ECE 448
FA 2020
Prof. Hongwei Wang

Assignment 1 Report
Search Algorithms

Name: Yucheng Jin

ID: 3170112253

Name: Yiqing Xie

ID: 3170111404

Name: Hangtao Jin

ID: 3170112247

Date: Sept. 30, 2020

Table of Contents

1 Section I

1.1 BFS

BFS means the breadth-first search algorithm. It uses a queue structure to realize its function. Queue is a first-in-first-out data structure, its *enqueue* function inserts a new item at the top of the queue, and its *dequeue* function removes the item at the top of the queue. BFS is a uniform cost search algorithm that guarantees optimal in the given scenario, because it expends uniformly until it detects the object. Beginning with the start point, BFS will search the neighbors of each point. When a new point is reached, BFS will mark it as "visited", and the next time when the point is reached again, BFS simply continues. We use a list to resemble the behavior of the queue, and another list to trace the points visited to realize BFS.

1. State: means the characteristics of a point (e.g. whether it is an objective point).

2. Node: means the coordinates of a point (e.g. (x, y)).

3. Frontier: the set of neighbor points of the current point.

4. We trace the explored states by a list in python.

5. We ignore the repeated states.

1.2 DFS

DFS means the depth-first search algorithm. It uses a stack structure to realize its function. Stack is a first-in-last-out data structure, its *push* function inserts a new item at the top of the stack, and its *pull* function removes the item at the bottom of the stack. Beginning with the start point, BFS will search the neighbors of each point, and let the first neighbor point detected but not visited as the new start point. The same procedure will be repeated until the objective point is found.

1. State: means the characteristics of a point (e.g. whether it is an objective point).

2. Node: means the coordinates of a point (e.g. (x, y)).

3. Frontier: the set of neighbor points of the current point.

4. We trace the explored states by a list in python.

5. We ignore the repeated states.

1.3 Greedy

Greedy best first search means that for each step, we want to get as closer as possible to the goal. When moving, we have four neighbors: up, down, left, right. We choose the position that is nearest to the goal.

During each step, our current state includes the current coordinates and current Manhattan distance to the goal. Our node is the same as a state. Our frontier is all neighbors of the current point. When we want to take an item from the frontier, we will take the one with the least Manhattan distance to the goal. Each time we add a new point into our queue, we will calculate its Manhattan distance to the goal and insert it into the queue according this value. We don't maintain an explored states list. However, we do store where the visited node comes from, which means the node who puts the visited node into the queue as a neighbor. When we go to the visited node next time, we will find it already has a "father", then we can simply skip it.

1.4 A*

A* search arranges nodes waiting to be examined by f(n) = g(n) + h(n), which means the Manhattan distance to the goal plus the path length it already has gone through. Our current state includes current position, passed path length, and Manhattan distance to the goal. Our node is the same as state. Our frontier is all possible next points to be explored. We will take the one with least f(n) to be our next node. Just like the above Greedy solution, we do not maintain an explored state list, but only a list of "father" modes. When we meet a new point with father, we will see whether the new path has a smaller f(n). if so, we will update the point.

2 Section II

For greedy BFS search, the heuristics is just the Manhattan distance to the goal. When we select the next state from frontier, we will choose the one with least heuristics.

For A* search, the heuristics for single dot and multiple-dot situations are kind of different (though they use the same code). The code of A* search has two parts. The first part finds the path between the two assigned points. The single dot situation is solved by sorely using this part. In this part, the heuristics is the passed path length plus the Manhattan distance to the goal. To show that this is admissible, we can see that for all points, the remaining path length must be longer or equal to the Manhattan distance to the goal. As the real path that is equal to the passed path length plus the remaining pass length, it must be longer or be equal to the passed path length plus the Manhattan distance to the goal. So the heuristics is admissible for single dot situation.

