**Artificial Intelligence Assignment 1**

Jiaming Deng 22302794

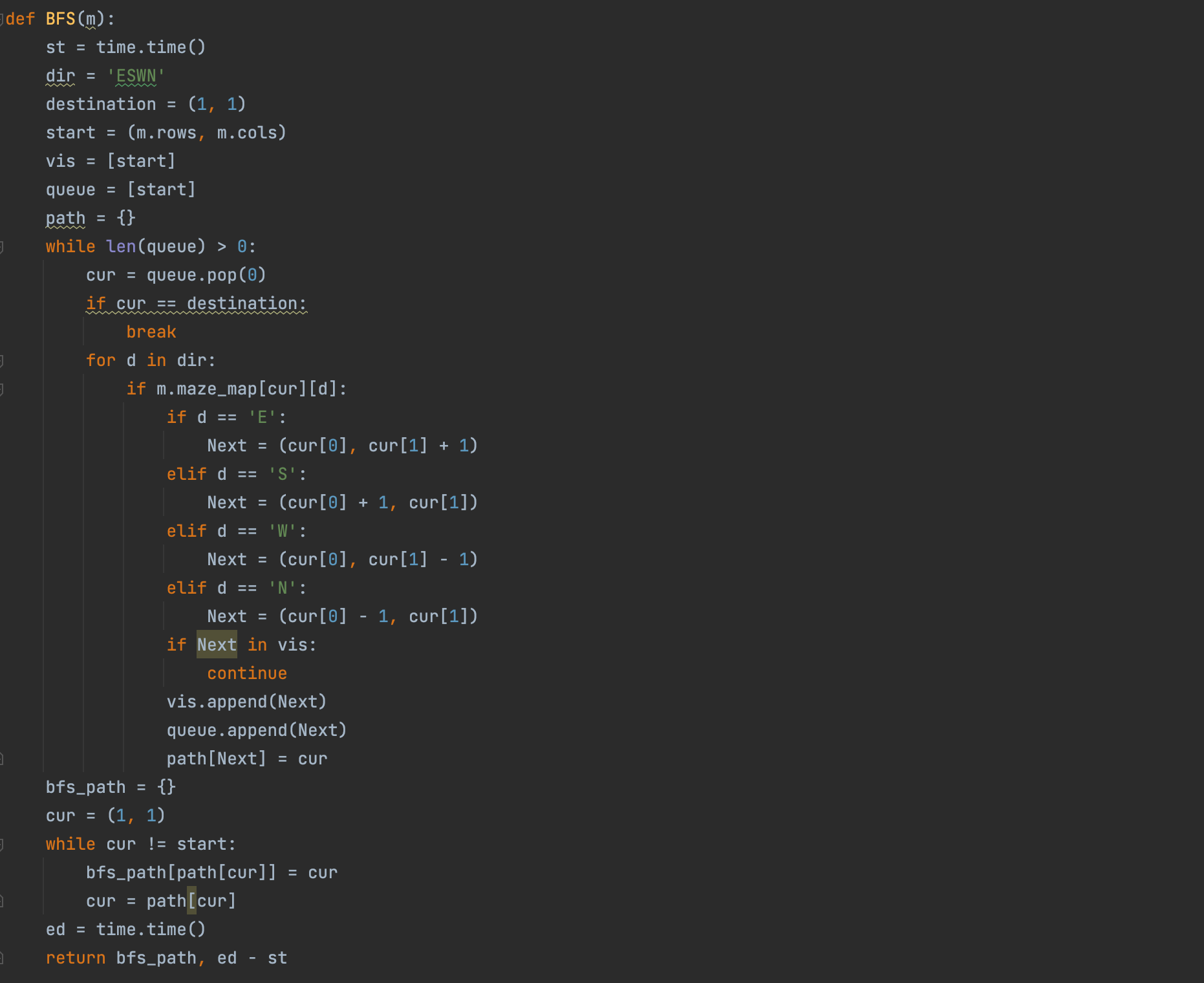
1. **Algorithm implementation**

In this part, I introduce the 5 codes that the job requires to implement, namely BFS, DFS, A-Star, MDP value iteration, MDP policy iteration, and explain my implementation process.

1. BFS

The BFS algorithm visits all vertices of the graph in breadth-first order, that is, visits all vertices of the same layer, and then proceeds to the next layer. The algorithm starts from a given source vertex and explores all vertices reachable from that source vertex in breadth-first order.

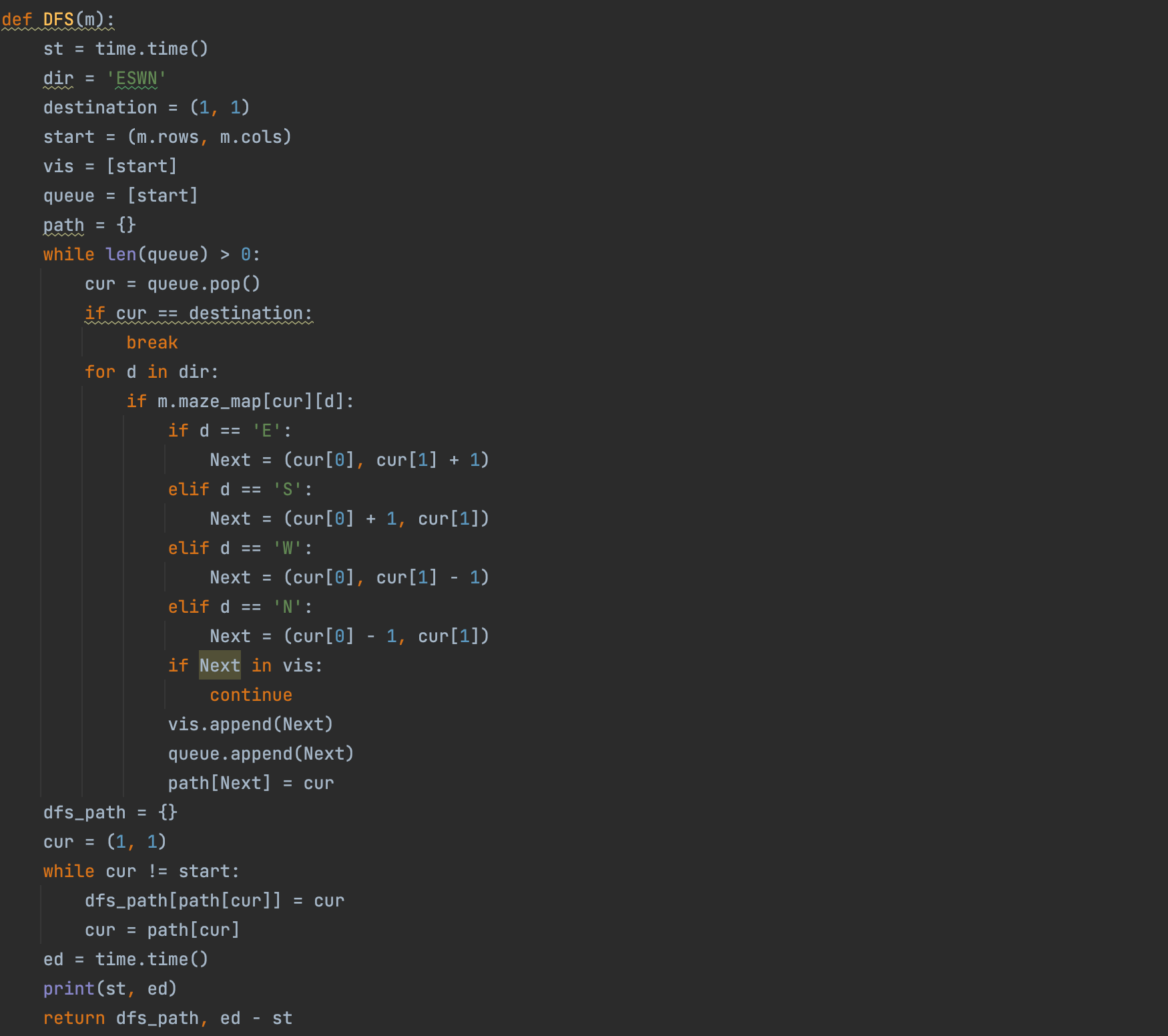
The maze is generated using the pyamaze library, so the input parameter of BFS is the maze m. In order to obtain the running time of the program, the current time is obtained at the beginning and end of the program respectively. The BFS code starts from the source vertex and adds the source vertex to the queue. When the queue is not empty, take a vertex from the queue and mark it as visited. For each unvisited neighbor of a dequeued vertex, enqueue it and mark it as visited. Repeat the above steps until the queue is empty.



