ECE 448/CS 440

Spring 2015

Assignment 1: Maze Search, Part 1

Harrison Kiang (3 credit), Jonathan Park (3 credit), Gabriella Quirini (3 credit)

NetIDs: hkiang2, jgpark2, gquirin2

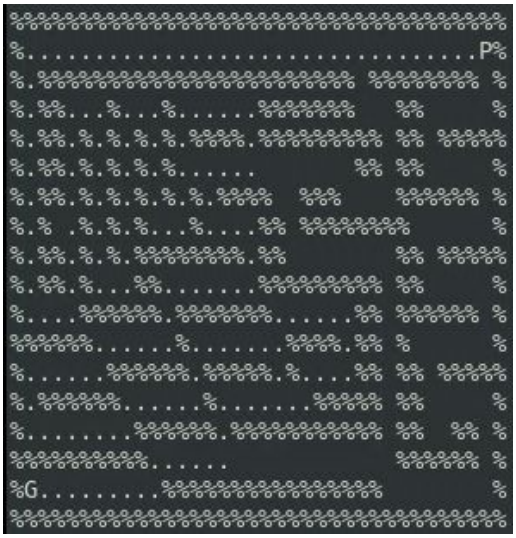
1.1 Basic Pathfinding

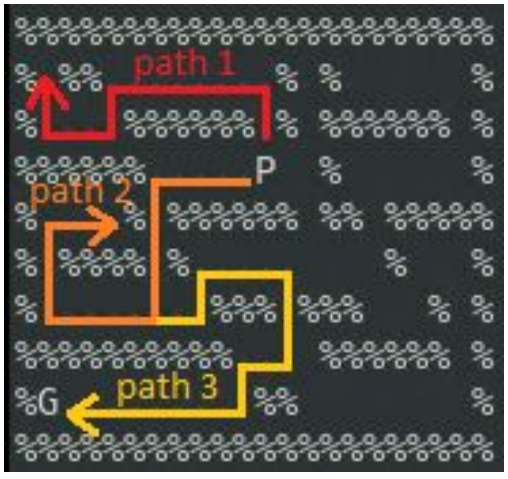
We begin with the problem of finding a path through a maze. Given a starting point, our search algorithm must traverse the maze to reach the goal state. The maze is made up of 1x1 cells, with one starting cell and one goal cell. The rest of the maze is defined by the alignment of open cells (with a step cost of 1) and wall cells (which our “pacman” cannot enter). Based on this structure, we implemented several search algorithms for solving mazes of different sizes and difficulties.

Depth-First Search

DFS works by expanding the deepest unexpanded node, and our approach does just that. We start by initializing an empty stack and pushing our sole starting cell into it. The stack is for the frontier--at each iteration we pop from the top of the stack (and mark the cell visited), use that as our current cell to explore all its neighbors. The neighboring nodes get pushed to the frontier stack if they are unvisited and not a wall. The ordering of the push is arbitrary, and in our case, we check the neighbors in the following order: right, down, left, up (so up will be at the top of the stack if valid). We repeat this process until the frontier is empty (which means we exhausted the search and we could not reach a goal), or we visit the goal node. When a dead end is reached in the maze, the top of the stack holds the neighbor of the last place we branched off in an arbitrary direction, so that we may now begin expanding the next deepest node. This method is illustrated below.

We begin by expanding upwards from our start, P. Once this path reaches a dead end, the next deepest node is all the way up the tree under the root (to the left of P). This is then explored in the same fashion. Path 3 branches off from path 2 because that point is deeper into the maze than our other option - the unexpanded node to the right of P.