For the multiple-dot situation, the calculation is a lot more complex. We first calculate all path between any two points in the objectives. That means we will get a complete graph. Our second part of A* search is to find the shortest path that connects all vertices in the graph. We start from the start point and keep going to the next vertex until all vertices are connected. To choose the next vertex, we have our heuristics equal to the passed path length plus the edge between the current vertex and the next vertex plus the length of the MST (minimum spanning tree) of all remaining vertices (including the next vertex). We prove this heuristic is admissible below: the passed path length plus the edge between the current vertex and the next vertex is the path length between the start point and the next vertex. The length of the MST is always smaller than the remaining path length between the next vertex and the end vertex (though we don't know which vertex is the end vertex yet). That means the sum of the values mentioned above, which is our heuristics, is always smaller than the real path, which equals to the path length between the start point and the next vertex plus the remaining path.

3 Section III

3.1 BFS

3.1.1 mediumMaze.txt

Path Length: 111

States Explored: 661

Figure 3.1.1. BFS Solution for mediumMaze.txt

3.1.2 bigMaze.txt

Path Length: 183

States Explored: 1422

Figure 3.1.2. BFS Solution for bigMaze.txt

3.1.3 openMaze.txt

Path Length: 52

States Explored: 1019

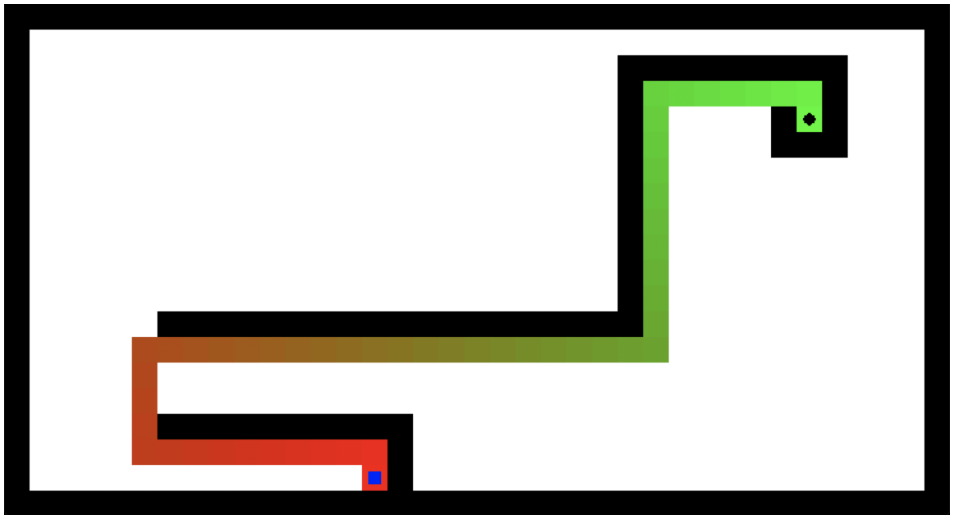

Figure 3.1.3. BFS Solution for openMaze.txt

3.2 DFS

3.2.1 mediumMaze.txt

Path Length: 115

States Explored: 647

Figure 3.2.1. DFS Solution for mediumMaze.txt

3.2.2 bigMaze.txt