1. DFS

The DFS algorithm visits all vertices of the graph in depth-first order, that is, starting from the starting vertex, all vertices of the path are visited before backtracking. The algorithm starts from a given source vertex and explores along each branch as possible before backtracking.

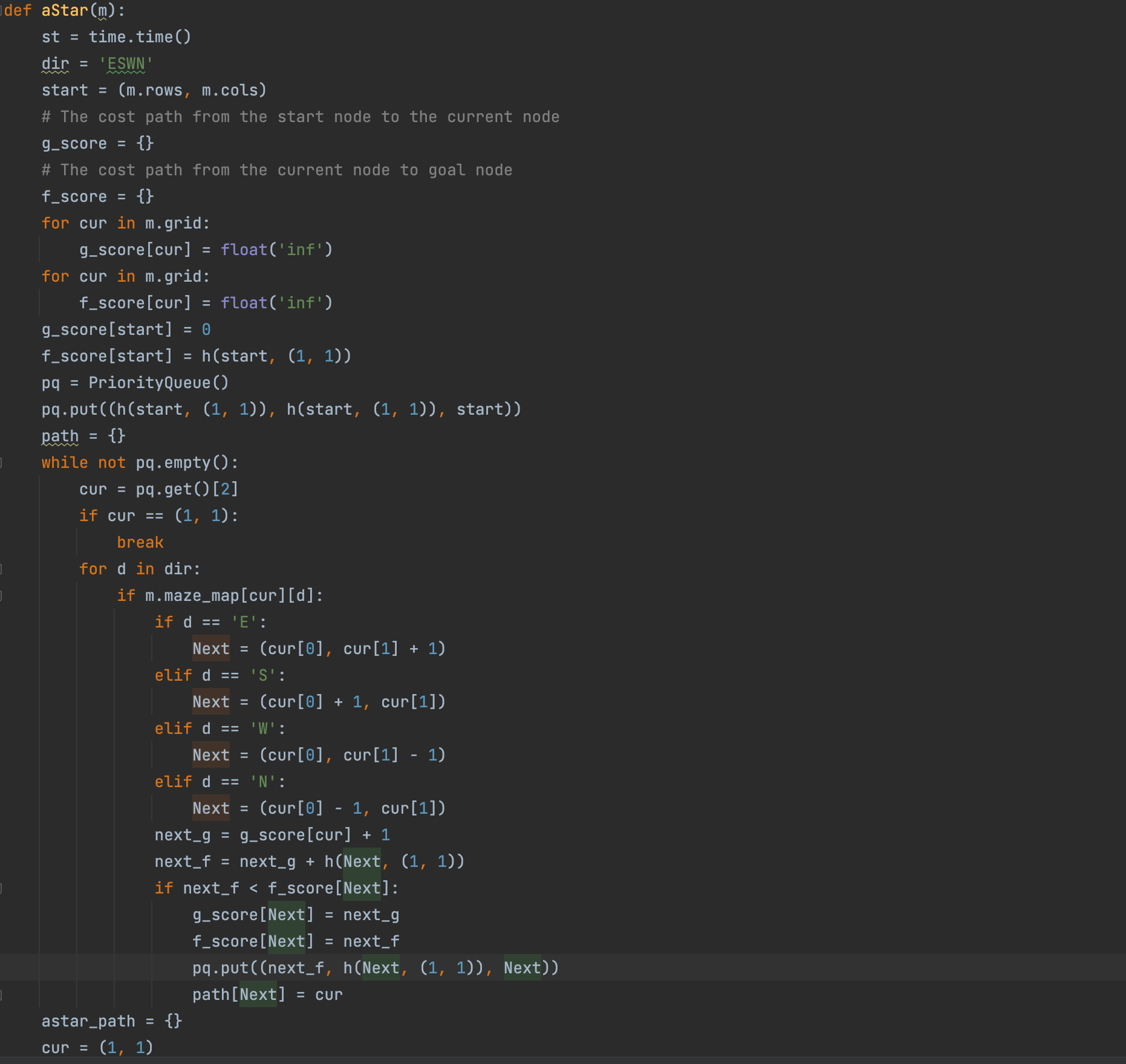
In my code, DFS starts with the source vertex, marking the source vertex as visited. For each unvisited neighbor of the source vertex, recursively treat them as the source vertex. Backtrack to the previous vertex until all paths have been explored.