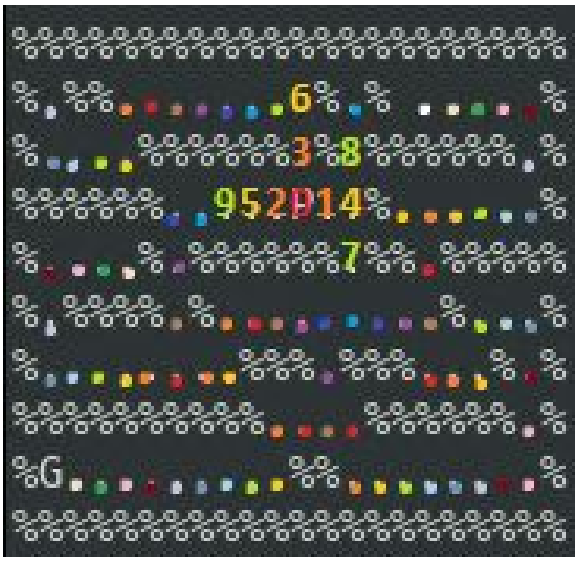


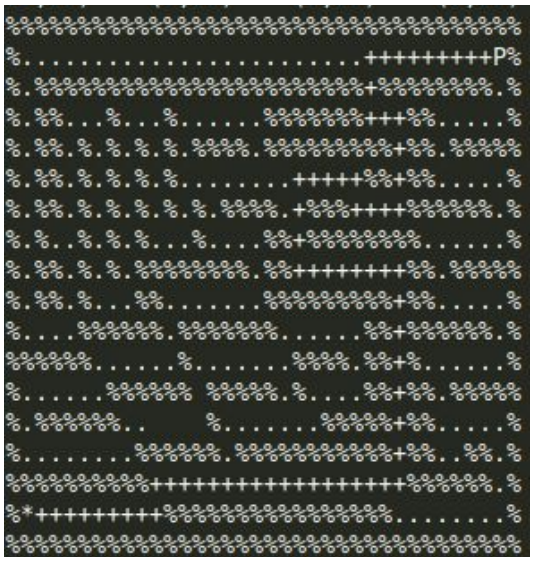


Breadth-First Search

BFS works by expanding the shallowest unexpanded node. For this we use a FIFO queue, where for each node we visit, we push it's neighbors. Because of this, for each iteration we expand outward in all directions from the start, rather than follow a single path like DFS. The order in which we expand each level is arbitrary, and when we reach a dead end, we just continue expanding the other shallowest branches. The problem with BFS is that if there are many available paths in the maze, whether they lead to the goal or not, the algorithm will expand many more nodes than necessary. This method is illustrated below, with the numbers being the order in which the first 10 nodes were expanded, and the colors corresponding to the depth of the node.

We begin by expanding the nodes nearest to the start, then outwards from there sequentially. There is high time and memory complexity, as there are many paths to explore, as each node from each level must be expanded before progressing down the correct path. Because of this, BFS finds an optimal path, but at the cost of expanding many nodes. In the following diagrams, the dots represent nodes that have been visited or are on the frontier, and the "+" represents the path taken.