Path Length: 183

States Explored: 1287

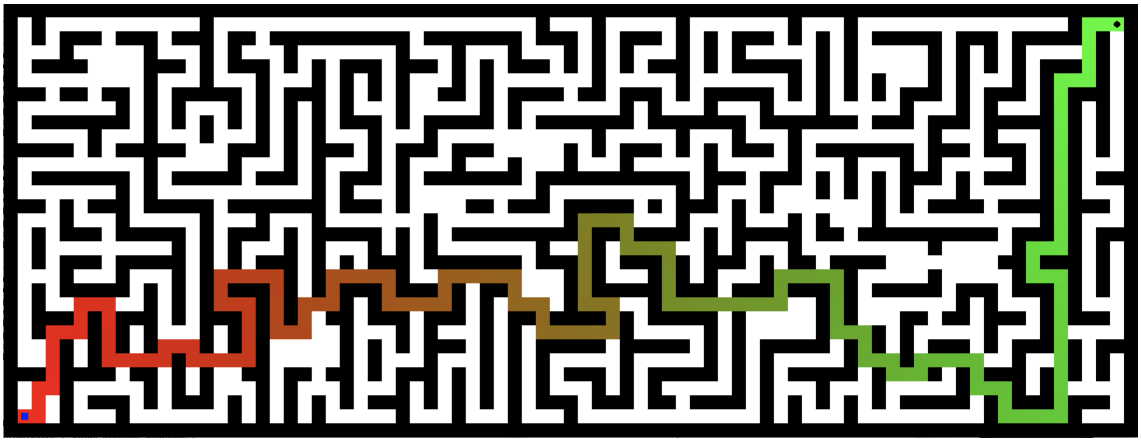

Figure 3.2.2. DFS Solution for bigMaze.txt

3.2.3 openMaze.txt

Path Length: 52

States Explored: 562

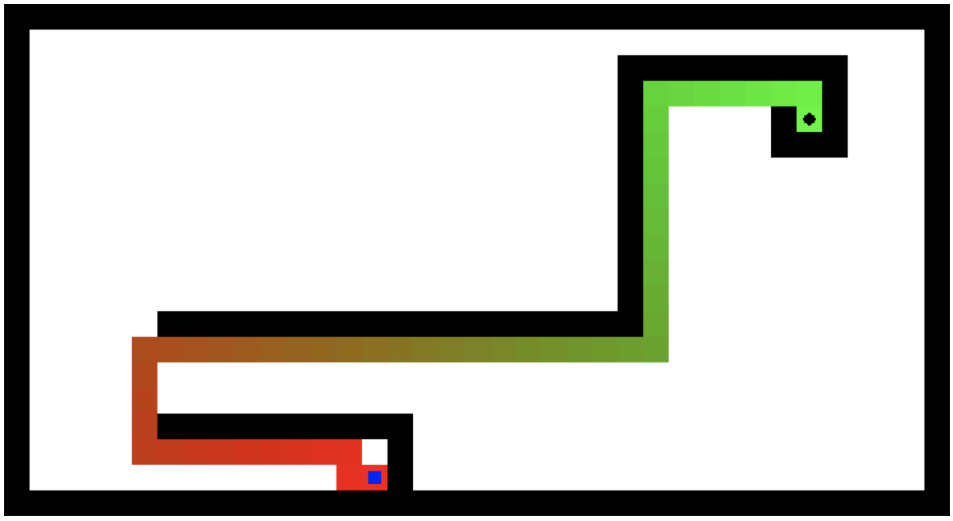

Figure 3.2.3. DFS Solution for openMaze.txt

3.3 Greedy

3.3.1 mediumMaze.txt

Path Length: 147

States Explored: 343

Figure 3.3.1. Greedy Solution for mediumMaze.txt

3.3.2 bigMaze.txt

Path Length: 277

States Explored: 468

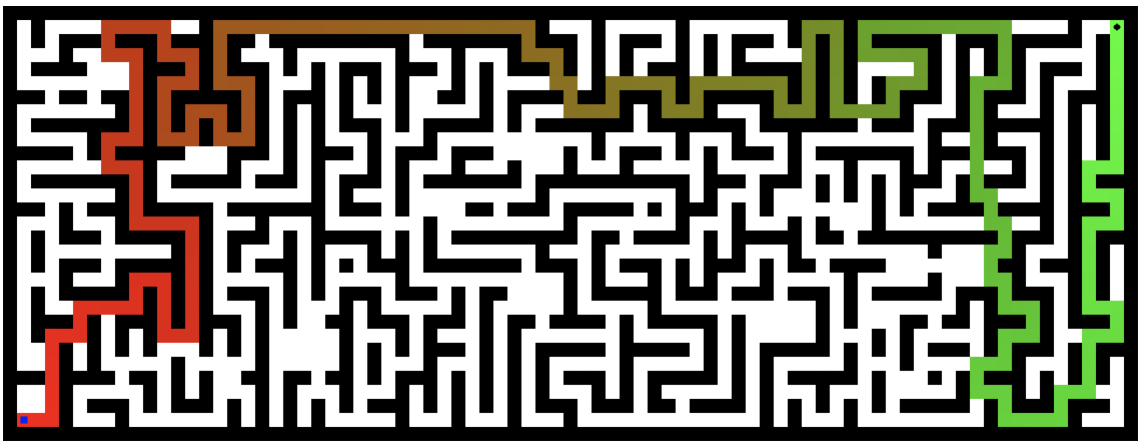

Figure 3.3.2. Greedy Solution for bigMaze.txt

3.3.3 openMaze.txt

Path Length: 56

States Explored: 74

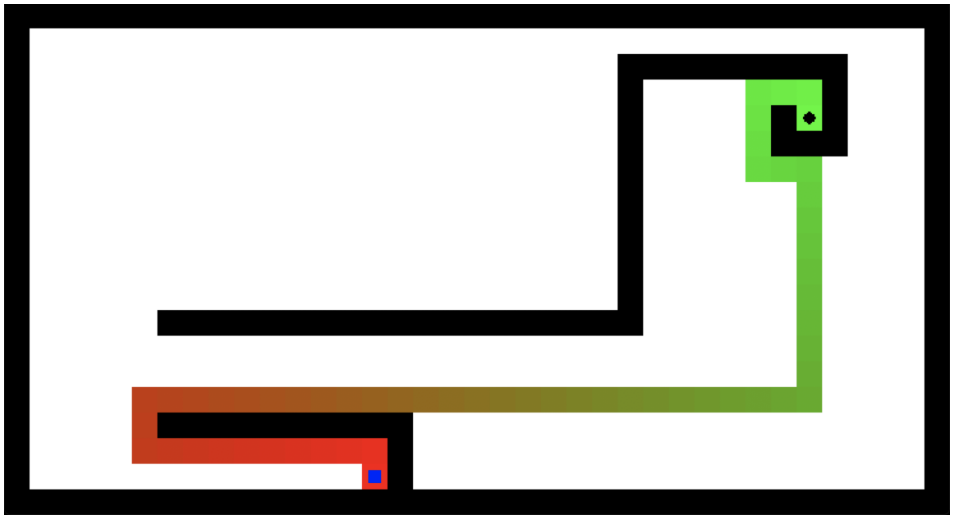

Figure 3.3.3. Greedy Solution for openMaze.txt

3.4 A*

3.4.1 mediumMaze.txt

Path Length: 111

States Explored: 363

Figure 3.4.1. A* Solution for mediumMaze.txt

3.4.2 bigMaze.txt

Path Length: 183

