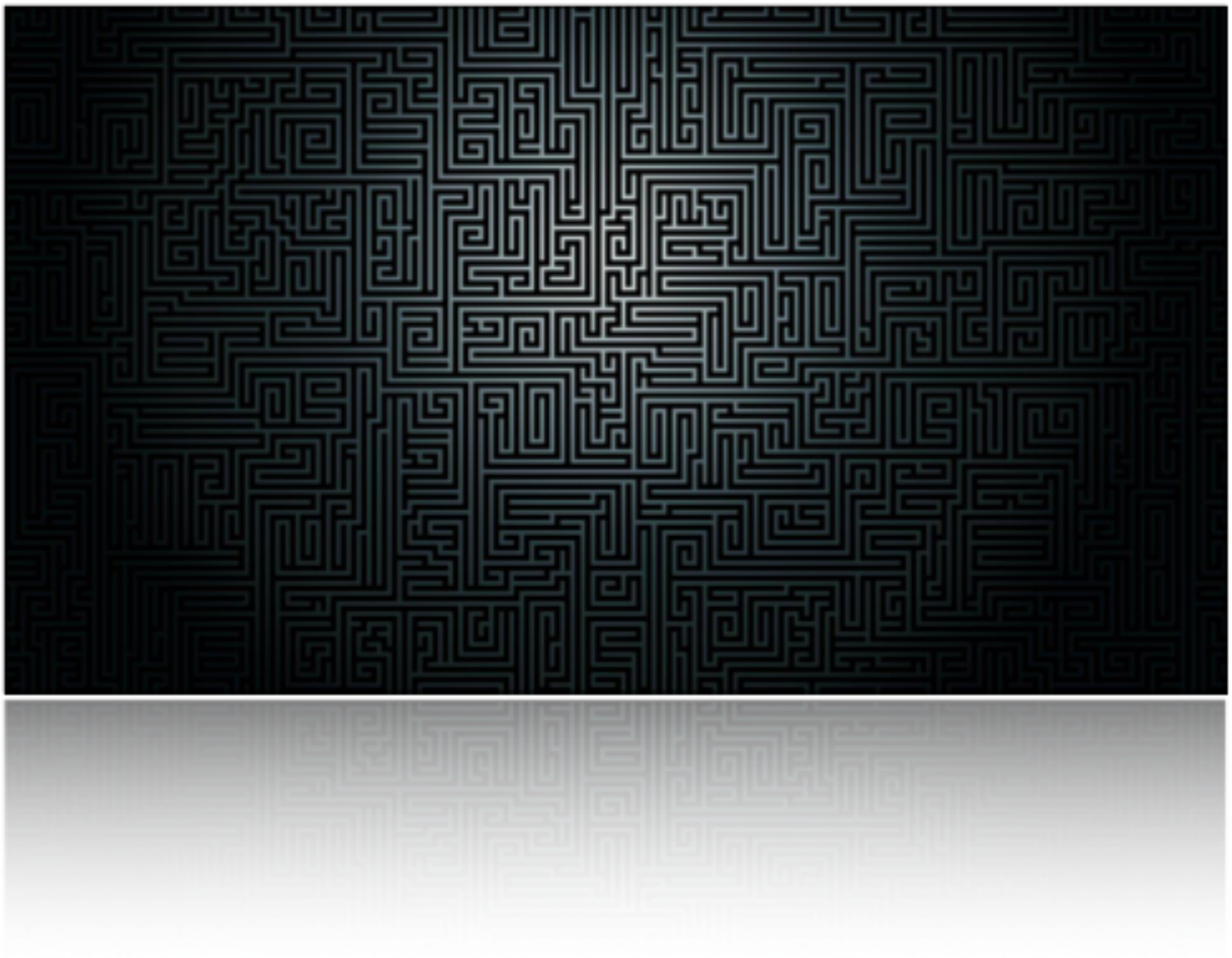
Search

CS 440 Assignment 1 Report

[Jinhao Hu, jhu41]   [Ruixi Zhang, rzhang76]   [Chen Liu, cliu110]

[31423 Q3]
[February 4, 2019]



Section I&II

For this assignment, we have implemented four search strategies — Breadth-first search, Depth-first search, Greedy best-first search, and A* search to solve puzzles. The puzzles are similar to that of the game Pac-man. The goal is that the agent has to find a path in the mazes which would consume all the dot(s) distributed throughout the maze along the way.

The workflow can be summarized as following:

Information
Organize
Instruct
Goal

The maze is the environment and the Pac-man is the agent. There are information about them that needs to be organized by data structures. Then algorithms are applied to carry out unambiguous instructions using these data structures to eventually achieve the goal.

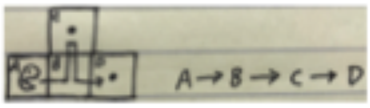
The environment is a fully observable, deterministic, discrete, static, and known one. The information of the environment includes but is not limited to: **(1)**the location (rows and columns) of each individual tile upon which the Pac-man could stand on; **(2)**how many objectives(food pellets) are in the maze; **(3)**the location of each objectives; **(4)**the starting position of the Pac-man; **(5)**which tile is walkable and which tile is a wall; etc.

There is only a single agent, the Pac-man. It does two things. First, it stands still and observe the maze to its fullest to find the appropriate path. Next, it walks that appropriate path to achieve the goal. We designed it in such way so that it possesses information includes but is not limited to: **(1)**its own location; **(2)**the previous tiles it walked on which got it to its current tile; **(3)**the cost (let's call it g(n)) to get it to its current tile from the starting position; **(4)**the manhattan distance from its current tile to the nearest tile containing the food pellet; **(5)**if the tile has been explored before; **(6)**the order for the exploration of the tiles **(7)**how many remaining pellets are in the maze; **(8)** how many tiles have we explored in total; etc.

We implemented various data structures to organize these information so that we could use different algorithms to inform the agent to take the appropriate actions. The column and row of the current tile, the cost, and the previous tiles the agent has walked on are all stored in an **object** named state. A state is basically how the environment, or how the interaction between the agent and the environment is, at that particular moment. We organize these components together because they are the most dynamic ones and they are the results of the action the agent took, not the thinking the agent was having in mind. We used **queue (or Lifo queue, priority queue**, depends on what algorithm to use later) to store certain states as frontiers. The frontiers are the states which the agent intends to search in the near future wether it is the immediate next state or not-so-immediate future state that the agent eventually wants to get to. One way to think about frontiers is that they are basically to-do-list, waiting to be visited next.