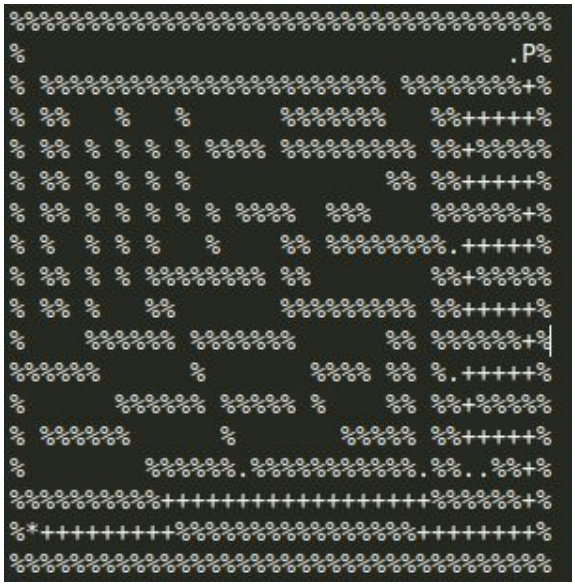


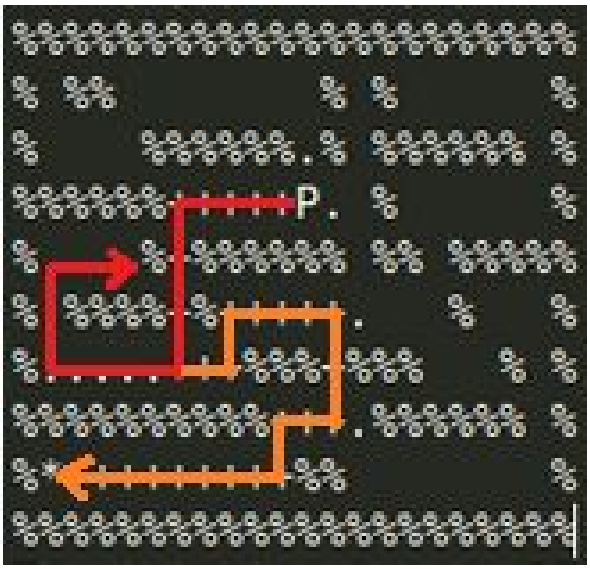


Greedy Best-First Search

Greedy search is an informed search algorithm, meaning that rather than blindly traversing the maze, it takes into account its position relative to the goal. A greedy search expands the node that has the lowest value of the heuristic function. Here, our heuristic function is the manhattan distance (distance from goal ignoring walls). For this we use a priority queue. At each visited node, the algorithm pushes its neighbors into the queue. The queue then prioritizes its entries based on heuristic value. We then pop

the node with the lowest manhattan distance as our next node to visit. If we reach a dead end, we begin expanding the next lowest manhattan value in the priority queue.

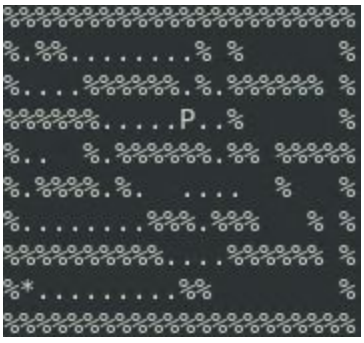




In the small maze above, we begin by moving left toward the goal. The heuristic value of the node at the end of the red path is 7, but since it is at a dead end it then follows the orange path. The first node of the orange path has a heuristic value of 8, and the heuristic increases before looping back as it progresses.

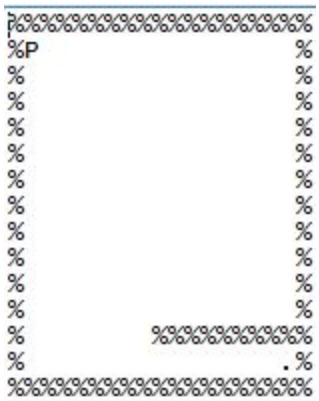
A✳ Search

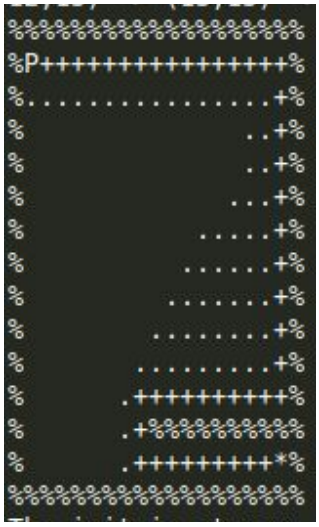
This algorithm is also an informed search. It takes the greedy search one step further by also taking into account the current path cost. The goal of this approach is to find a more efficient path than greedy. In order to implement this, we used an evaluation function to estimate the *total* cost of the path: $f(n) = g(n) + h(n)$ where g(n) is the cost so far to reach n (path cost) and h(n) is the heuristic value of node n. We again use the manhattan distance for our heuristic, as well as a priority queue.

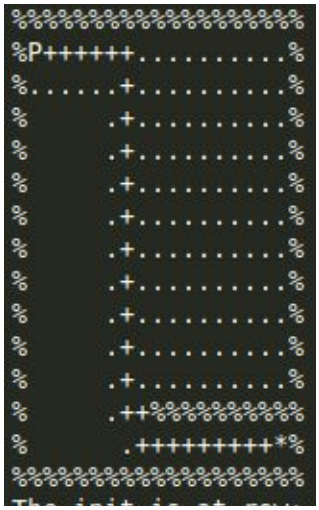


1.2 Designing “Difficult” Mazes

In this portion of the assignment we designed two mazes, one to be easier to solve with greedy best first search rather than A* and vice-versa. The difficulty is evaluated by the number of nodes expanded (or the time taken to find a path). The mazes are as follows below, starting with the maze easier for greedy:

