# Mazeworld Solution

Junjie Guan <gjj@cs.dartmouth.edu>

January 22, 2014

## Contents



## 1 Introduction

Sovling maze is one of the most classic and popular problems in Artificial Intelligence. This report majorly cover three parts, 1) indroducing 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 anohter value called heuristic. Figure 1 is a demostration of A* algorithm. On one hand, the solid line represent the path that we alredy 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:

$$f = g + h$$

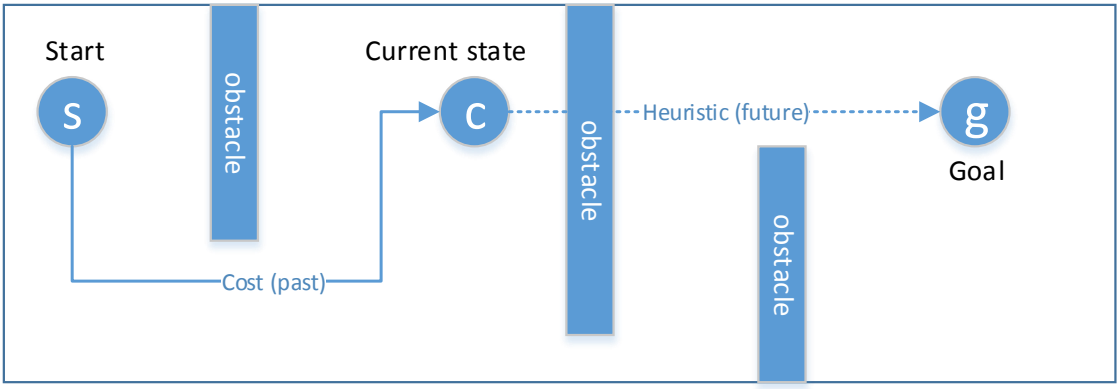


Figure 1: A demostration of A* algorithm

## 2.2 Single robot problem model

The basic idea of single robot is to find a path from start position to the goal position. The state here is the coordinate of the robot, $(x, y)$.

## 2.3 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 prioirty. 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() {
    resetStats();
3   // implementing priority queue for the frontiers
    PriorityQueue<SearchNode> frontiers = new PriorityQueue<>();
```

```
5     // implementing hashmap for the chain and the visited nodes
      HashMap<SearchNode, SearchNode> reachedFrom = new HashMap<>();
7     HashMap<SearchNode, Double> visited = new HashMap<>();

9     // initiate the visited with startnode
      reachedFrom.put(startNode, null);
11    // initiate the frontier
      frontiers.add(startNode);
13    while (!frontiers.isEmpty()) {
        // keep track of resource
15      updateMemory(frontiers.size() + reachedFrom.size());
        incrementNodeCount();
17      // retrieve from queue
        SearchNode current = frontiers.poll();
19      // discard the node if a shorter one is visited
        if (visited.containsKey(current)
21          && visited.get(current) <= current.priority())
          continue;
23      else
          visited.put(current, current.priority());
25      // mark the goal
        if (current.goalTest())
27        return backchain(current, reachedFrom);
        // keep adding the frontiers and update visited
29      ArrayList<SearchNode> successors = current.getSuccessors();
        for (SearchNode n : successors) {
31        if (!visited.containsKey(n) || visited.get(n) > n.priority()) {
            reachedFrom.put(n, current);
33          frontiers.add(n);
          }
35      }
      }
37    return null;
}
```

**Line 20-28:** After poping the node, we check for two condition. One is if it is the goal, we simply return the solution path and terminate the search. Antoher condition is, if the node has been visited before, we don't push it into *frontiers* unless it has shorter cost than before.

**Line 29-35:** Get the successors of currrent node, and push those un-visited nodes or node has shorter cost than before, into the *frontiers*.

## 2.4 Output demonstration

I use Simplex Noise to generate the maze.[1] Figure 5 shows a $40 \times 40$ maze, where all three robots try to move from bottom-left to middle-right. I leave the direction of robot at every single node. You can see that bfs and A* perform quite almost equally well, while dfs goes through a lengthy path.

The following output shows that, A* explored significantly less node than bfs, while the path length (cost) almost as less as bfs, while comparing to dfs.

