

# Mazeworld Solution

Junjie Guan <gjj@cs.dartmouth.edu>

January 21, 2014

## 1 Introduction

Solving maze is one of the most classic and popular problems in Artificial Intelligence. This report majorly cover three parts, 1) introducing the A\* algorithm as a searching method; 2) Multirobot problem, where we need to take collision into consideration; 3) Blind robot problem, where the robot need to find out its current coordinate in the maze; 4) finally, some further discussion.

## 2 A-star search

### 2.1 Basic Idea

A\* is a kind of informed search, which is different from traditional uninformed search (such as bfs). One huge difference is, instead of searching while trying to maintain as least cost as possible in bfs, A\* also consider another value called heuristic. Figure 1 is a demonstration of A\* algorithm. On one hand, the solid line represent the path that we already go through, which is an evaluation of the past. On the other hand, the dash line, it represents the estimated/expected cost from current state to the goal, which is an evaluation of the future.

At every time A\* pick the new state with lowest priority from a priority queue. Usually, it considers the past and the future simultaneously. Let's say the priority value is  $f$ , value of cost is  $g$ , and heuristic is  $h$ , then:

$$f = g + h$$

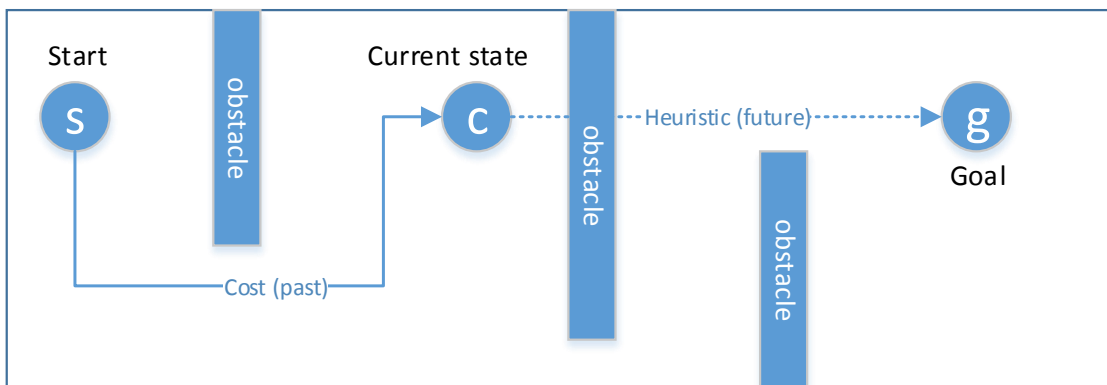


Figure 1: A demonstration of A\* algorithm

## 2.2 Code implementation

**Line 3-7:** There are three major data structure to help me code astar.

- Priority queue: I use a priority queue to store the *frontiers*, sorted by priority. Every search I will pop a node from the head of the queue.
- Hash Map  $\times 2$ : One hashmap maps from node to node, for creating a backchain at the end. Another hashmap maps from node to priority, for the situation when we re-visit a node, it only worth expanding only if it has higher priority/cost than before.

```
1 public List<SearchNode> astarSearch() {
2     resetStats();
3     // implementing priority queue for the frontiers
4     PriorityQueue<SearchNode> frontiers = new PriorityQueue<>();
5     // implementing hashmap for the chain and the visited nodes
6     HashMap<SearchNode, SearchNode> reachedFrom = new HashMap<>();
7     HashMap<SearchNode, Double> visited = new HashMap<>();
8
9     // initiate the visited with startnode
10    reachedFrom.put(startNode, null);
11    visited.put(startNode, startNode.priority());
12    // initiate the frontier
13    frontiers.add(startNode);
14    while (!frontiers.isEmpty()) {
15        // keep track of resource
16        updateMemory(frontiers.size() + reachedFrom.size());
17        incrementNodeCount();
18        // retrieve from queue
19        SearchNode current = frontiers.poll();
20        // discard the node if a shorter one is visitedxx
21        if (visited.containsKey(current)
22            && visited.get(current) <= current.priority())
23            continue;
24        // mark the goal
25        if (current.goalTest())
26            return backchain(current, reachedFrom);
27        // keep adding the frontiers and update visited
28        ArrayList<SearchNode> successors = current.getSuccessors();
29        for (SearchNode n : successors) {
30            if (!visited.containsKey(n) || visited.get(n) > n.priority()) {
31                reachedFrom.put(n, current);
32                visited.put(n, n.priority());
33                frontiers.add(n);
34            }
35        }
36    }
37    return null;
38 }
```

**Line 21-26:** After popping the node, we check for two condition. One is if it is the goal, we simply return the solution path and terminate the search. Another condition is, if the node has been visited before, we don't push it into *frontiers* unless it has shorter cost than before.