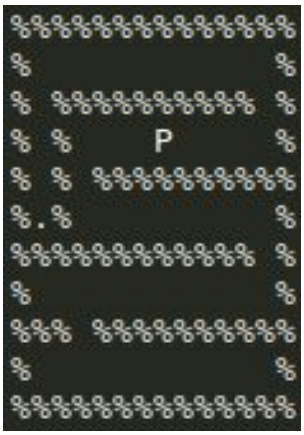


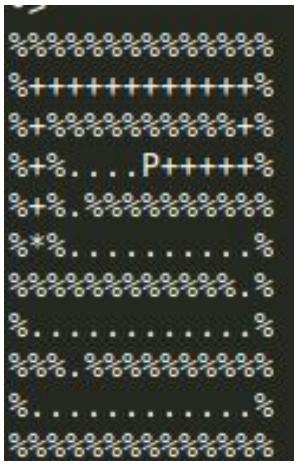


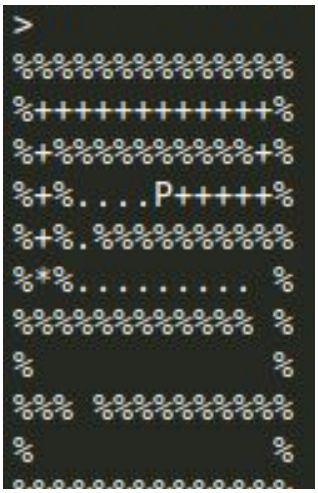


the greedy easy maze **greedy search** **A* search**

The above left-most maze, called “greedyEasyMaze.txt”, is basically an open square (image not to scale) maze with the start at the upper-left and the goal in the lower-right. For tie breakers, because our algorithms look at the cell to its right first over the cell under it, both algorithms will travel all the way to the right, then try to reach the node by going straight down. This is where they run into the wall. As shown in the middle, greedy best-first search will travel from the bottom-right-most corner (before the wall) to either left or up in order to find a path to go around the obstruction. This is because it only considers the lowest distances of a node to the goal. A* on the other hand will search not only from the bottom-right-most corner, but also regressing back the path it came from and spreading out in an attempt to prevent the obstruction to begin with and correct the path it took since the beginning. This is because the path’s node’s preferences are not only determined by how close they are to the goal, but also the path cost required to reach the current location. Although A*’s method allows the much more optimal path, it unfortunately ends up exploring more nodes than greedy does. This can be seen on the right-most picture above.







the A* easy maze **greedy search** **A* search**

This above left-most maze is called “astarEasyMaze.txt”. Both algorithms are enticed to take the left path from the start as the path leads closer to the goal via manhattan distance. Because greedy only looks at the distance from a node to the goal, it will deter away from the top-right-most cells as much as possible. As such, greedy algorithm will continue to explore down the enclosed left path, and only until much later, will it go back and try to explore the top right direction. This can be seen in the middle picture. Because A* keeps track of the current path cost as well as the distances from a node to the goal, the winding incorrect left path will become undesirable much earlier than for the greedy algorithm. This causes A* to check the top right direction much earlier in the run, saving a lot of node expansions.

1.3 - Modified A*

Here we are tasked with finding an efficient path through multiple goals throughout a given maze (either smallSearch.txt, trickySearch.txt, mediumSearch.txt, or bigSearch.txt). In our modified A* search algorithm (astar_3 in main.cpp), we account for all of these goals in a vector called maze_props. The Manhattan distances for each cell in the maze are evaluated according to the nearest goal. Like the A* algorithm described above, we expand the node with the smallest total cost comprising of the sum of the Manhattan distances and the step cost.

There is an open list and a closed list. These lists allow for backtracking. The open list contains cells to explore, while the closed list contains processed cells. Each time a goal is “eaten”, the Manhattan distances, and therefore also the total distances, are re-evaluated. For each expanded node in the open list, its neighbors are stored in a local vector and popped on to the open list if they are not yet processed (not in the closed list) and if the path through the neighbor is evaluated to be the shortest path to a certain goal. The algorithm runs until the open list is empty. At this point, if a solution is found, it will be printed.