```
BFS:
2   Nodes explored during search:  1064
```

[1] I use authoried code from http://webstaff.itn.liu.se/ stegu/simplexnoise/SimplexNoise.java to help me with the noise

```
  Maximum space usage during search 1099
  path length: 60

DFS:
  Nodes explored during search:  399
  Maximum space usage during search 242
  path length: 242

A*:
  Nodes explored during search:  79
  Maximum space usage during search 224
  path length: 72
```

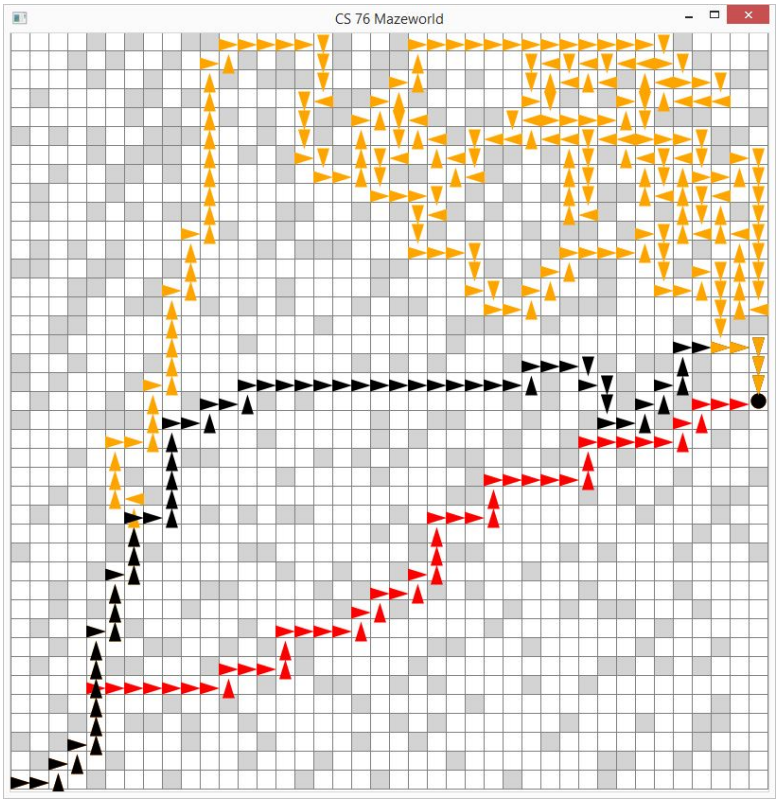


Figure 2: Searhing path of dfs(orange), bfs(red) and A*(black)

## 2.5 Discussion 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 $f$?

Before I begin to discuss, I want to introduce a new kind of maze, where the obstacles can be crossed, while there is a certain amount of penalty. While I am first use Simplex Noise generate a maze, i replace the wall with number ranging from 1 to 10 to represent the weights. I use the following functions to differentiate

nodes with different weights.

$$Grayscale = 255 - 255 \times weight/20$$

Here I modify the expression of priority as following:

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

Then I vary $\alpha$ from 0 to 1, and observe their path length and cost. Figure 7 shows the visual path with different configuration.

- Red ($\alpha = 0.0$) is an extreme case when A* only considers the cost, and becomes uninformed search. (I think it act like Dijkstra Algorithm). Since we are using Manhattan distance and the goal is at top-right, every action of moving south or west is a compromise of cost, and neglecting of path length ( search speed).
- Brown ($\alpha = 1.0$) is another extreme case when A* only considers the heuristic, and becomes best-first search. We can observe that it totally neglect the cost/penalty, rushing to the goal in the simplest path.