We also maintain an explored states list as a **dictionary** with the key being different states and value being true or false. All states in the maze have default value of false. If they are visited by Pac-man's search, the value will change to true. An objective list is also stored in memory as a **list**. The purpose of such list is to keep track of how many food pellets are remaining in the list. It is important because if there is no more food pellet, then the game is considered over and goal accomplished. Also, a **list** is created to include all the states that the Pac-man have to actually walk to eat all food pellets. We call it the path list, and the length of this list has to be returned at the end to prove that we have successfully solved the puzzle. At last, an **int**, number of nodes expanded, is incremented during the search process to keep track of the time complexity of the algorithm we devised to eat all the food pellets.

We treated node and state as the same thing in our implementation because upon visiting the new tile, the state of the environment changes anyway — this distinction by itself is enough to tell different nodes apart from each other, so we just use states to represent nodes. For the single-pellet maze, the condition for a state to be put into the frontier queue is that it must not be in the explored set, because we don't want to revisit and repeat unnecessary state. However, revisit is necessary in some cases in a multi-pellet maze. Consider the following situation:



The state B must be revisited for all pellets to be claimed. The way we handle such repetition is to set all the value to be false in the explored set immediately when a pellet is being claimed. In another word, we treat the last state in which we claimed the pellet to be a new start state and thus enable the agent to revisit the previous state(s) to not get stuck in the maze.

Section II

For the greedy BFS algorithm, we have established a priority queue to organize frontiers so that the frontier with the highest priority shall be visited and expanded first. The **heuristic value** we use to rank the priority of frontiers is the manhattan distance between the Pac-man and the food pellet in the single-pellet maze. And the **heuristic value** we use to rank the priority in a multi-pellet maze is the manhattan distance between the Pac-man and the nearest food pellet to the Pac-man. The shorter the distance, the higher it is in priority, and the more immediate we expand it.

For the A* algorithm, we have two different methods to solve the puzzle. We have built two helper function to deal with the single-pellet and multi-pellet system separately. For the single-pellet system, the heuristic value is still the manhattan distance between the Pac-man and the food pellet. The majority of the implementation is very similar to the greedy algorithm. However, there is a little twist. In each state, we have stored the **cost to reach this state so far** as c(r, c). We also stored the manhattan distance between the Pac-man and the food pellet as the **heuristic value**, h(r, c). We then add them up as the **total estimation cost**, T(r, c).

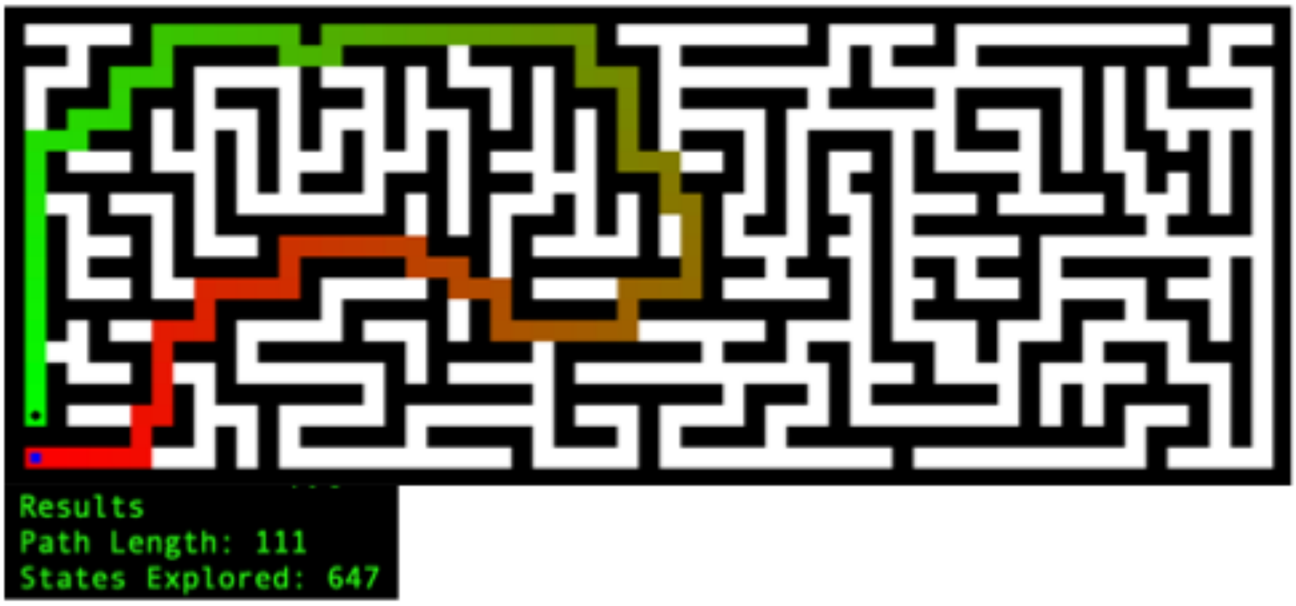
$$T(r, c) = c(r, c) + h(r, c)$$

Whenever a state is repeatedly appeared in the neighbor set and not allowed to put into the frontier state again, we would compare the total estimation cost of that state to the total estimation cost of that state that has already existed in the frontier list. If the new estimation cost is lower, we would then put that state in the frontier list to explore in the future because it would give us a shorter path than the path we have searched before. If not, we go on and ignore it.

For the multi-pellet system, the heuristic value and the cost calculation is the same as the single-pellet system. We used the minimum spinning tree to keep track of the most optimal search and eventually chose the one with the lowest total cost.