Below is an image depicting the running of the modified A* search algorithm on smallSearch.txt. The order of the goals explored are printed in a list below the maze itself. The other mazes’ order of goals explored are printed in tables below.

```
%%%%%%%%%%%%%%%%%%%%
%*......     ***P.*%
%*%%*%%*%%*%%*%% %*%
% %%.%*****...   %*%
%%%%%%%%%%%%%%%%%%%%
Nodes expanded: 29

Goal | coordinates
1   (15, 1)
2   (14, 1)
3   (13, 1)
4   (13, 2)
5   (18, 1)
6   (18, 2)
7   (18, 3)
8   (10, 3)
9   (9, 3)
10  (8, 3)
11  (7, 3)
12  (6, 3)
13  (10, 2)
14  (7, 2)
15  (4, 2)
16  (1, 1)
17  (1, 2)
The init is at row: 16 and col: 1
The goal is at row: 18 and col: 3
```

<- smallSearch.txt results
medium result ->

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%************%%%%%G***********%
%%%*%***%%%*********%*%**G%*%%%
%***%%%*%*%%%%*%*%%%%%%*%%%***%
%*%G****%******%******%*****%*%
%*%%%*%%%%%*%%%%%%%*%%%*%*%%%%%
%*****%*********P****%***%*****%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Nodes expanded: 106

Goal | coordinates
1   (25, 2)
2   (3, 4)
3   (18, 1)
4   (14, 6)
5   (13, 6)
6   (12, 6)
7   (11, 6)
8   (10, 6)
9   (9, 6)
10  (8, 6)
11  (7, 6)
12  (11, 5)
13  (11, 4)
14  (10, 4)
15  (9, 4)
16  (9, 3)
17  (16, 6)
18  (17, 6)
19  (18, 6)
20  (19, 6)
21  (19, 5)
22  (19, 4)
23  (18, 4)
24  (17, 4)
25  (16, 4)
26  (16, 3)
27  (16, 2)
28  (15, 2)
29  (14, 2)
30  (13, 2)
31  (12, 2)
32  (11, 2)
33  (11, 1)
34  (10, 1)
35  (9, 1)
36  (8, 1)
37  (7, 1)
38  (6, 1)
39  (5, 1)
40  (4, 1)
41  (3, 1)
42  (2, 1)
43  (1, 1)
44  (3, 2)
45  (3, 3)
46  (2, 3)
```