Figure 6 shows the change of path length and cost while varying $\alpha$. It depends on the tradeoff on searching speed and cost. It also greatly related to the problem definition, which affects the apperance of state space. For example, if path length and cost are on the scale $a : b$, then we should find $\alpha$, s.t. $min(a \cdot cost + b \cdot length)$.

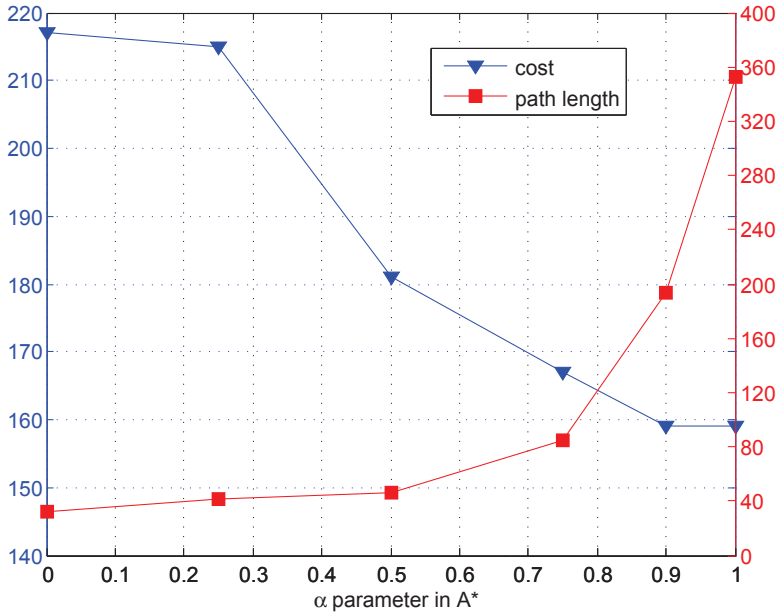


Figure 3: Plot describes the relationship between cost, path length, and $\alpha$

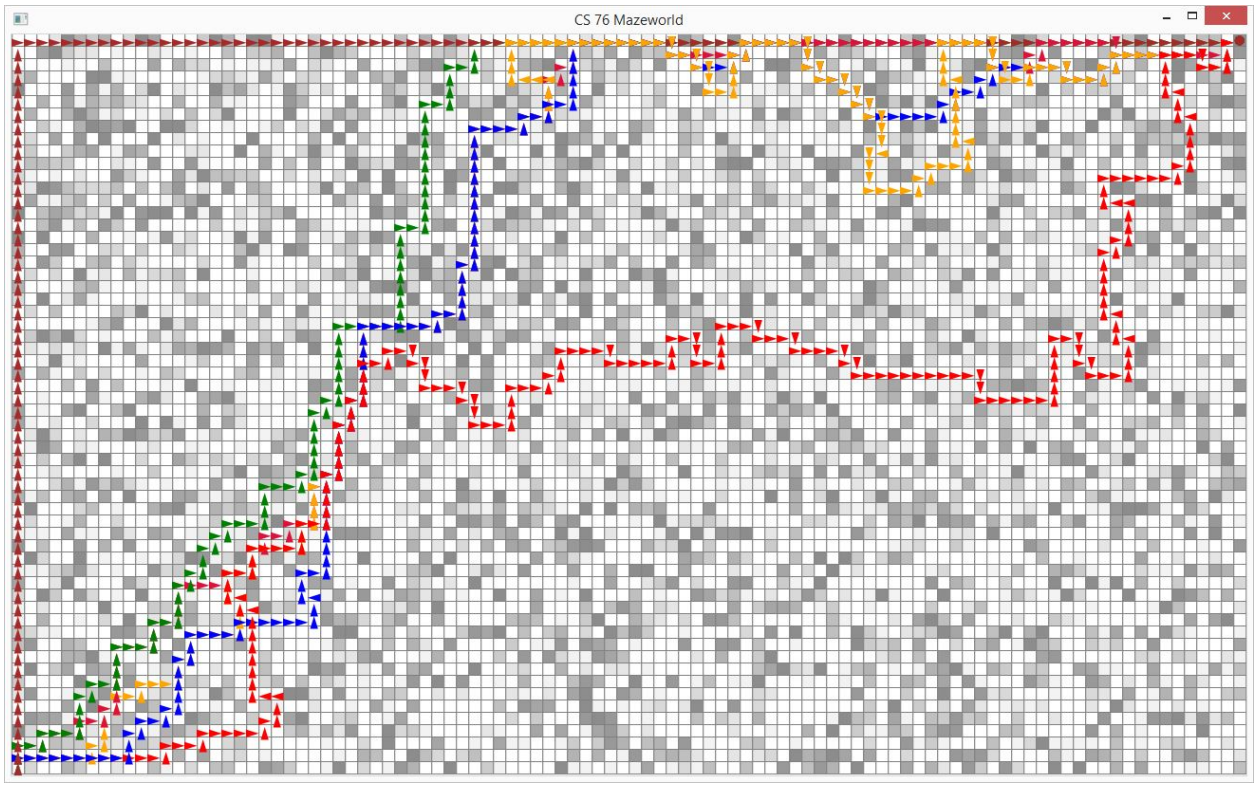


Figure 4: Varying the $\alpha$ in A* as 0.0 (red), 0.25 (orange), 0.5 (blue), 0.75 (crimson), 0.9 (green), 1.0 (brown)

# 3 Multirobot problem

## 3.1 Problem definition and states

The different between single robot problem and multi robot problem is, every robot take turns to make actions, and it needs collision detection. In thiscase, we decide the a whole state for all the $k$ robots, like this:

$$\begin{pmatrix} x_0 & y_0 \\ x_1 & y_1 \\ \vdots & \vdots \\ x_{k-1} & y_{k-1} \end{pmatrix}$$

However, this is not enough for the states. We still one parameter, which the turn of the robot. This parameter is not necessary, which means you can still solve the problem without it in the state. However, returning all the possible states of $k$ can be very redundant and time consumming for latter searching. If I have k robot, and five actions (4 directions plus not moving), the upper bound of states would be $5^k$. It is like giving too much options more next step, while I am not making any decision.

## 3.2 Code implementation

### 3.2.1 getSuccessors

**Line 4:** Iterate through all the 5 possible actions (4 directions plus not moving).

**Line 6-7:** Initiate the coordinates for the successor's state, noted that only the robot in turn can take action here.

**Line 11-13:** Construct the successor with new coordinates, and new cost based on whether it moves, and also keep looping the turn through $R$ robots.