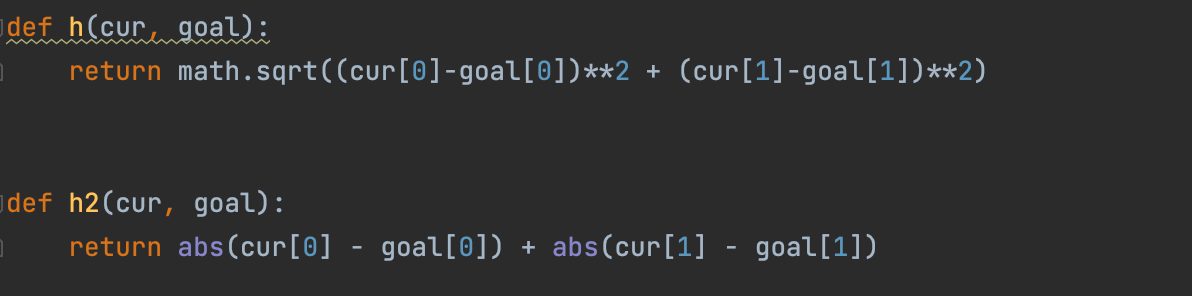
1. A-Star

A\* is a popular heuristic search algorithm. It chooses paths by dynamically computing the score of transferable points, that is, it uses heuristics to guide the search toward the goal.

The A-star algorithm starts from the source vertex and initializes the cost of reaching it (g) to 0, calculates the heuristic cost (h) of reaching the target vertex from the current vertex, calculates the total cost of reaching the target vertex from the current vertex and Placed in priority queue, sorted by total cost (f). When the priority queue is not empty, dequeue the vertex with the lowest total cost and mark it as visited. If the dequeued vertex is the destination vertex, stop and return the path from the source vertex to the destination vertex. For each unvisited neighbor of a dequeued vertex, compute the cost to reach it from the source vertex and compute its heuristic cost to reach the destination vertex. Enqueue neighboring vertices and their calculated total costs into a priority queue. Repeat the above steps until the priority queue is empty or the target vertex is found.

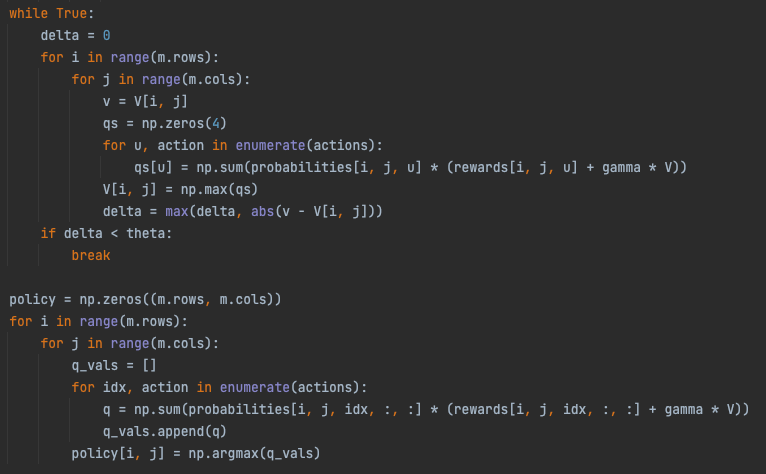


I tried two evaluation functions, the first is using Euclidean distance and the second is using Manhattan distance. The second performance is better after testing.

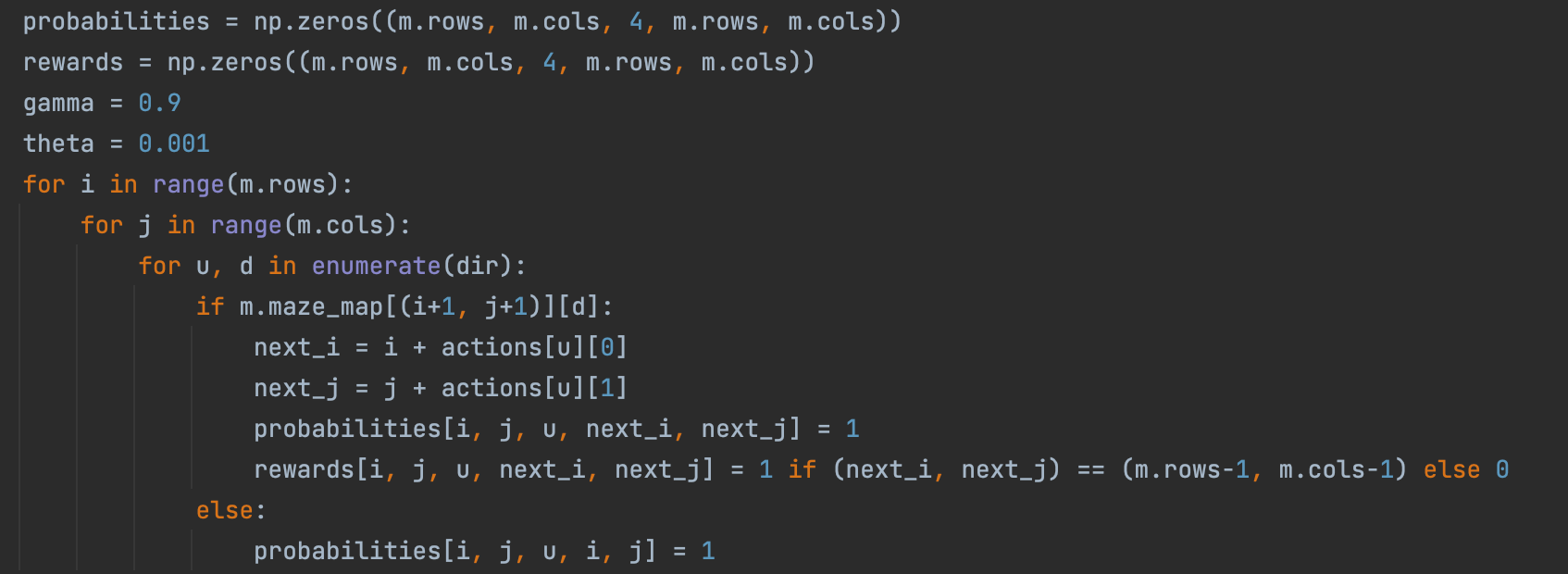


1. MDP Value Iteration

The value iteration algorithm works by starting with an initial estimate of the value function for each state, and then iteratively updates the value of each state until convergence. At each iteration, the algorithm updates the value function for each state by maximizing the expected reward over all possible actions that can be taken in that state.



I set gamma to 0.9, theta to 0.001, and set the probabilities of being between two points without walls to 1, otherwise set the probabilities of the note to itself as 1, and set the reward to the end point to 1.