```
46  (2, 3)
47  (1, 3)
48  (1, 4)
49  (1, 5)
50  (1, 6)
51  (2, 6)
52  (3, 6)
53  (4, 6)
54  (5, 6)
55  (5, 5)
56  (5, 4)
57  (4, 4)
58  (5, 2)
59  (6, 4)
60  (7, 4)
61  (7, 3)
62  (14, 3)
63  (14, 4)
64  (13, 4)
65  (20, 4)
66  (21, 4)
67  (6, 2)
68  (7, 2)
69  (12, 4)
70  (17, 2)
71  (18, 2)
72  (19, 2)
73  (19, 1)
74  (20, 1)
75  (21, 1)
76  (22, 1)
77  (23, 1)
78  (24, 1)
79  (25, 1)
80  (26, 1)
81  (27, 1)
82  (28, 1)
83  (29, 1)
84  (27, 2)
85  (27, 3)
86  (28, 3)
87  (29, 3)
88  (29, 4)
89  (24, 2)
90  (27, 4)
91  (26, 4)
92  (25, 4)
93  (24, 4)
94  (23, 4)
95  (23, 3)
96  (21, 2)
97  (23, 5)
98  (23, 6)
99  (22, 6)
100 (21, 6)
101 (25, 5)
102 (25, 6)
103 (26, 6)
```

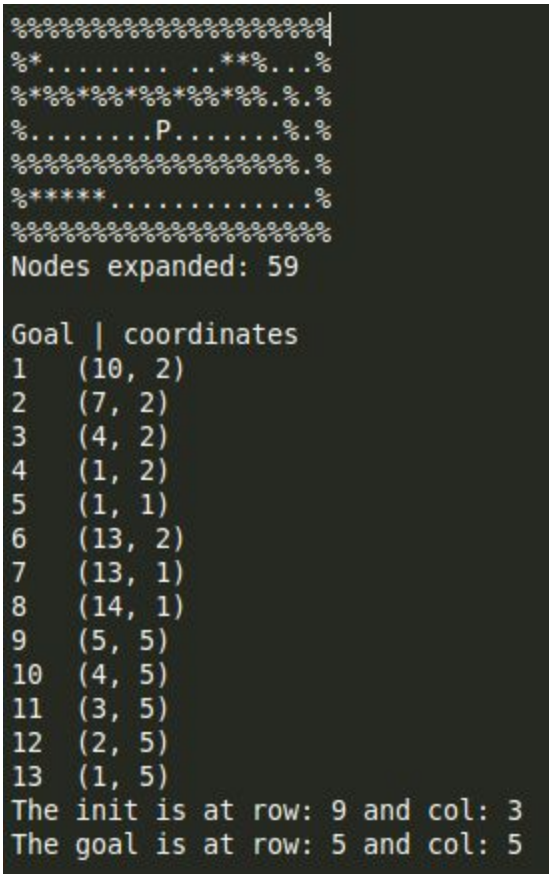
```
100 (21, 6)
101 (25, 5)
102 (25, 6)
103 (26, 6)
104 (27, 6)
105 (28, 6)
106 (29, 6)
107 (12, 1)
108 (23, 2)
The init is at row: 15 and col: 6
The goal is at row: 29 and col: 6
```

bigSearch results

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%*****%*****************%*****%
%*%%%*%*%%%*%%%%%%%*%%%*%*****%
%*%***%*%******%******%*%G**G*%
%***%%%*%*%%%%%*%*%%%%***%%***%
%%%*%*%*%*******%***%*%***%*%%%
%***%*%%%*%*%%%.%%%*%G%%%*%***%
%*%%%*******%.....%*******%%%*%
%***%*%%%%%*%%%%%%%*%G%%%*%***%
%%%*%***%*%*****%*****%*%***%%%
%***%%%*%*%%%%%*%*%%%%*%%%%***%
%******%*******%*******%****%*%
%*****%*%%%G%%%%%%%*%%%*%G%%%*%
%*****%********P****%***%*****%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Nodes expanded: 227

Goal | coordinates
1   (21, 8)
2   (28, 3)
3   (11, 12)
4   (25, 12)
5   (21, 6)
6   (25, 3)
7   (14, 13)
8   (13, 13)
9   (12, 13)
10  (11, 13)
11  (10, 13)
12  (9, 13)
13  (8, 13)
14  (7, 13)
15  (7, 12)
16  (7, 11)
17  (6, 11)
18  (5, 11)
19  (4, 11)
20  (3, 11)
21  (2, 11)
22  (1, 11)
23  (1, 10)
24  (2, 12)
25  (2, 13)
26  (1, 13)
27  (3, 10)
28  (3, 9)
29  (3, 8)
30  (2, 8)
31  (1, 8)
32  (1, 7)
33  (1, 6)
34  (2, 6)
35  (3, 6)
36  (3, 5)
37  (3, 4)
38  (2, 4)
39  (1, 4)
40  (1, 3)
41  (1, 2)
42  (1, 1)
43  (2, 1)
44  (3, 1)
45  (4, 1)
46  (5, 1)
47  (5, 2)
48  (5, 3)
49  (4, 3)
50  (3, 13)
51  (4, 13)
52  (5, 13)
53  (7, 10)
54  (7, 9)
55  (6, 9)
56  (5, 9)
57  (5, 8)
58  (5, 7)
59  (6, 7)
60  (7, 7)
61  (8, 7)
62  (9, 7)
63  (10, 7)
64  (11, 7)
65  (11, 6)
66  (11, 5)
67  (12, 5)
68  (13, 5)
69  (14, 5)
70  (15, 5)
71  (16, 5)
72  (16, 4)
73  (16, 3)
74  (17, 3)
75  (18, 3)
76  (19, 3)
77  (20, 3)
78  (21, 3)
79  (21, 4)
80  (22, 4)
81  (23, 4)
82  (23, 3)
83  (23, 2)
84  (23, 1)
85  (22, 1)
86  (21, 1)
87  (20, 1)
88  (19, 1)
89  (18, 1)
90  (17, 1)
91  (16, 1)
92  (15, 1)
93  (14, 1)
94  (13, 1)
95  (12, 1)
96  (11, 1)
97  (10, 1)
98  (9, 1)
99  (8, 1)
100 (7, 1)
101 (7, 2)
102 (7, 3)
103 (7, 4)
104 (7, 5)
105 (11, 2)
106 (11, 3)
107 (10, 3)
108 (9, 3)
109 (9, 4)
110 (9, 5)
111 (12, 3)
112 (13, 3)
113 (14, 3)
114 (23, 5)
115 (24, 5)
116 (25, 5)
117 (25, 6)
118 (25, 7)
119 (24, 7)
120 (23, 7)
121 (22, 7)
122 (21, 7)
123 (20, 7)
124 (19, 7)
125 (19, 6)
126 (19, 5)
127 (18, 5)
128 (21, 5)
129 (19, 8)
130 (19, 9)
131 (18, 9)
132 (17, 9)
133 (16, 9)
134 (16, 10)
135 (16, 11)
136 (17, 11)
137 (18, 11)
138 (19, 11)
139 (20, 11)
140 (21, 11)
141 (21, 10)
142 (21, 9)
143 (19, 12)
144 (19, 13)
145 (18, 13)
146 (17, 13)
147 (25, 8)
148 (25, 9)
149 (24, 9)
150 (23, 9)
151 (23, 10)
152 (23, 11)
153 (24, 11)
154 (25, 11)
155 (26, 11)
156 (27, 11)
157 (27, 10)
158 (28, 10)
159 (29, 10)
160 (29, 11)
161 (29, 12)
162 (29, 13)
163 (28, 13)
164 (27, 13)
165 (26, 13)
166 (25, 13)
167 (23, 12)
168 (23, 13)
169 (22, 13)
170 (21, 13)
171 (11, 8)
172 (11, 9)
173 (12, 9)
174 (13, 9)
175 (14, 9)
176 (14, 10)
177 (14, 11)
178 (13, 11)
179 (12, 11)
180 (11, 11)
181 (10, 11)
182 (9, 11)
183 (9, 10)
184 (9, 9)
185 (5, 6)
186 (5, 5)
187 (3, 12)
188 (27, 9)
189 (27, 8)
190 (28, 8)
191 (29, 8)
192 (29, 7)
193 (29, 6)
194 (28, 6)
195 (27, 6)
196 (27, 5)
197 (27, 4)
198 (28, 4)
199 (29, 4)
200 (29, 3)
201 (29, 2)
202 (28, 2)
203 (27, 2)
204 (26, 2)
205 (25, 2)
206 (25, 1)
207 (26, 3)
208 (27, 1)
209 (26, 1)
210 (29, 1)
211 (28, 1)
212 (27, 3)
213 (1, 12)
214 (4, 12)
215 (5, 12)
216 (2, 10)
217 (3, 3)
218 (19, 2)
219 (9, 6)
220 (16, 13)
221 (14, 4)
The init is at row: 15 and col: 13
The goal is at row: 29 and col: 13
```