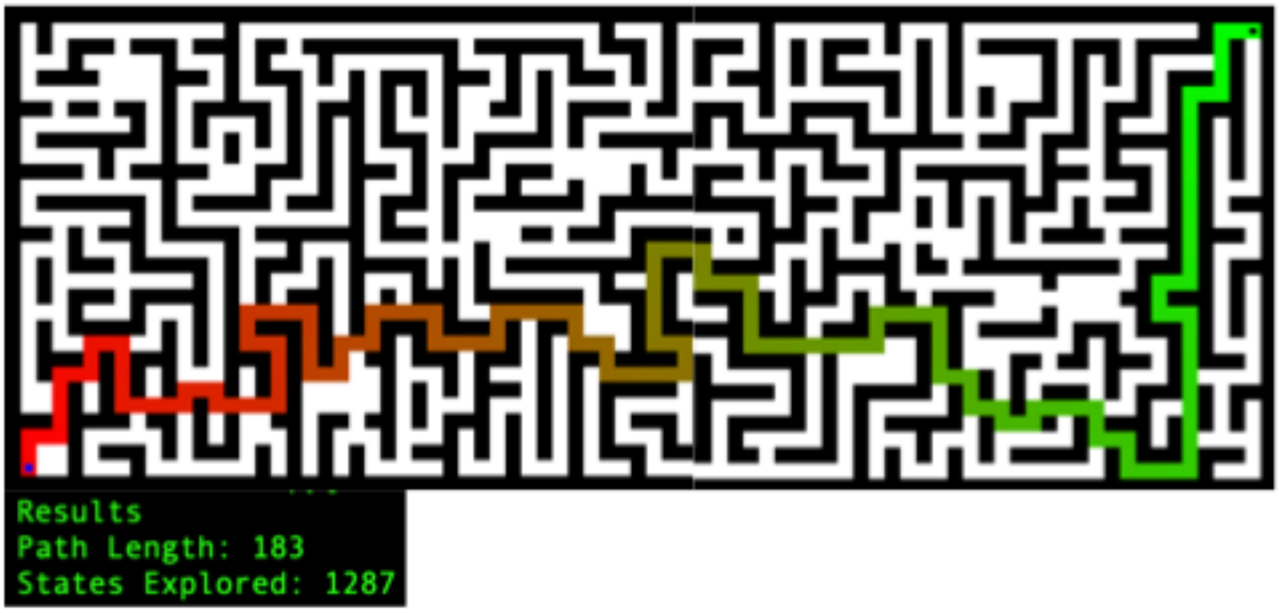
Section III

Screenshot 1.1 (bfs, medium maze)



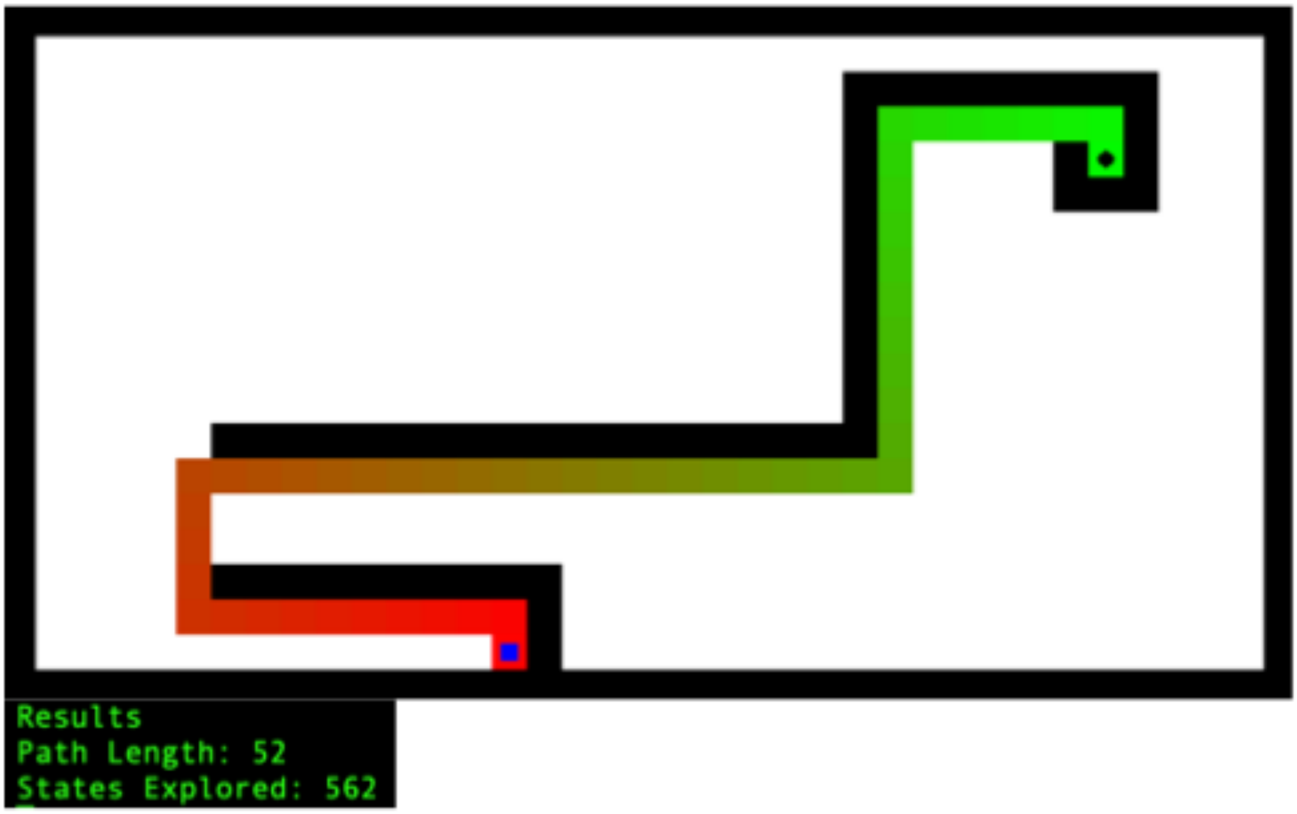
```
Results
Path Length: 111
States Explored: 647
```

Screenshot 1.2 (bfs, big maze)



