UUSearchNode>();
  updateMemory(explored.size());
  incrementNodeCount();
```

```
    for(md=0;path.isEmpty();md++){
      if(md>maxDepth)
        return path;

      explored.clear();
      path=dfsrm(startNode, explored, 0, md);//the path to goal
    }
    return path;
  }
```

The IDS calls the path checking dfs. It adds up the max depth every time it fail to find a path until it can find a path in one depth. One point should be aware is that every time it calls the method to find path we should clear the expored list since in java we pass the name not the list itself in function.

### 6.1 Q&A

Q:On a graph, would it make sense to use path-checking dfs, or would you prefer memoizing dfs in your iterative deepening search? Consider both time and memory aspects.
A:Basically, I would use BFS. However, I prefer using memoizing than path checking IDS. The outputs for using different DFS methods in IDS are shown below:

```
Iterative deepening (path checking) path length:24 [[851], [740],
[751], [640], [651], [630], [641], [530], [541], [520], [531], [420],
[431], [410], [421], [310], [321], [300], [311], [200], [211], [100], [111], [000]]
Nodes explored during last search:  31281497
Maximum memory usage during last search 24

Iterative deepening (memoizing) path length:24 [[851], [740],
[751], [640], [651], [630],[641], [530], [541], [520], [531], [420],
[431], [410], [421], [310], [321], [300],[311], [200], [211], [100], [111], [000]]
Nodes explored during last search:  4622
Maximum memory usage during last search 51
```

Path checking saves memory since it only need to store states on path. However, when nodes become too much, the path checking method will explore more nodes than memoizing dfs to find a way cannot get to goal. Than this processor will repeat since the max depth accumulates. Hence the path checking method will explored too much nodes which is time consuming.
On the other hand, memoizing method needn't explore too much nodes that it save time compared to pcdfs. However the iterative still cost more time than BFS. To this cannibal problems, BFS is still a better method. However if graph contains more nodes and goal is far, BFS will cost more time to go through almost all nodes, however dfs may be fast to get to goal when it begin to search on depth that is the goal's depth.

## 7 Lossy missionaries and cannibals

Basically if no missionary be eaten, the total states(upper bound) number is:
(totalm+1)*(totalc+1)*(totalb+1)
Take [3,3,1] as an example, it should be 32 kinds of states.
In this scenario, E(E is positive number) missionaries can be eaten. That means in some points after one m is eaten, the totalm will decrease for 1 and at least currenttotalm will be totalm-E. So the total states(upper bound) is:
(totalm+1+totalm-1+1+totalm-2+1+...+totalm-E+1)*(totalc+1)*(totalb+1)

The `isSafe` function should be changed. E should be pass into this function. Outside the current if-else we should examine whether totalMissionaries-currentMissionaries is no greater than E. if no we don't change the former condition. Otherwise we change condition by adding more to allow m be eaten. Then if the some m be eaten in this state, the totalM should be deducted.
In every node by checking the currentM and the totalM we can know whether more M can be eaten.