tricky results



Nodes Expanded					
	Algorithm				
Maze	BFS	DFS	Greedy Best First	A*	modified A*
smallMaze.txt	94	53	39	49	
mediumMaze.txt	274	176	78	220	
bigMaze.txt	620	439	463	539	
greedyEasyMaze.txt			78	93	
astarEasyMaze.txt			62	35	
smallSearch.txt					29
trickySearch.txt					59
mediumSearch.txt					106
bigSearch.txt					227
Path Length/Cost					
	Algorithm				
Maze	BFS	DFS	Greedy Best First	A*	modified A*
smallMaze.txt	19	29	29	19	
mediumMaze.txt	68	174	74	68	
bigMaze.txt	210	210	210	210	
greedyEasyMaze.txt			46	28	
astarEasyMaze.txt			22	22	
smallSearch.txt					
trickySearch.txt					
mediumSearch.txt					
bigSearch.txt					

References:

http://web.engr.illinois.edu/~slazebni/spring15/assignment1.html

- The assignment

http://cplusplus.com

- Libraries for… fstream, iostream, sleep in unistd.h, stack, queue, priority queue, math, overloading operators, etc

https://math.stackexchange.com/questions/139600/euclidean-manhattan-distance

- Manhattan distances

https://en.wikipedia.org/wiki/Breadth-first_search

- BFS pseudo code

https://en.wikipedia.org/wiki/Depth-first_search

- DFS pseudo code

http://web.engr.illinois.edu/~slazebni/spring15/lec06_informed_search.pdf

- Greedy and A* concepts

https://en.wikipedia.org/wiki/Best-first_search

- Greedy pseudo code

https://en.wikipedia.org/wiki/A*_search_algorithm

- A* pseudo code

http://www.laurentluce.com/posts/solving-mazes-using-python-simple-recursivity-and-a-search/

- More A* concepts

Contributions:

Harrison: coding, images, table
John: coding
Ellie: report, references