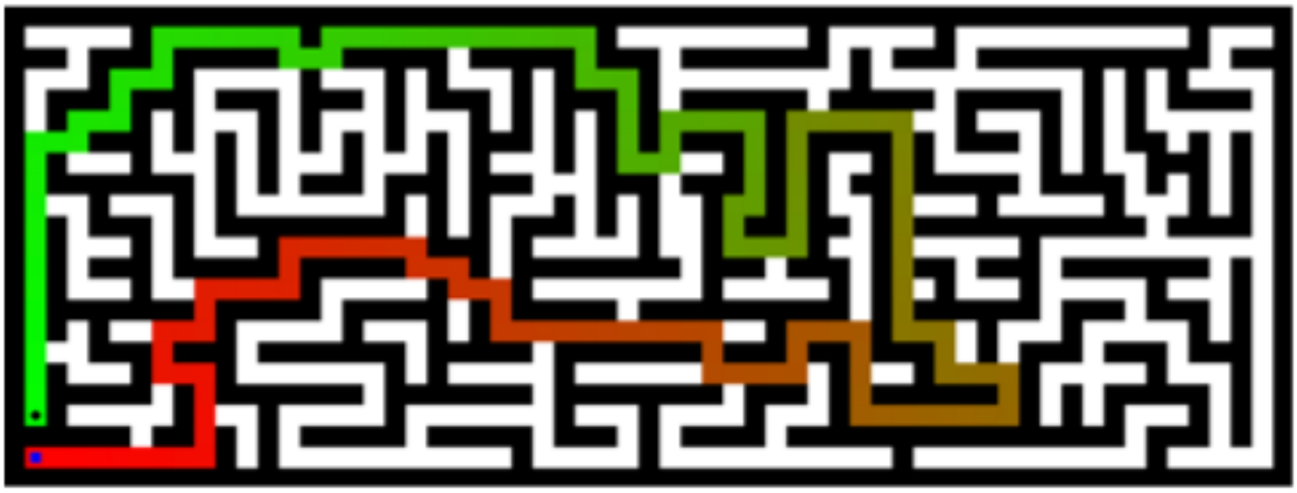
```
Results
Path Length: 183
States Explored: 1287
```

Screenshot 1.3 (bfs, open maze)

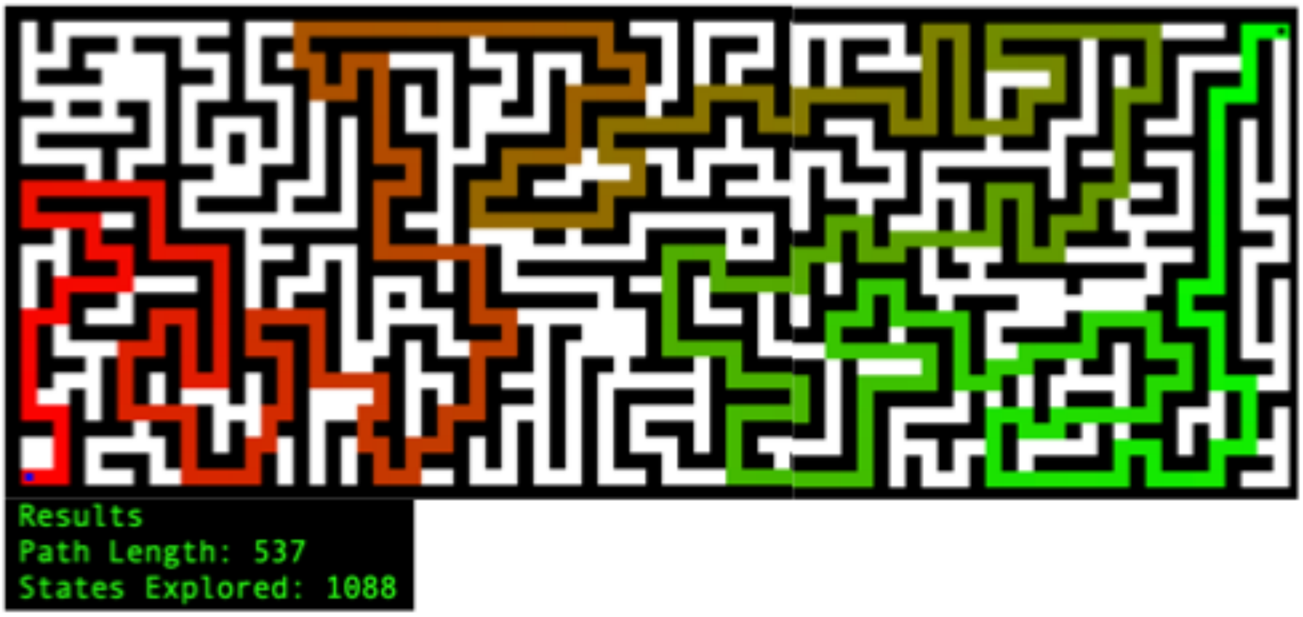


```
Results
Path Length: 52
States Explored: 562
```

Screenshot 2.1 (dfs, medium maze)