**Line 28-36:** Get the successors of current node, and push those un-visited nodes or node has shorter cost than before into the *frontiers*.

## 2.3 Simple research on cost and heuristic

A\* leverage both the cost (penalty) and the heuristic (search speed). Currently we use  $f = h + g$  for priority, which seems quite a balanced solution. What happens if we go to extrem where the priority only related to  $g$  or  $h$ ? Or, is 50:50 the best choice for  $f$ ?

Here I modify the expression of priority as following:

$$f = \alpha \cdot h + (1 - \alpha) \cdot g$$

The basic idea of `getSuccessors` is to traverse through every possible state that is constraint by the problem definition (line 6-8), such as total number of missionary and carnivals, the size of the boat. After getting a new state, it immediately checks if it is legal (line 11-12). Finally it put the state node into successors list if it is safe.

The order of getting successors is vital to depth first search. I did optimize the codes for DFS, so that during each action it can go to the optimal state, which is transmitting as much people as possible. Simply by replacing line 6-7 as following:

```
for (int i = Math.min(state[2] * BOAT_SIZE, state[0]); i >= 0 ; i--) {  
    for (int j = Math.min(state[2] * BOAT_SIZE - i,  
        state[1]); j >= 0; j--) {
```

However, this might lose a lot fun, because by doing so dfs might perform as well as bfs. So I stick to the non-optimized one.

You may find that the `getSuccessors()` a little verbose. I did try to implement in a more contract way, but it makes the codes more difficult to understand. If you have any short and easy-understand expression, please do tell me.

I used a method called `isSafeState` that returns `true` if the state is legal. Basically it checks the legal state for both side of the river, which is the number of missionaries should not be less than carnivals'. Or if all the missionaries are at the one side, it is also safe. Here are the codes:

```
private boolean isSafeState(CannibalNode tryNode) {  
    if (tryNode.state[0] == 0 || tryNode.state[0] == totalMissionaries)  
        return true;  
    else if (tryNode.state[0] >= tryNode.state[1]  
        && (totalMissionaries - tryNode.state[0])  
        >= (totalCannibals - tryNode.state[1])  
        && tryNode.state[0] >= 0 && tryNode.state[1] >= 0)  
        return true;  
    else  
        return false;  
}
```

## 3 Breadth-first search

In the following BFS search, I use a queue to keep track of the nodes to be visited. How to determine whether a adjacent node should be put into the queue? I maintain a hash map for the nodes that have been visited.

Of course the while loop can terminate in two cases, One is when the the current node is the goal, which means we have reached the destination. In this case I use backchain to build the path. The other is when the queue is empty, which means we've already search the whole connected graph and yet goal is not found. In this case I return null.

```

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

3
    UUSearchNode node;
5    HashMap<UUSearchNode, UUSearchNode> visited = new HashMap<>();
    Queue<UUSearchNode> nqueue = new LinkedList<UUSearchNode>();
7    List<UUSearchNode> successors;

9    nqueue.add(startNode);
    while (!nqueue.isEmpty()) {
11        // get node from the queue
        node = nqueue.poll();

13
        // check if arrives destination
15        if (node.goalTest()) {
            updateMemory(visited.size() + 1); // add the start node
17            nodesExplored = visited.size() + 1;
            return backchain(node, visited);
19        }

21        // if not destination, keep searching and tracking
        successors = node.getSuccessors();
23        for (UUSearchNode n : successors) {

25            if (!visited.containsValue(n)) {
                visited.put(n, node);
27                nqueue.add(n);
            }
29        }
    }
31    // if destination not found, return null
    return null;
33 }

```

With the input of (3,3,1), the output is as following. The path is in a reverse style:

```

1  bfs path length:  12 [0, 111, 10, 31, 20, 221, 110, 311, 300, 321, 220, 331]
Nodes explored during last search:  15
3  Maximum memory usage during last search 15
Total execution time: 0.6 seconds

```

As you can see the path is consistent with my drawing in Figure ??.