```
1  public ArrayList<SearchNode> getSuccessors() {
2    ArrayList<SearchNode> successors = new ArrayList<SearchNode>();
3    Integer[] xNew = new Integer[R], yNew = new Integer[R];
4    for (int[] action : actions) {
5      for (int r = 0; r < R; r++) {
6        xNew[r] = robots[r][0] + action[0] * (r == turn ? 1 : 0);
7        yNew[r] = robots[r][1] + action[1] * (r == turn ? 1 : 0);
8      }
9      if (maze.isLegal(xNew[turn], yNew[turn])
10         && noCollision(xNew, yNew)) {
11       SearchNode succ = new MultirobotNode(xNew, yNew, getCost()
12           + Math.abs(action[0]) + Math.abs(action[1]),
13           (turn + 1) % R);
14       successors.add(succ);
15     }
16   }
17   return successors;
18 }
```

### 3.2.2 Ohter methods

Node constructor, it initiates the positions of robots iteratively.

```
public MultirobotNode(Integer[] x, Integer[] y, double c, int t) {
  robots = new int[R][2];
  for (int i = 0; i < R; i++) {
    this.robots[i][0] = x[i];
    this.robots[i][1] = y[i];
  }
  turn = t;
  cost = c;
}
```

**Line 20-28:** After poping the node, we check for two condition. One is if it is the goal, we simply return the solution path and terminate the search. Antoher condition is, if the node has been visited before, we don't push it into *frontiers* unless it has shorter cost than before.

**Line 29-35:** Get the successors of currrent node, and push those un-visited nodes or node has shorter cost than before, into the *frontiers*.

### 3.3 Output demonstration

I use Simplex Noise to generate the maze.[2] Figure 5 shows a $40 \times 40$ maze, where all three robots try to move from bottom-left to middle-right. I leave the direction of robot at every single node. You can see that bfs and A* perform quite almost equally well, while dfs goes through a lengthy path.

The following output shows that, A* explored significantly less node than bfs, while the path length (cost) almost as less as bfs, while comparing to dfs.

```
BFS:
  Nodes explored during search:  1064
  Maximum space usage during search 1099
  path length: 60

DFS:
  Nodes explored during search:  399
  Maximum space usage during search 242
  path length: 242