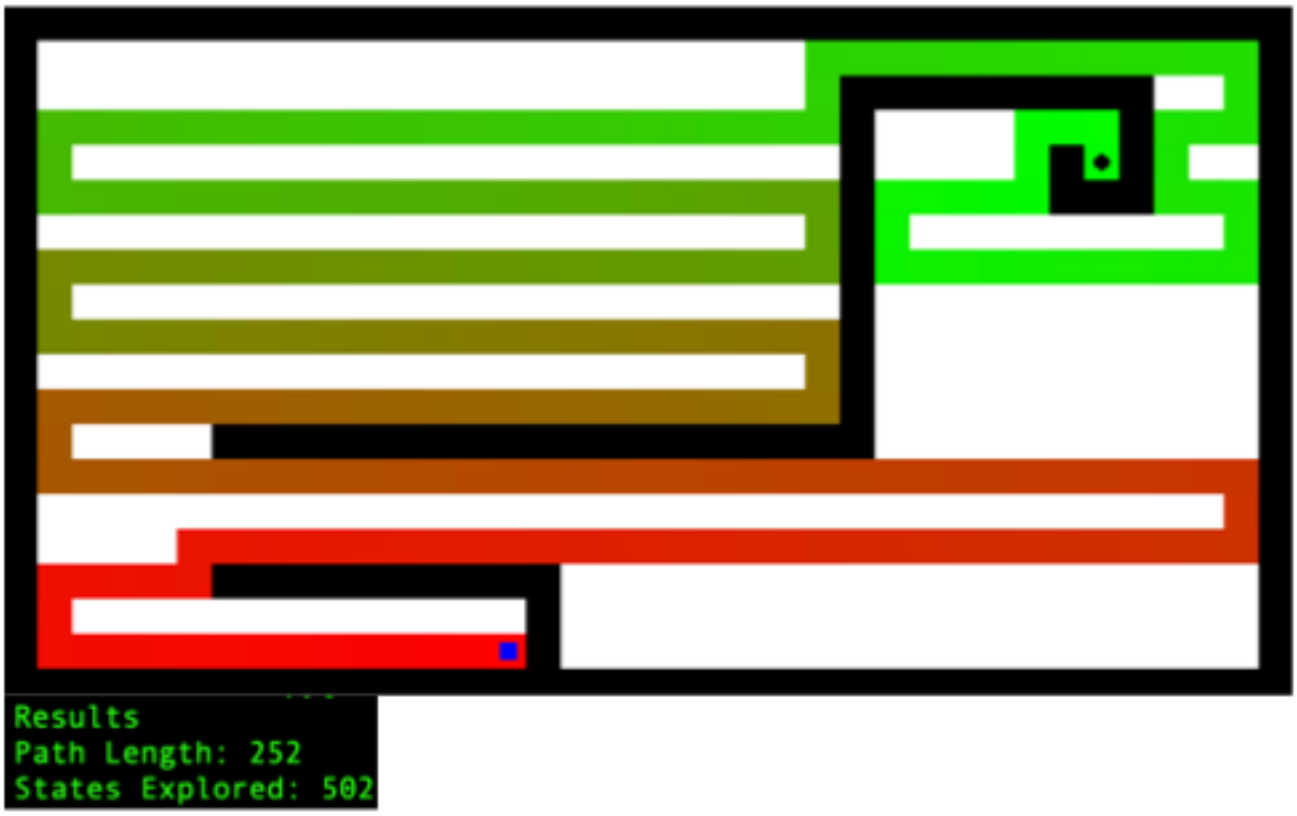


Screenshot 2.2 (dfs, big maze)



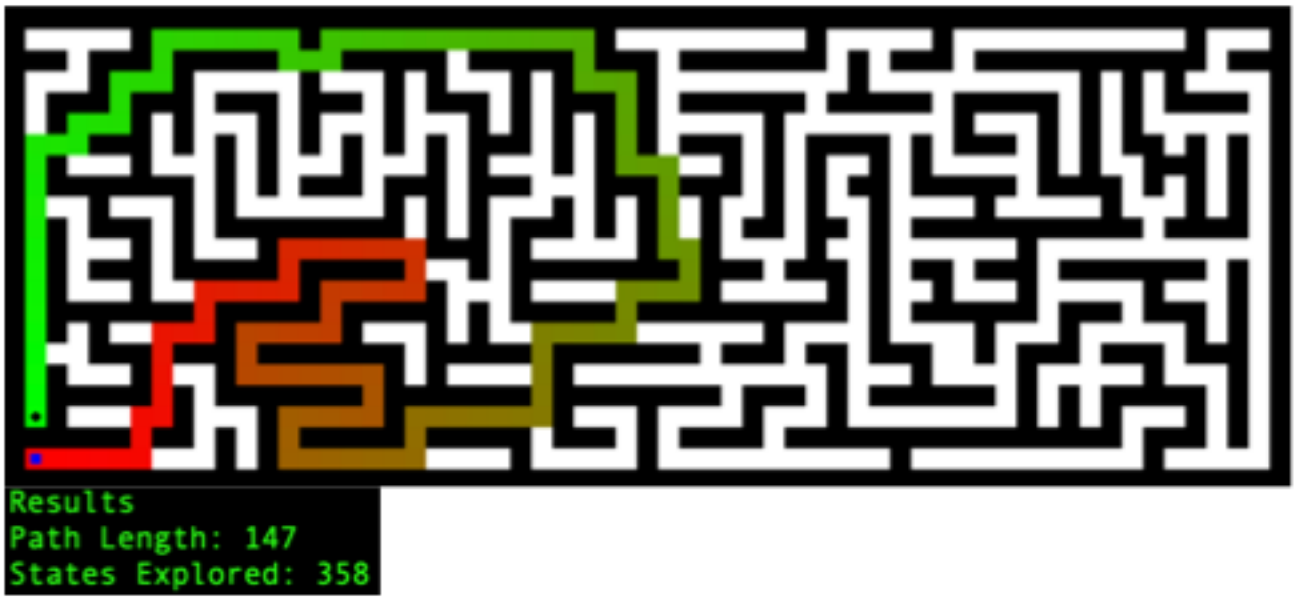
```
Results
Path Length: 537
States Explored: 1088
```

Screenshot 2.3 (dfs, open maze)



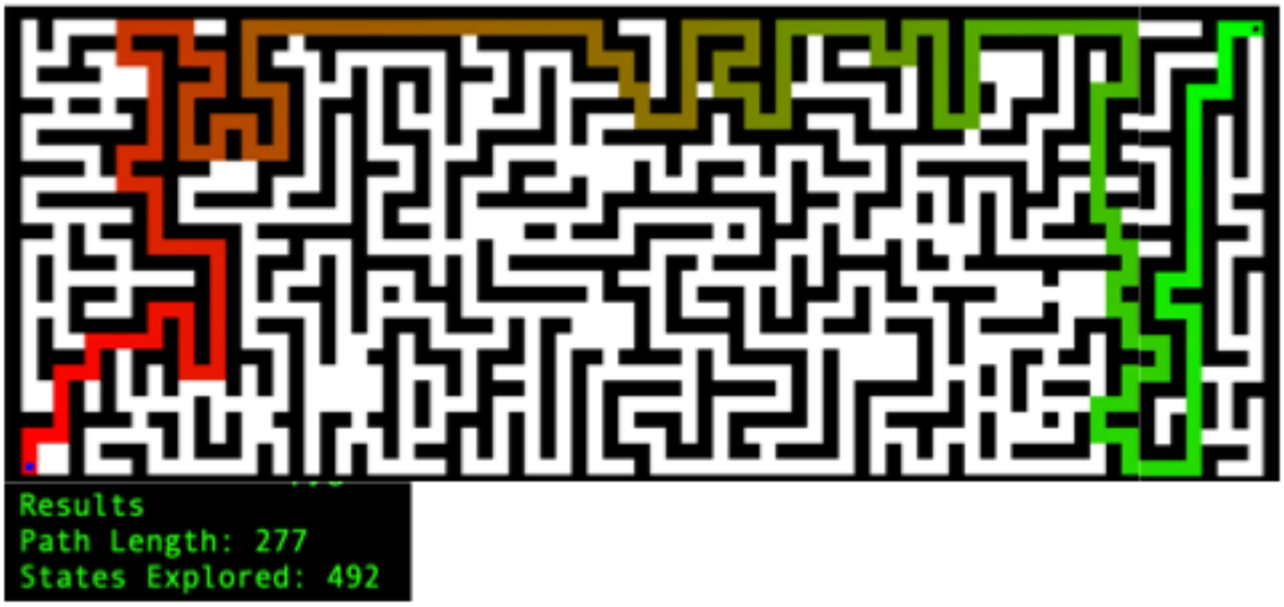
```
Results
Path Length: 252
States Explored: 502
```