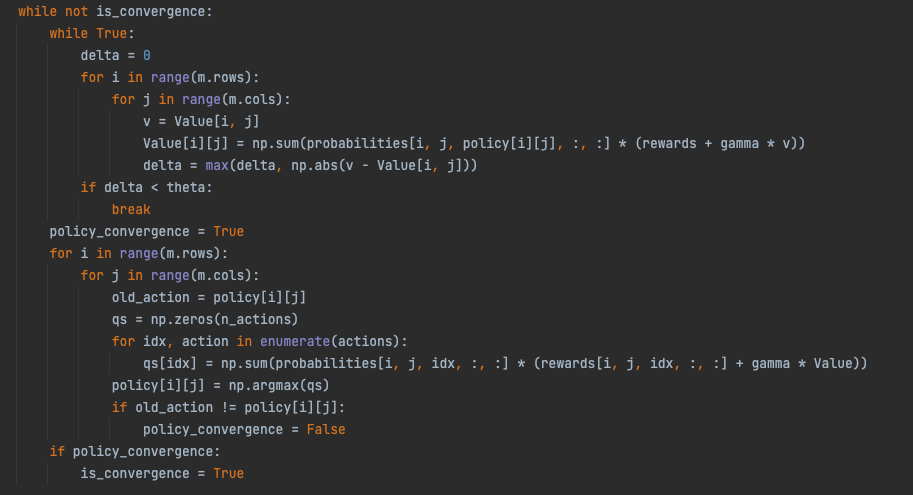


1. MDP Policy Iteration

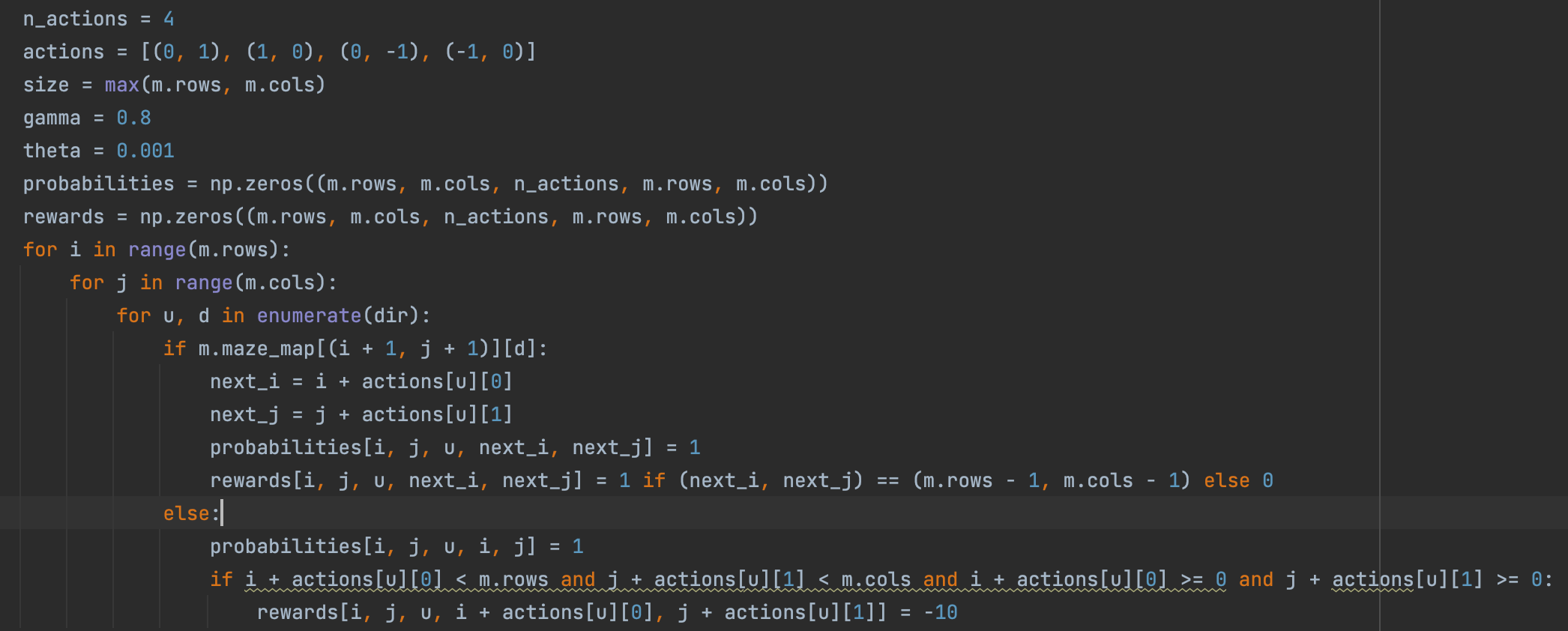
The MDP policy iteration principle consists of two main steps:

Policy Evaluation: In this step, the value function for a given policy is evaluated. The value function represents the expected total reward that can be obtained following a given policy. The value function is updated iteratively until convergence.

Policy Improvement: In this step, the current policy is improved based on the current value function. New policies are created by choosing actions that maximize the expected total reward for each state until convergence is achieved.



I set gamma to 0.8, theta to 0.001, and set the probability that there is no wall between two points to 1, otherwise I set the probability of the note to itself to 1. Set reward to -10 if there is a wall between the two points.