A*:
  Nodes explored during search:  79
  Maximum space usage during search 224
  path length: 72
```

### 3.4 Discussion 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 $f$?

Before I begin to discuss, I want to introduce a new kind of maze, where the obstacles can be crossed, while there is a certain amount of penalty. While I am first use Simplex Noise generate a maze, i replace the wall with number ranging from 1 to 10 to represent the weights. I use the following functions to differentiate nodes with different weights.

$$Grayscale = 255 - 255 \times weight/20$$

[2] I use authoried code from http://webstaff.itn.liu.se/ stegu/simplexnoise/SimplexNoise.java to help me with the noise

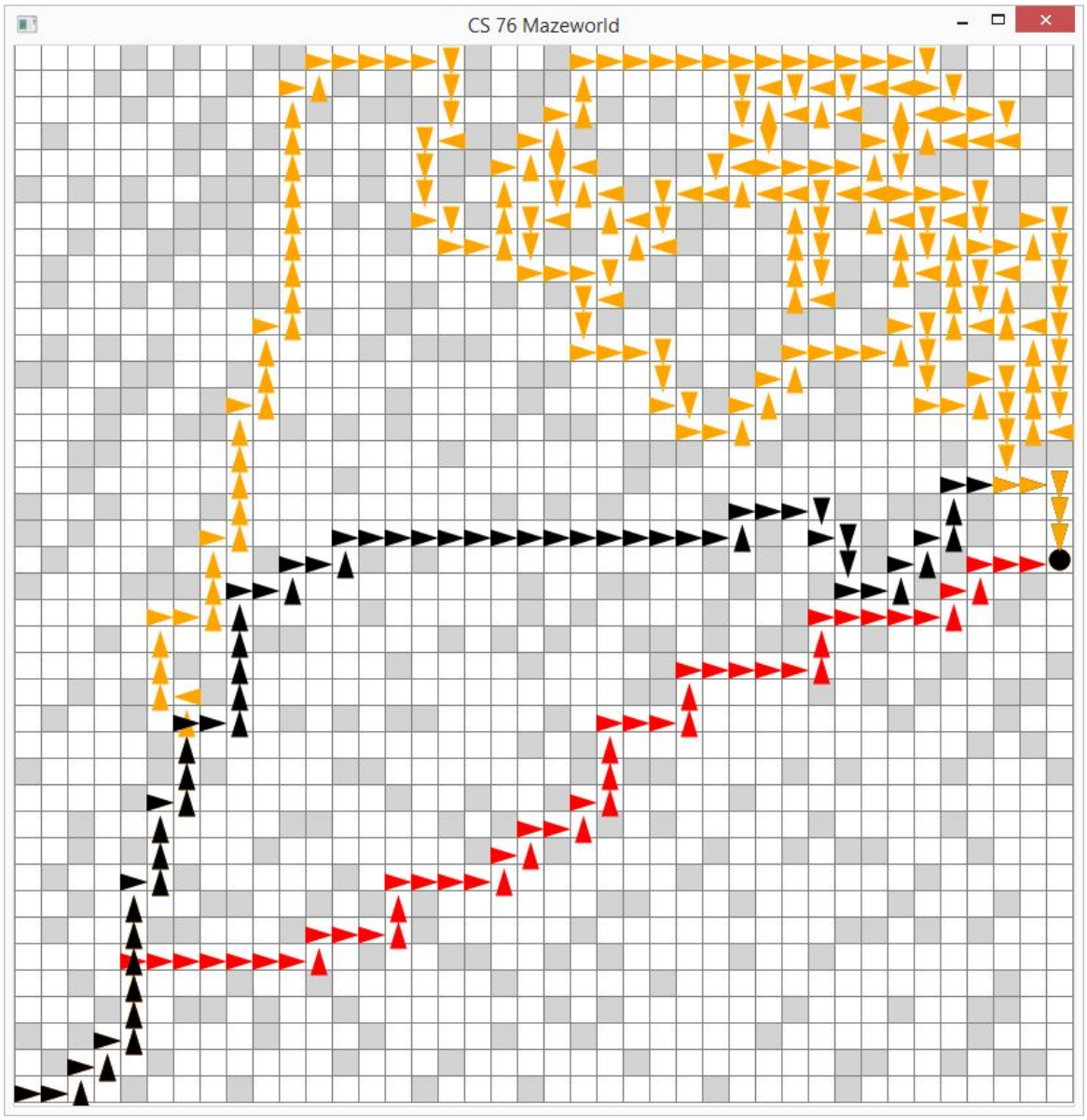


Figure 5: Searhing path of dfs(orange), bfs(red) and A*(black)

Here I modify the expression of priority as following:

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

Then I vary $\alpha$ from 0 to 1, and observe their path length and cost. Figure 7 shows the performance of different configuration.

- Red ($\alpha = 0.0$) is an extreme case when A* only considers the cost, and becomes uninformed search. (I think it act like Dijkstra Algorithm). Since we are using Manhattan distance and the goal is at top-right, every action of moving south or west is a compromise of cost, and neglecting of path length ( search speed).