Screenshot 3.1 (greedy, medium maze)



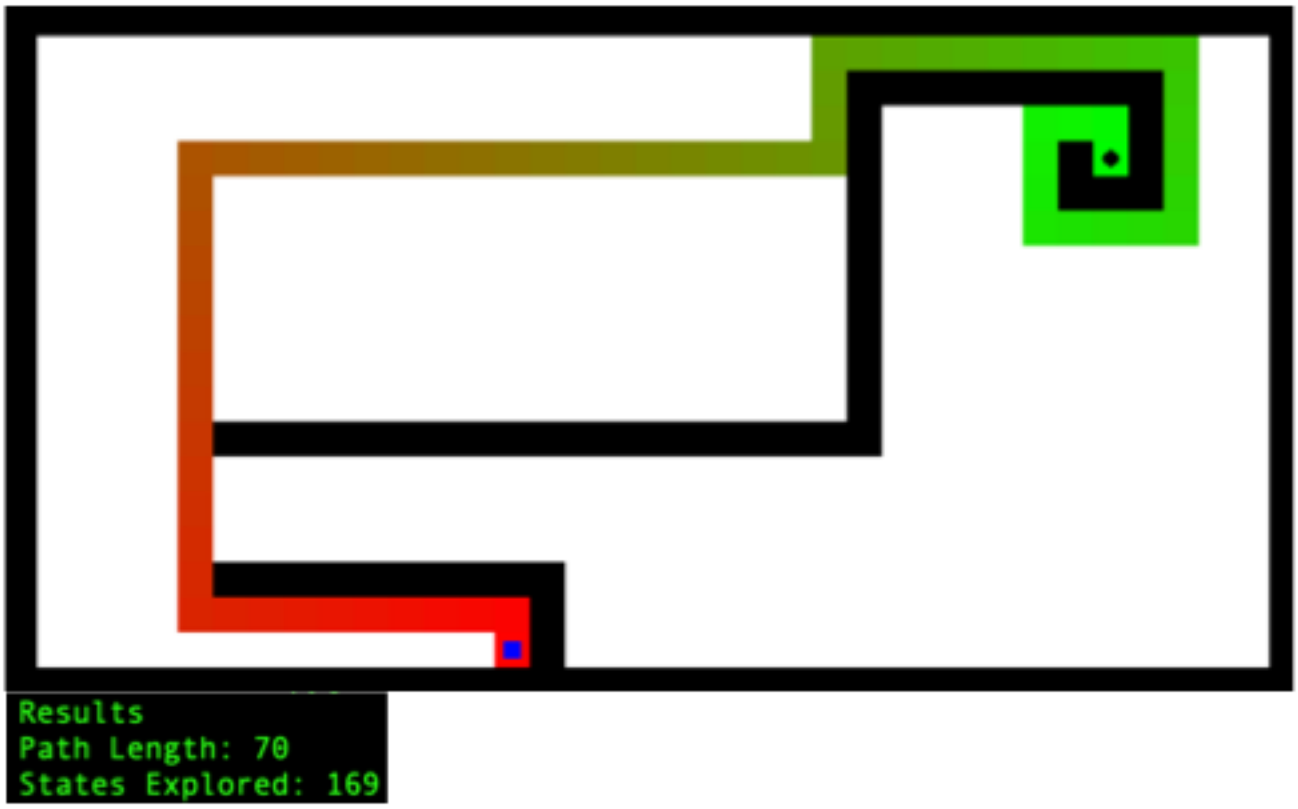
```
Results
Path Length: 147
States Explored: 358
```

Screenshot 3.2 (greedy, big maze)



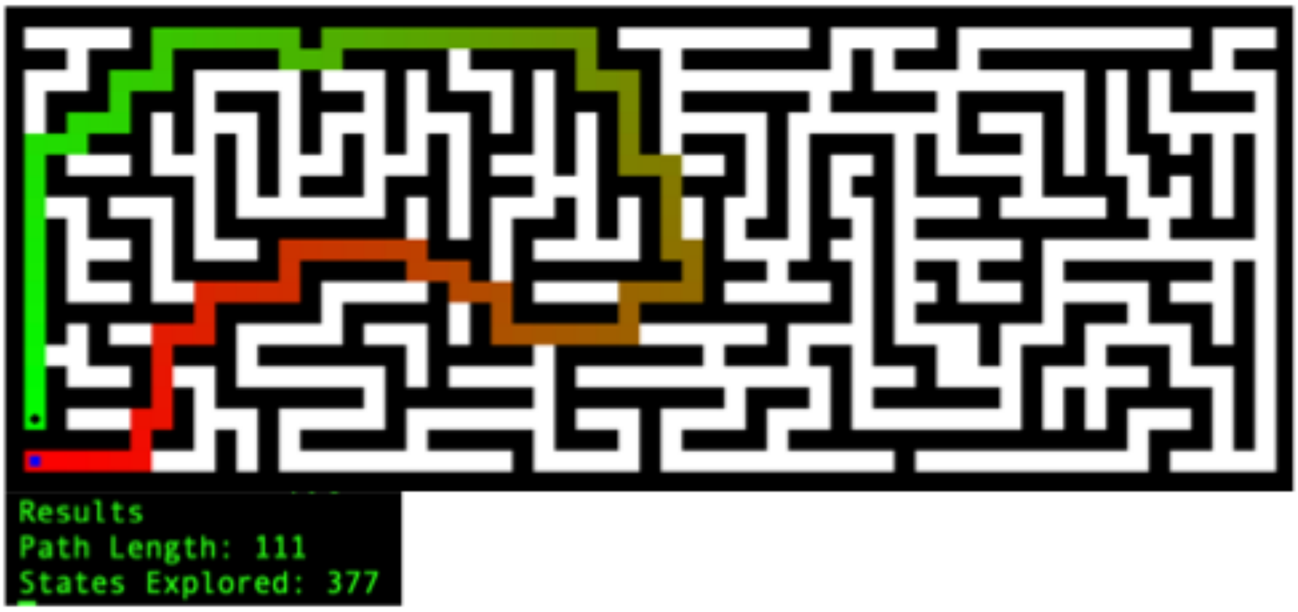
```
Results
Path Length: 277
States Explored: 492
```

Screenshot 3.3 (greedy, open maze)