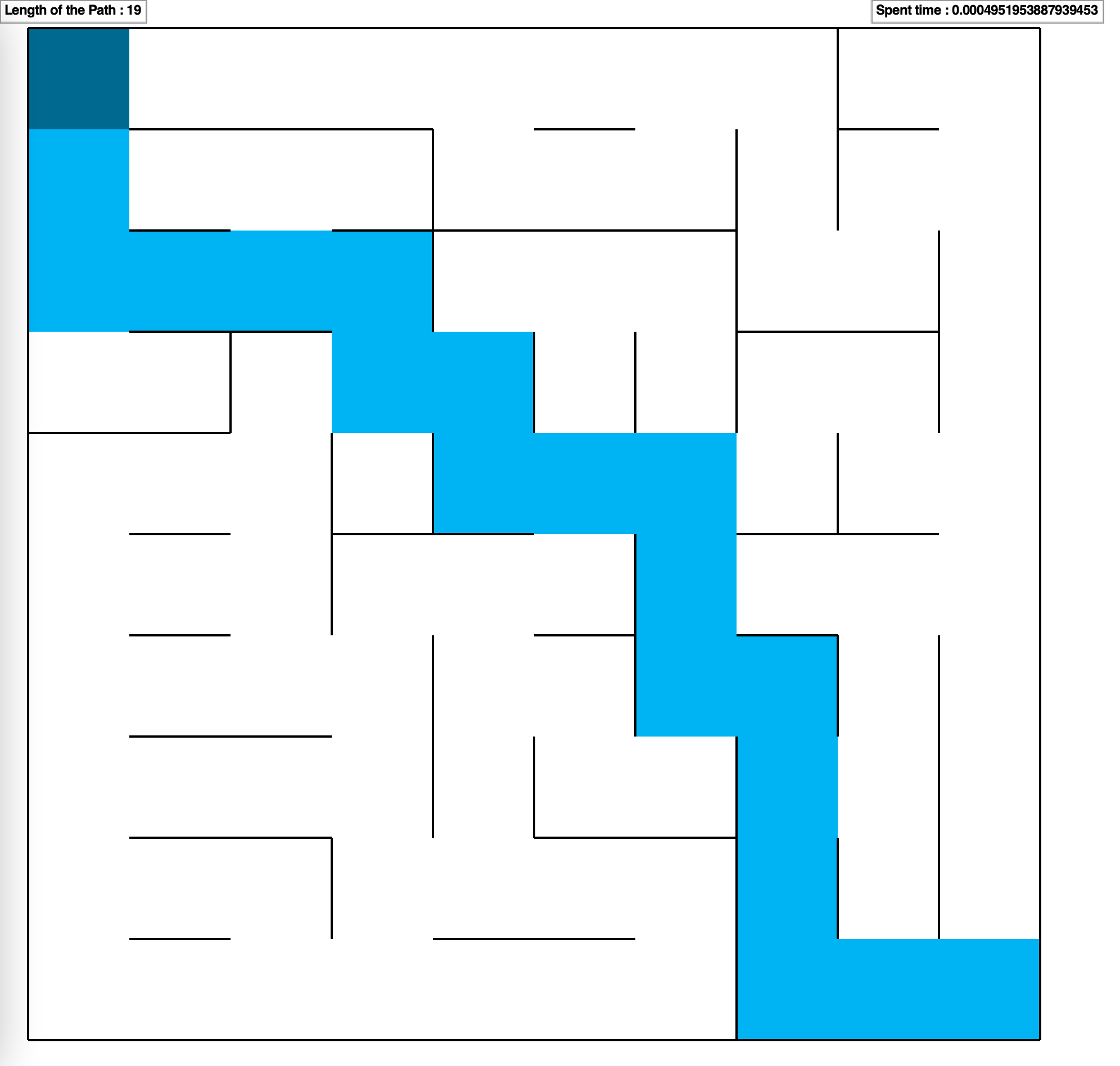
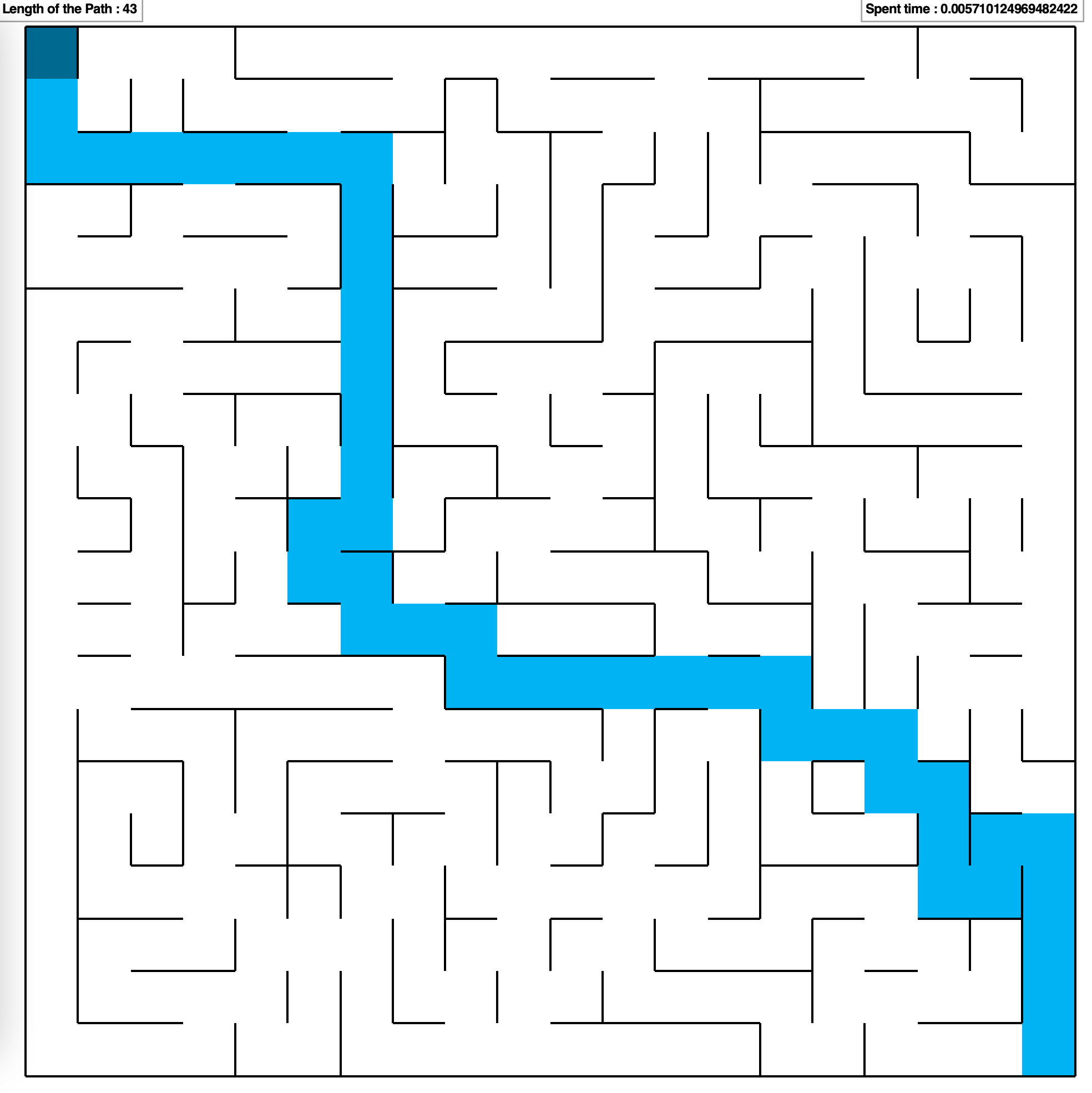
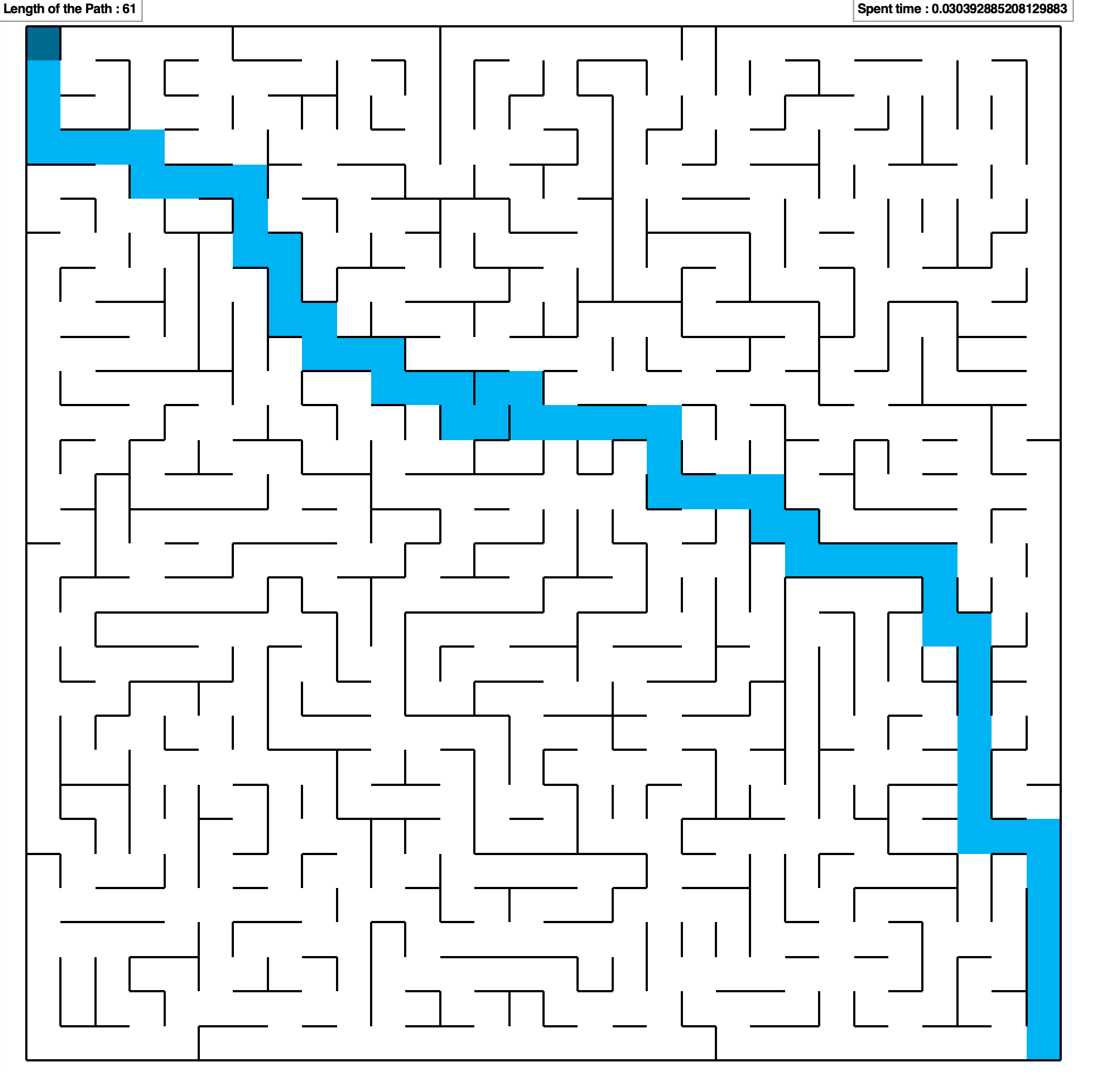