- Brown ($\alpha = 1.0$) is another extreme case when A* only considers the heuristic, and becomes best-first search. We can observe that it totally neglect the cost/penalty, rushing to the goal in the simplest path.

Figure 6 shows the change of path length and cost while varying $\alpha$. Depending on whether

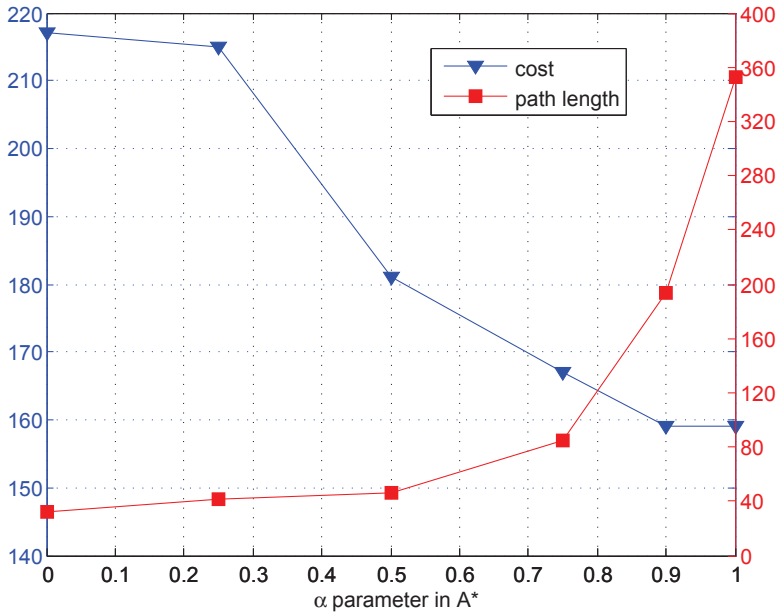


Figure 6: Plot describes the relationship between cost, path length, and $\alpha$

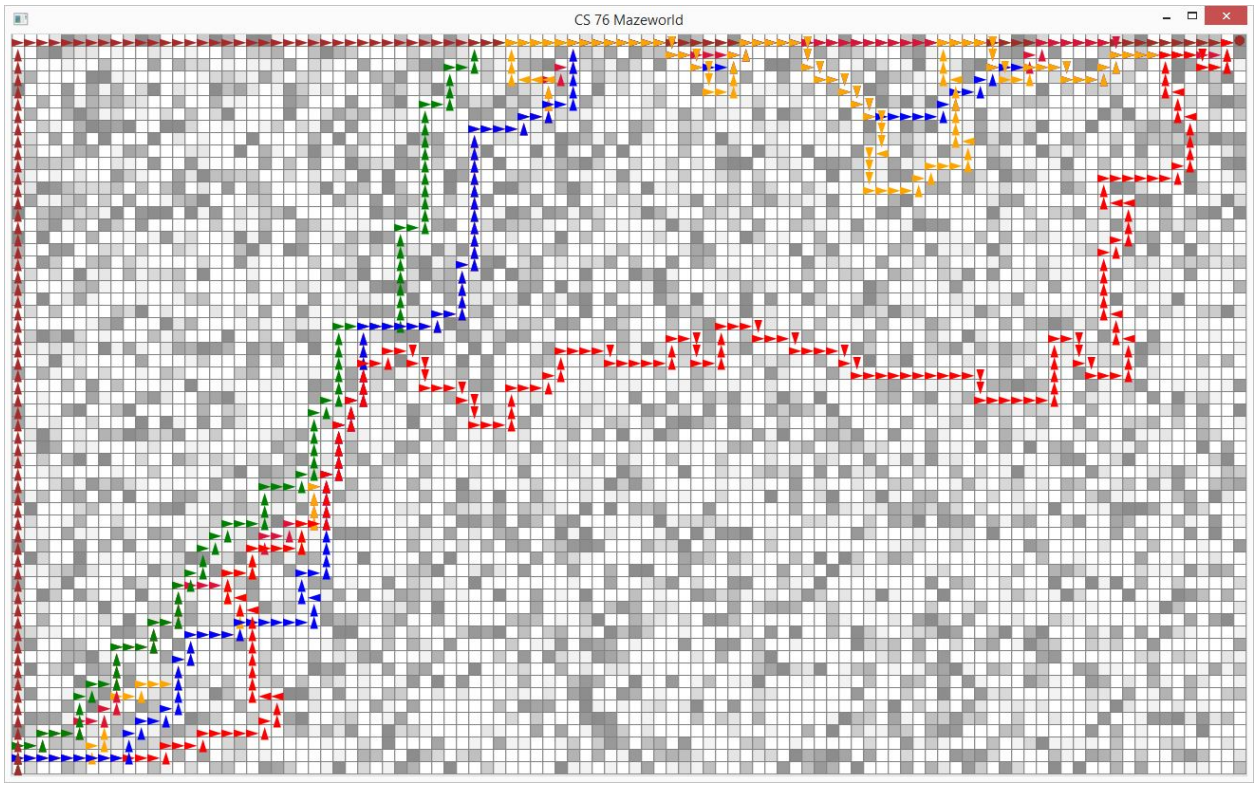


Figure 7: Varying the $\alpha$ in A* as 0.0 (red), 0.25 (orange), 0.5 (blue), 0.75 (crimson), 0.9 (green), 1.0 (brown)