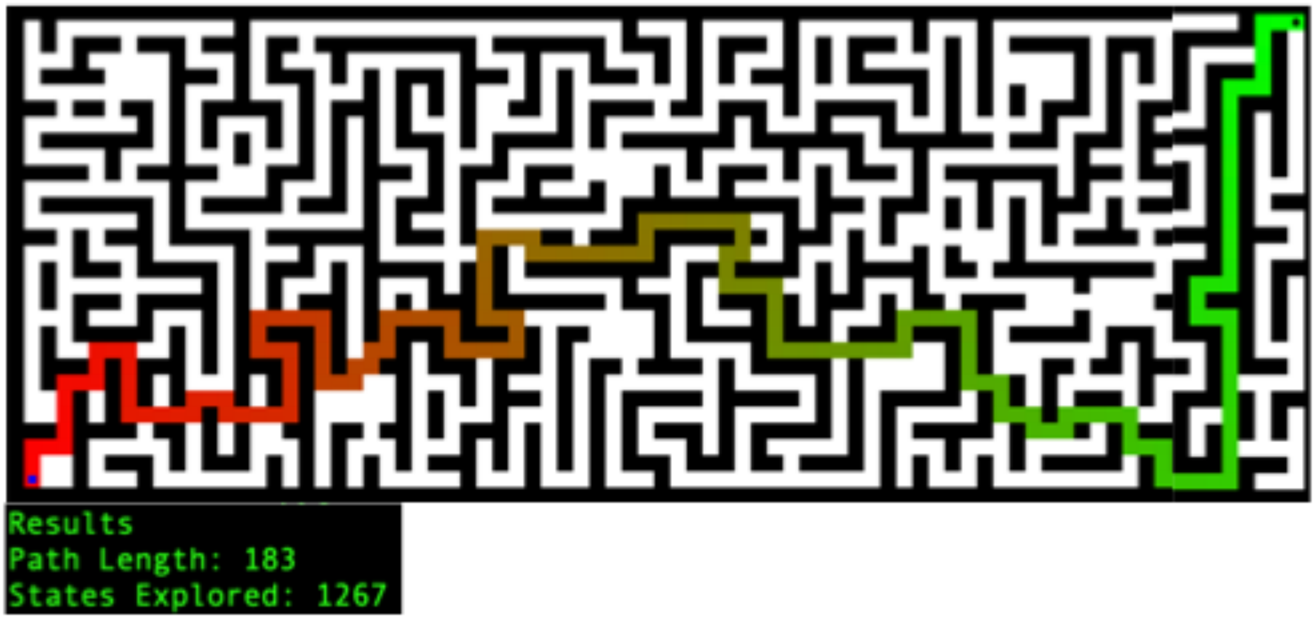
```
Results
Path Length: 70
States Explored: 169
```

Screenshot 4.1 (A*, medium maze)



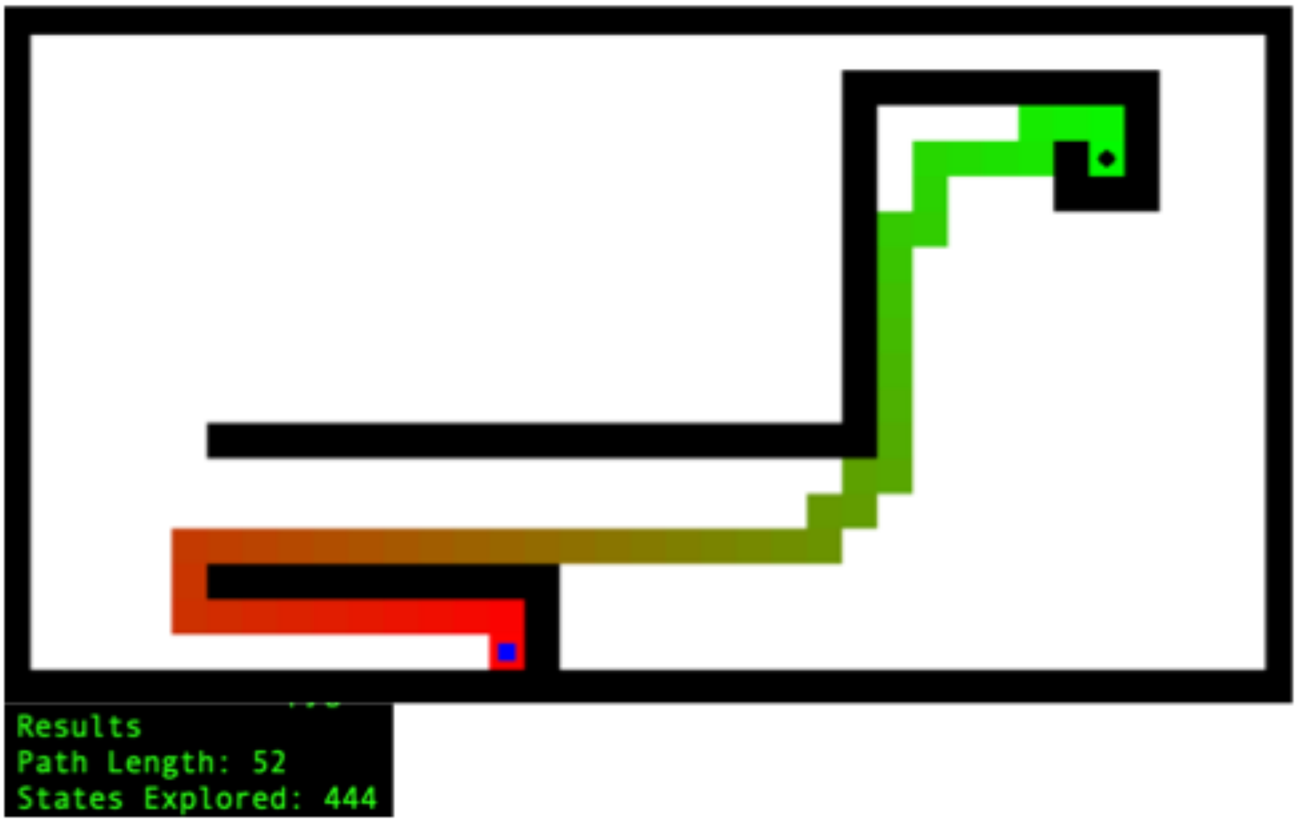
```
Results
Path Length: 111
States Explored: 377
```