1. **Results of the Algorithms**

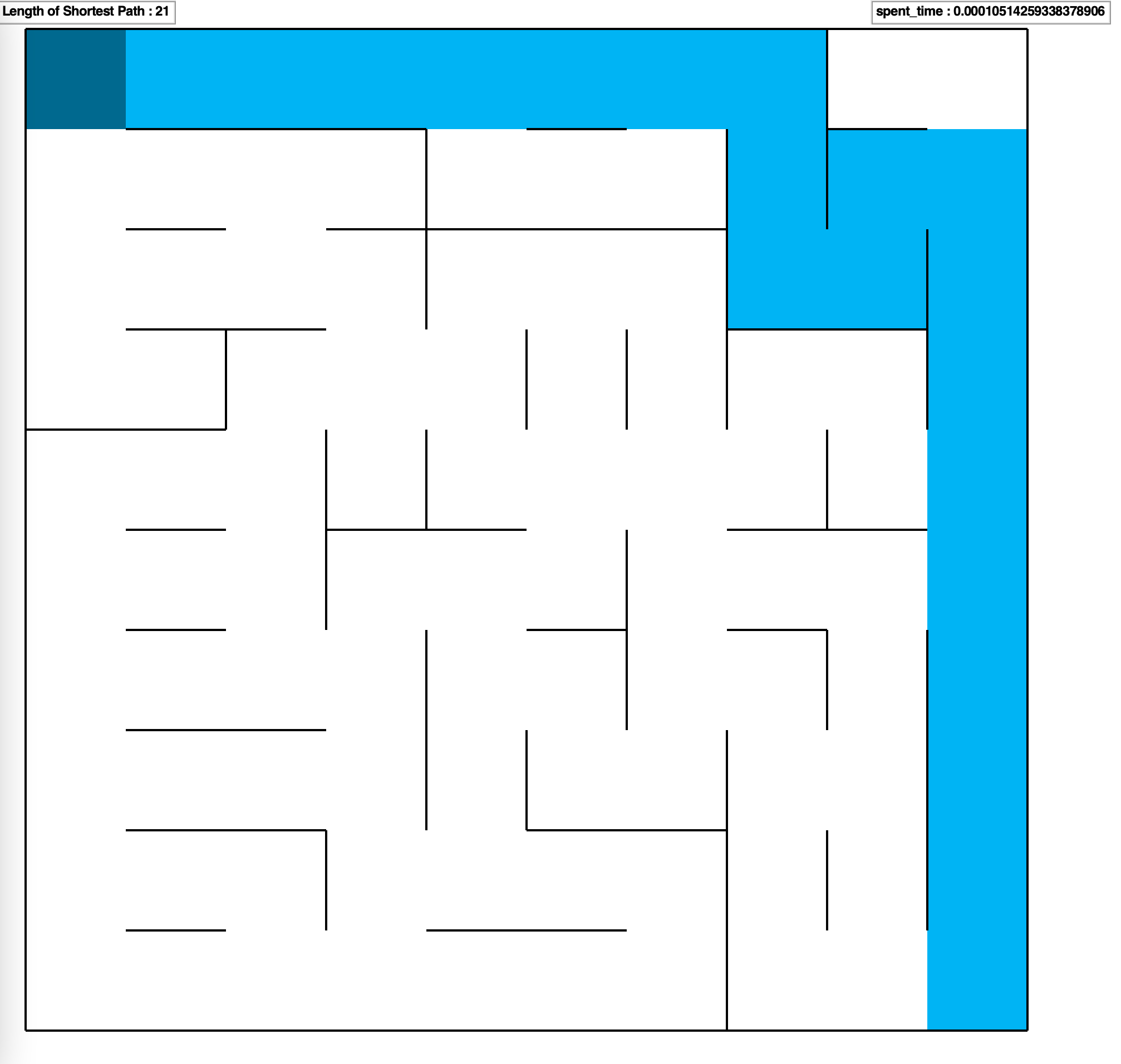
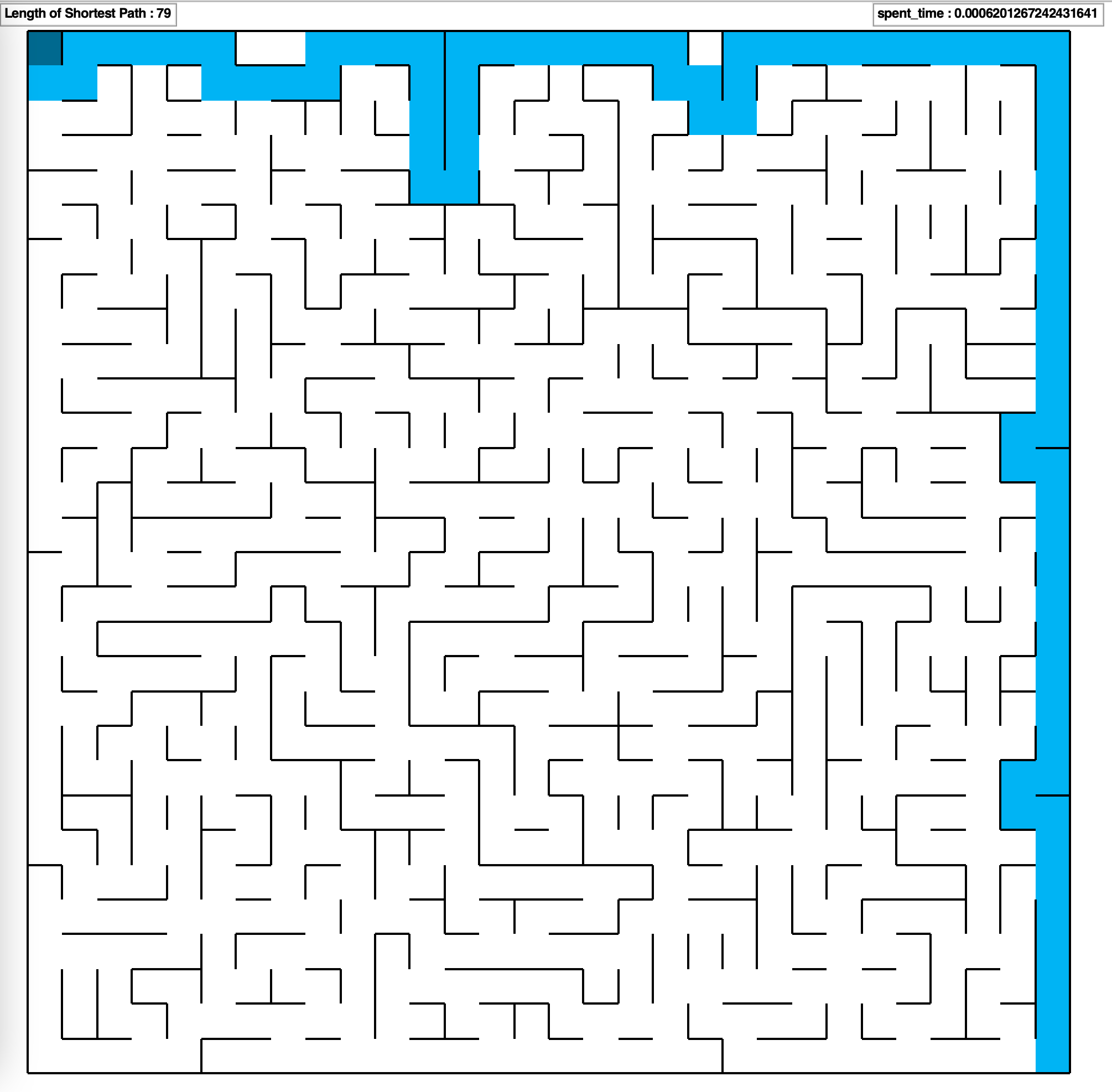
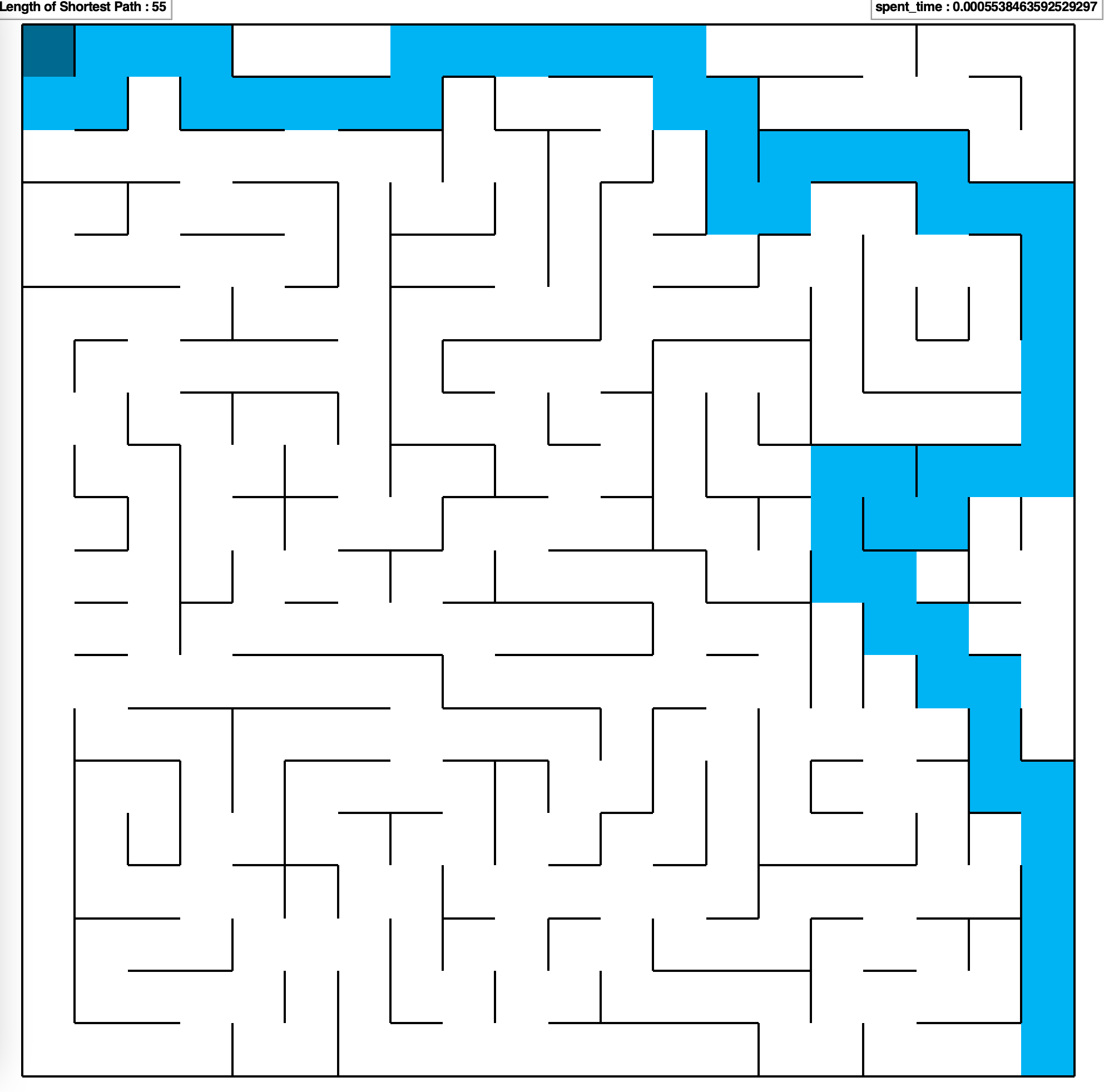
This section shows the experimental results of the five algorithms, including path selection, the length of the shortest path found, and the time spent.