With the input of (8,5,1), the output is as following:

```

1  bfs path length:  24 [0, 111, 10, 211, 110, 311, 210, 321, 220, 421, 320,
2      431, 330, 531, 430, 541, 440, 641, 540, 651, 550, 751, 650, 851]
Nodes explored during last search:  62
4  Maximum memory usage during last search 62

```

## 4 Memoizing depth-first search

DFS in recursive style is fairly simple, though I am understand why the provided code indicate me to use **HashMap**<UUSearchNode, Integer> **visited**, while it could be better done by using **HashSet**.

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

    // you write this method. Comments *must* clearly show the
    // "base case" and "recursive case" that any recursive function has.

    //System.out.println(currentNode);
    List<UUSearchNode> tryPath, path = new ArrayList<UUSearchNode>(
        Arrays.asList(currentNode));
    List<UUSearchNode> successors;

    visited.put(currentNode, depth);
    if (depth > maxDepth)
        return null;

    if (currentNode.goalTest()) {
        return path;
    } else {
        successors = currentNode.getSuccessors();
        for (UUSearchNode n : successors) {
            if (!visited.containsKey(n) || visited.get(n) > depth + 1) {
                tryPath = dfsrm(n, visited, depth + 1, maxDepth);
                if (tryPath != null) {
                    path.addAll(tryPath);
                    return path;
                }
            }
        }
    }
    return null;
}
```

The basic idea is, inside dfs, we run dfs for each legal node, and keep track of each node that have been visited. The trick of using **HashMap** is, in depth limited search, a node may fail due to the limitation of depth, not because there is a dead end. **HashMap** keeps tracking of the shortest path to the node, so dfs can find the solution as much as it can.

With the input of (8,5,1), the output is as following:

```

1 dfs memoizing path length:34 [851, 830, 841, 820, 831, 630, 641, 540, 551,
   530, 541, 520, 531, 430, 441, 420, 431, 410, 421, 320, 331, 310, 321, 300,
3   311, 210, 221, 200, 211, 100, 111, 10, 21, 0]
Nodes explored during last search: 40
5 Maximum memory usage during last search 39
Total execution time: 0.6 seconds

```

As you can see, memorizing dfs use less memory while finding a less optimal path comparing to bfs. Actually in the worst case, when the path is not found until the last try. Whether it uses less memory when comparing to, depends on the situation that whether the solution is far away from the start node, and whether dfs is using a better algorithm for searching.

What's more, dfs do not guarantee an optimal solution/goal, unless you can prove that your search algorithm is optimal.

## 5 Path-checking depth-first search

```

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

6
private List<UUSearchNode> dfsrpc(UUSearchNode currentNode,
8   HashSet<UUSearchNode> currentPath, int depth, int maxDepth) {

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

14   List<UUSearchNode> successors, tryPath, path
   = new ArrayList<UUSearchNode>(Arrays.asList(currentNode));
16

   //System.out.println(currentNode);
18   if (depth > maxDepth)
       return null;
20   else
       currentPath.add(currentNode);

22

   // This is base case where search reaches the goal
24   if (currentNode.goalTest()) {
       return path;
26   } else {

28       successors = currentNode.getSuccessors();
       for (UUSearchNode n : successors) {
30           if (!currentPath.contains(n)) {
               // This is the recursive function
32               tryPath = dfsrpc(n, currentPath, depth + 1, maxDepth);
               if (tryPath != null) {
34                   path.addAll(tryPath);

```