Screenshot 4.1 (A*, big maze)



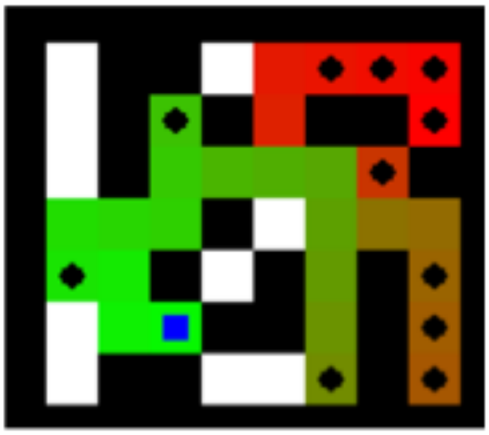
```
Results
Path Length: 183
States Explored: 1267
```

Screenshot 4.1 (A*, open maze)



```
Results
Path Length: 52
States Explored: 444
```

Screenshot 5.1 (A*, tiny Search)

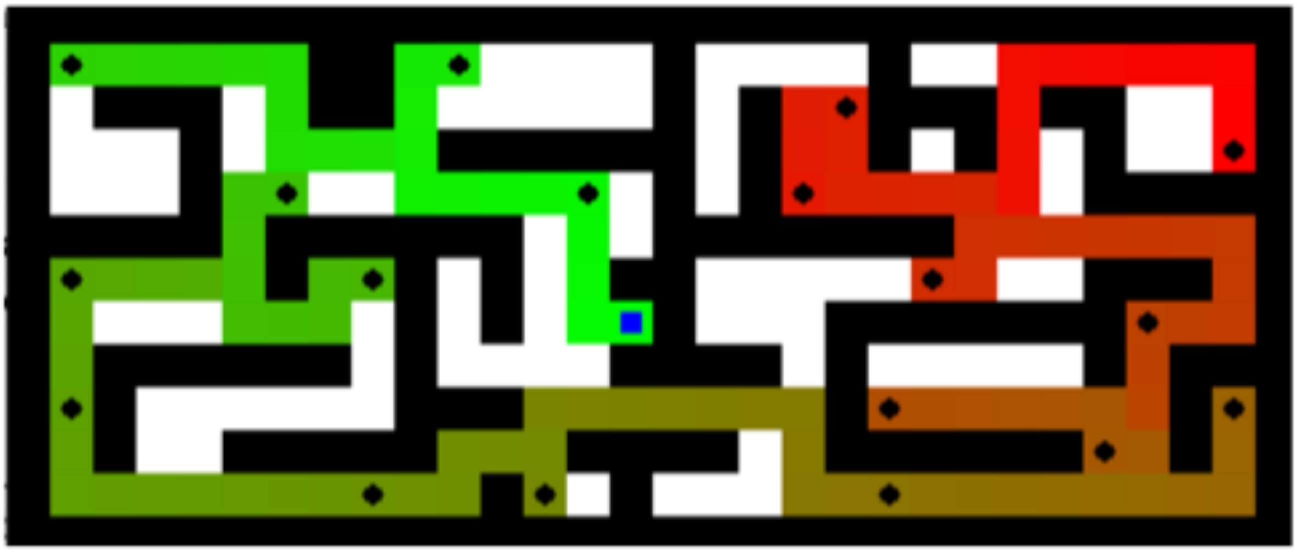


Results

Path Length: 38

States Explored: 294

Screenshot 5.2 (A*, small Search)

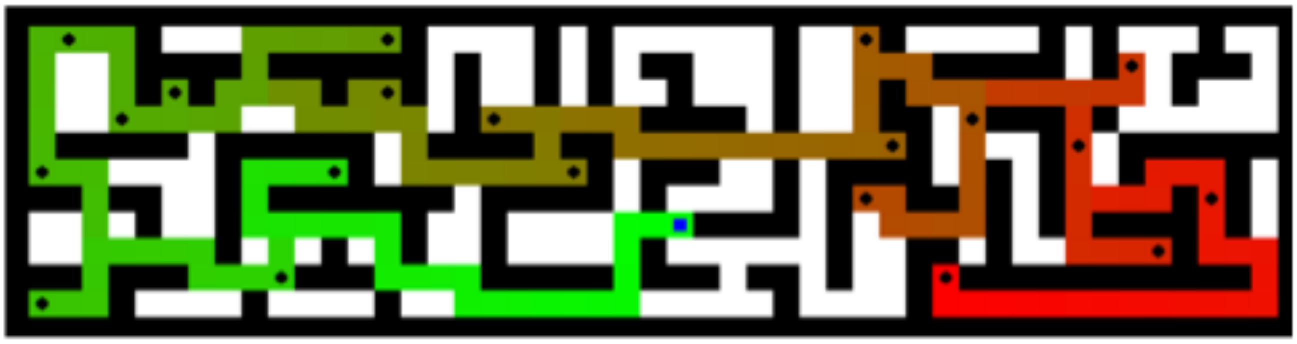


Results

Path Length: 149

States Explored: 14481

Screenshot 5.1 (A*, medium Search)



Results

Path Length: 209

States Explored: 64061

Extra credit:

When we first saw the graph of bigDots, we realized that there are so many dots we need to eat before getting to the end. And our current Astar would spend a lot of time on it, since we need to build a dictionary with the distance between every two dots, and find the MST with these distances. And the time of this part would increase with n square with the increasing number of n. Calculating Manhattan distance would save us much more time than calculating the Astar path. And also, since the dots are pretty close to each other, the MST decided by the Manhattan distance would be pretty close to the MST decided by the Astar path. This method does not apply to any maze, but for maze like this, I would lower a little accuracy on finding the shortest path, but saving much on the time cost.

Statement of Contribution

All three group members think and come up with a mental blue print before the actual implementation of algorithms. Cooperatively, all three of us familiarized each other with the main concepts from the lecture. We communicated extensively and designed the structure, pseudo-code of how we are going to design the bfs, dfs, greedy, and A* search algorithms.

All of us did pair programming on the big screen with **Jinhao** playing the major rule in the actual implementation and thus contributing the bulk. **Ruixi** managed to gather relevant explanations for many concepts and utilized her net-working resources to explain to us certain major muddy points, such as the concept of revisiting the previous states, etc.. Ruixi also contributed partially in part 1 of the actual implementation. **Chen** contributed partially in part 2 and explained the idea of explore set. Chen also wrote the report.