States Explored: 1301

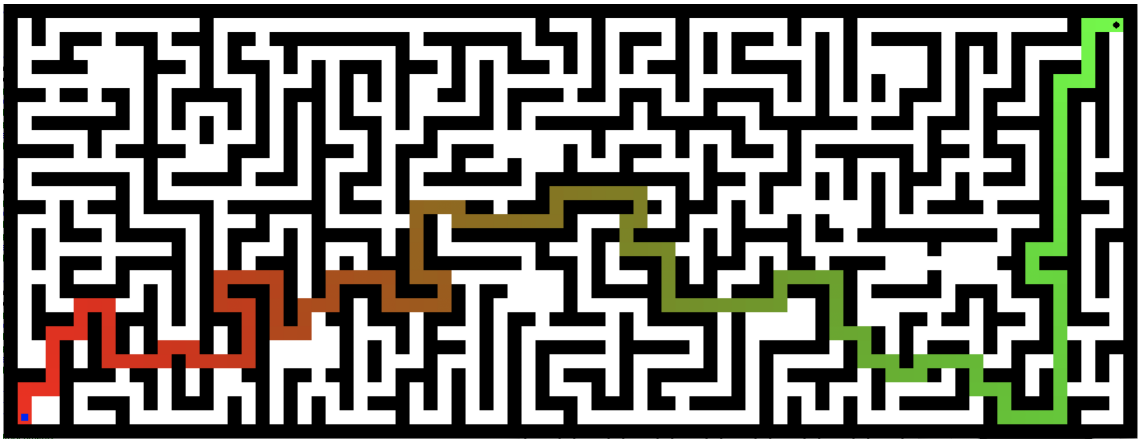

Figure 3.4.2. A* Solution for bigMaze.txt

3.4.3 openMaze.txt

Path Length: 52

States Explored: 384

Figure 3.4.3. A* Solution for openMaze.txt

4 Section IV

4.1 tinySearch.txt

Path Length: 38

States Explored: 830

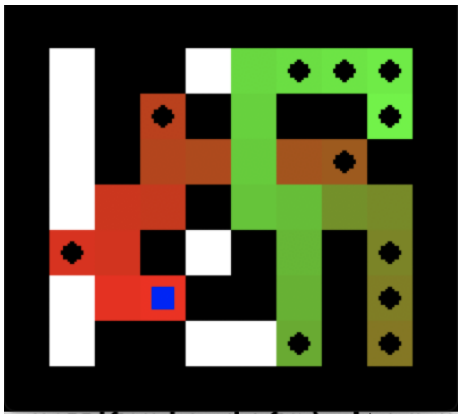

Figure 4.1. A* Solution for tinySearch.txt

4.2 smallSearch.txt

Path Length: 149

States Explored: 9261

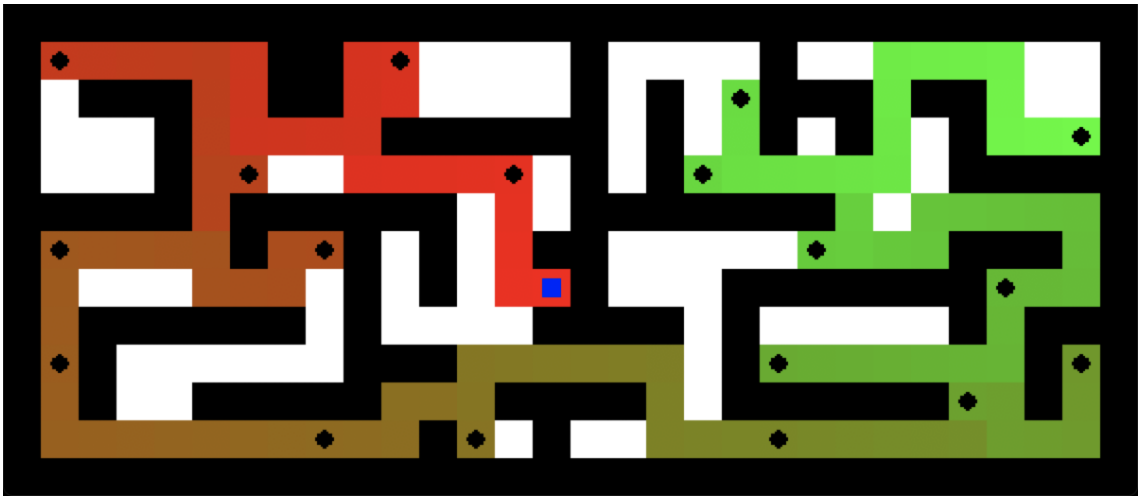

Figure 4.2. A* Solution for smallSearch.txt

4.3 mediumSearch.txt

Path Length: 209

States Explored: 24051

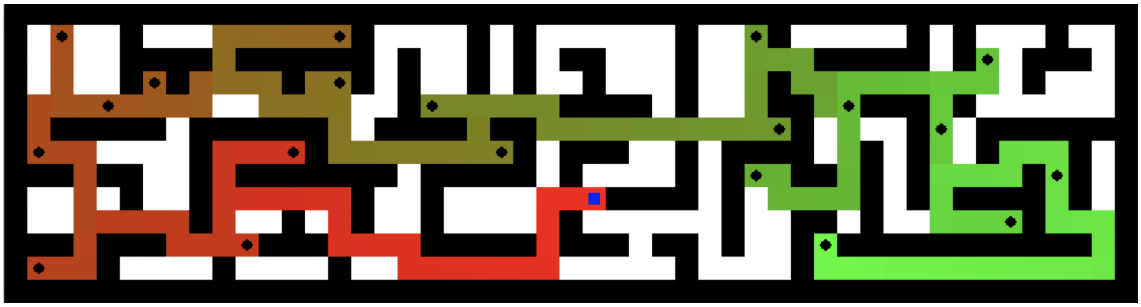

Figure 4.3. A* Solution for mediumSearch.txt

5 Statement of Contribution

Jin Yucheng implemented BFS and DFS, Jin Hangtao also implemented BFS and DFS. Xie Yiqing implemented Greedy BFS and A*. The report is written by Jin Yucheng and Xie Yiqing.