```

36         return path;
37     }
38 }
39 }
40 currentPath.remove(currentNode);
41 return null;
42 }

```

The basic idea is similar to memorizing dfs. The only difference is when rolling the path, it marked the node as not visited. We don't have to use **HashMap** here, because any node being rolled is legal to revisit again.

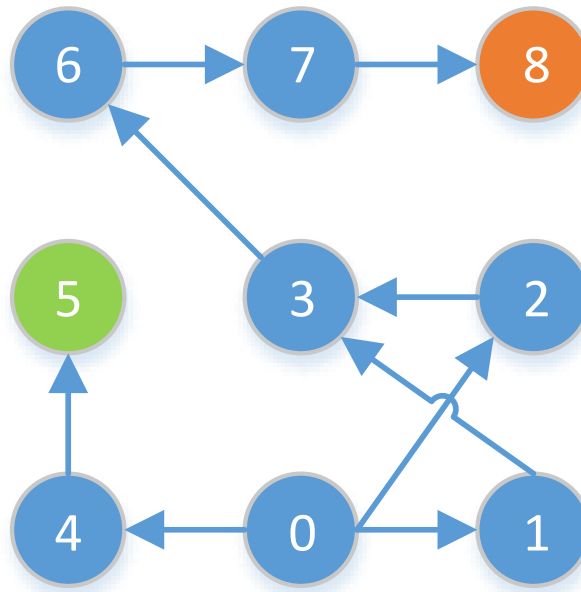


Figure 2: An example that demonstrate difference between memorizing and path-checking dfs

This may cause some redundance when searching the graph. Take Figure 2 for example, in memorizing, if we first check  $(0 \rightarrow 1 \rightarrow 3 \rightarrow 6 \rightarrow 7 \rightarrow 8)$ , and turn out to be dead end. Then we check  $(0 \rightarrow 2)$ , but we won't go further anymore, since node 3 is visited. But in path-checking, it probably check  $(0 \rightarrow 1 \rightarrow 3 \rightarrow 6 \rightarrow 7 \rightarrow 8)$ ,  $(0 \rightarrow 1 \rightarrow 2 \rightarrow 6 \rightarrow 7 \rightarrow 8)$ , and finally  $(0 \rightarrow 4 \rightarrow 5)$ .

An obvious advantage of path-checking over memorizing is using less memory, the maximum possible memory used is the longest path, which is usually much smaller the size of graph.

With the input of (8,5,1), the output is as following:

```

dfs path checking path length:34 [851, 830, 841, 820, 831, 630, 641, 540,
2  551, 530, 541, 520, 531, 430, 441, 420, 431, 410, 421, 320, 331, 310,
   321, 300, 311, 210, 221, 200, 211, 100, 111, 10, 21, 0]
4 Nodes explored during last search: 40
Maximum memory usage during last search 33
6 Total execution time: 0.6 seconds

```

## 6 Iterative deepening search

```
public List<UUSearchNode> IDSearch(int maxDepth) {  
2   resetStats();  
   HashSet<UUSearchNode> currentPath = new HashSet<UUSearchNode>();  
4   List<UUSearchNode> path;  
   for (int i = 1; i <= maxDepth; i++) {  
6       currentPath.clear();  
       path = dfsrpc(startNode, currentPath, 0, i);  
8       if (path != null)  
           return path;  
10  }  
   return null;  
12 }
```

The basic idea is use several path-checking dfs, while gradually increasing the maximum search depth, until we reach the goal. The problem is, should we choose memorizing or path-checking style dfs as the iterative function? In terms of memory, no doubt that path-checking will be very likely use less memory, since memorizing need to store all the visited nodes. While in terms of time, memorizing may save some time, when there are two path overlap and path-checking need to re-visit it.

But, on the other hand, if the goal is not far away from start point, or there is no optimal searching algorithm for dfs, there are high chances that ids with memorizing dfs performs worse than simply bfs, in terms of both memory (also need to save a lot of nodes, and also need extra space in the stack for recursive function) and time. So, we might have just well using bfs. In short, there is no strong reason to use memorizing as iterative function in ids.

With the input of (8,5,1), the output is as following:

```
Iterative deepening (path checking) path length:24 [851, 830, 841, 820,  
2   831, 630, 641, 530, 541, 520, 531, 420, 431, 410, 421, 310, 321, 300,  
   311, 200, 211, 100, 111, 0]  
4 Nodes explored during last search: 38918999  
Maximum memory usage during last search 24  
6 Total execution time: 4.953 seconds
```

You can see that ids use less memory than bfs, while also returning the best path.

## 7 Lossy missionaries and cannibals

In this situation, no more than  $E$  missionaries can be eaten. Then I can use a new state to describe:  $(m, c, b, e)$ , where  $m, c, b$  is the same as before,  $e$  represent whether the missionaries are eatable. I will basically change the conditions in **isSafe** depends on how many missionaries have already been eaten, if  $n_e$  exceeds  $E$ , I will set  $e = false$ , as no more missionaries should be eaten. An upper bound should be like this:

$$b_{upper} = \sum_{e=0}^E [(n_m + 1 - e)(n_c + 1)(n_b + 1) - 2]$$