**MP1 Report**

**Language C++**

**General settings**

To get well prepared for implementation, we create a class “coord” which saves a dot’s (x, y) position and declares two overloading operators “==” and “<”. Then we set up some global variables which bring us great convenience. They are,

**coord** **start**, **coord[] endpoints** for start and ending point

**vector<coord> endset** for multiple end points in Part 1.2

**int x\_max** and **y\_max** for maze’s size

**int** **dot\_counter** for number of dots

**int nvisit** for nodes visited and **pcost** for path cost.

main.cpp contains functions main, printMaze and draw\_path. main() reads in the maze file and gets the maze’s size in the meantime. After the initial work, it calls printMaze to perform any searches; printMaze will print the value of maze’s size and the maze itself at first. Then it finds out the starting point P and the ending point (ending **points** for Part 1.2), and print them out. After the printing work, it will require user to choose which algorithm, i.e. BFS, DFS, Greedy or A\* search to perform. We will call the corresponding function which is placed in algorithm.cpp to perform the search and the function will figure out the path cost, number of nodes that have been visited and return a path solution. Finally, we call draw\_path with that solution as a parameter to print out the path, path cost and number of nodes visited.

algorithm.cpp contains our implementation for Manhattan distance, overloading operators, goal test and a getter function to get coordinate offset. More notably, it contains implementation of 3 algorithms, BFS, DFS and Greedy search to find the path.

a\_star.cpp

**Algorithms**

* For all

‘X’ stands for invalid nodes we have explored.

‘.’ stands for valid nodes included in a path

‘P’ stands for ending point

* BFS

Use a queue to represent the frontier. Construct a 1-d array called “parent” to keep track of the proceeding path. To begin with, we push the start point onto the frontier and mark it the root node as (-1, -1) in “parent” array.

Call a recursive helper function, BFS\_R to explore the neighbors(up, down, left and right), record all nodes we have explored, get total number of them as well, find out and save the path. If all neighbors are invalid, go to call BFS\_R itself recursively. As for base case, if it reaches the end point successfully, returns 1.

With success and the path got in BFS\_R, we use a while loop to write the path out in ‘.’ into the maze. Meanwhile, we increment “pcost” and finally get the total path cost when the loop finishes. Before BFS finishes, we safely clear the dynamic memory we allocated for the “parent” array and return success.

* DFS

We use a stack to represent the frontier and construct a 1-d array called “parent” to keep track of the proceeding path. To begin with, we push the start point onto the frontier and mark it the root node as (-1, -1) in “parent” array.

Call a recursive helper function, DFS\_R to explore the neighbors(up, down, left and right), record all nodes we have explored, get total number of them as well, find out and save the path. If all neighbors are invalid, go to call DFS\_R itself recursively. As for base case, if it reaches the end point successfully, returns 1.

With success and the path got in DFS\_R, we use a while loop to write the path out in ‘.’ into the maze. Meanwhile, we increment “pcost” and finally get the total path cost when the loop finishes. Before DFS finishes, we safely clear the dynamic memory we allocated for the “parent” array and return success.

* Greedy
* A\*

**Part 1.1**

1. medium maze

* BFS

maze with computed path

solution cost

number of expanded nodes

* DFS

maze with computed path

solution cost

number of expanded nodes

* Greedy

maze with computed path

solution cost

number of expanded nodes

2. big maze

* BFS

maze with computed path

solution cost

number of expanded nodes

* DFS

maze with computed path

solution cost

number of expanded nodes

* Greedy

maze with computed path

solution cost

number of expanded nodes

3. open maze

* BFS

maze with computed path

solution cost

number of expanded nodes

* DFS

maze with computed path

solution cost

number of expanded nodes

* Greedy

maze with computed path

solution cost

number of expanded nodes

**Part 1.2: A\***

1. tiny maze

maze with computed path

solution cost

number of expanded nodes

2. small maze

maze with computed path

solution cost

number of expanded nodes

3. medium maze

maze with computed path

solution cost

number of expanded nodes

**Heuristic for A\***

**Individual contribution**

Jishuai Zhang:

Chengyun Wu:

Hanyu Wang: Work