Firstly, the path selection of each algorithm is shown. The three pictures from left to right are 10x10, 20x20, and 30x30 mazes. The starting point of the maze is the lower right corner and the ending point is the upper left corner. The blue part represents the path through the maze.

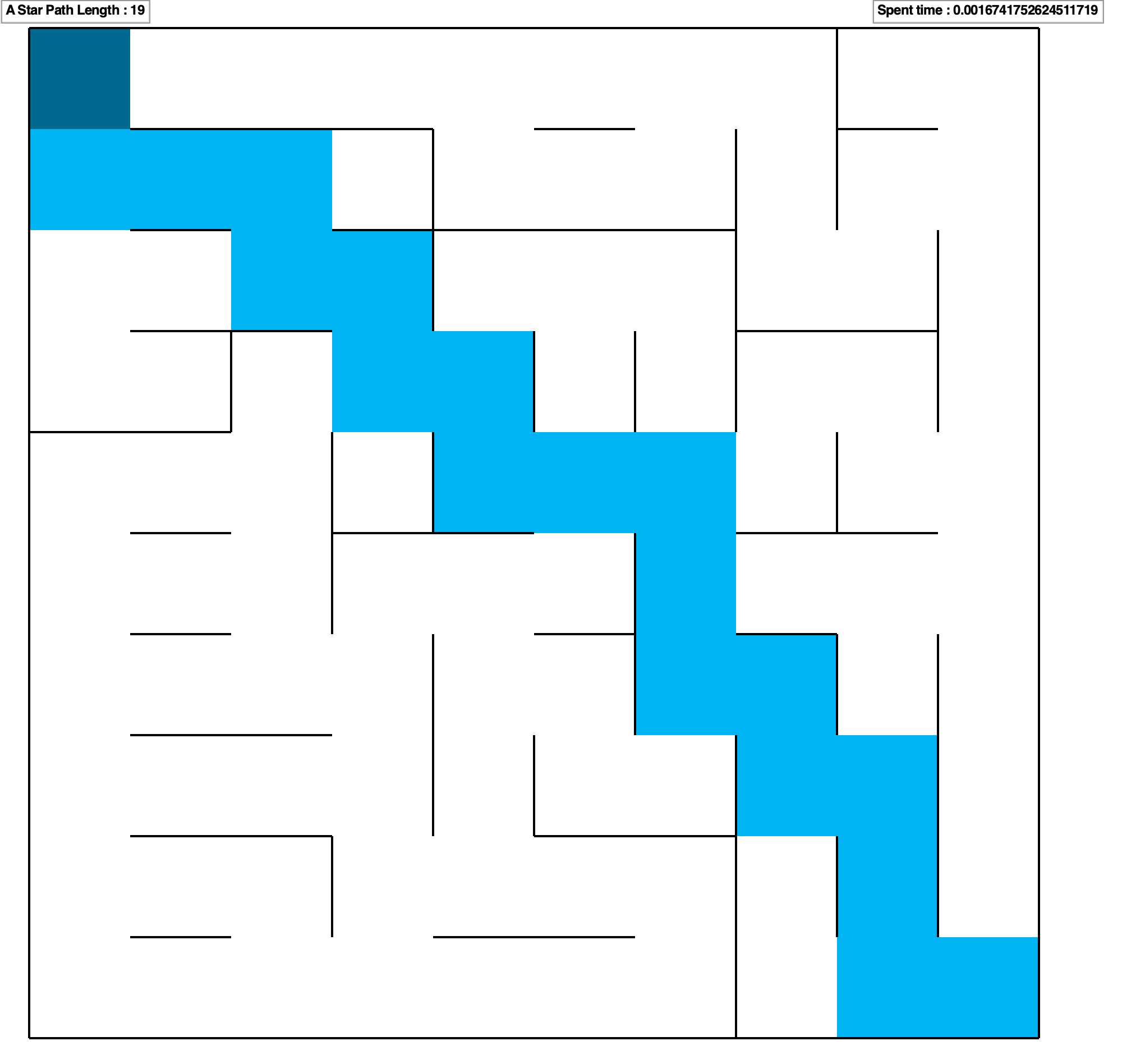
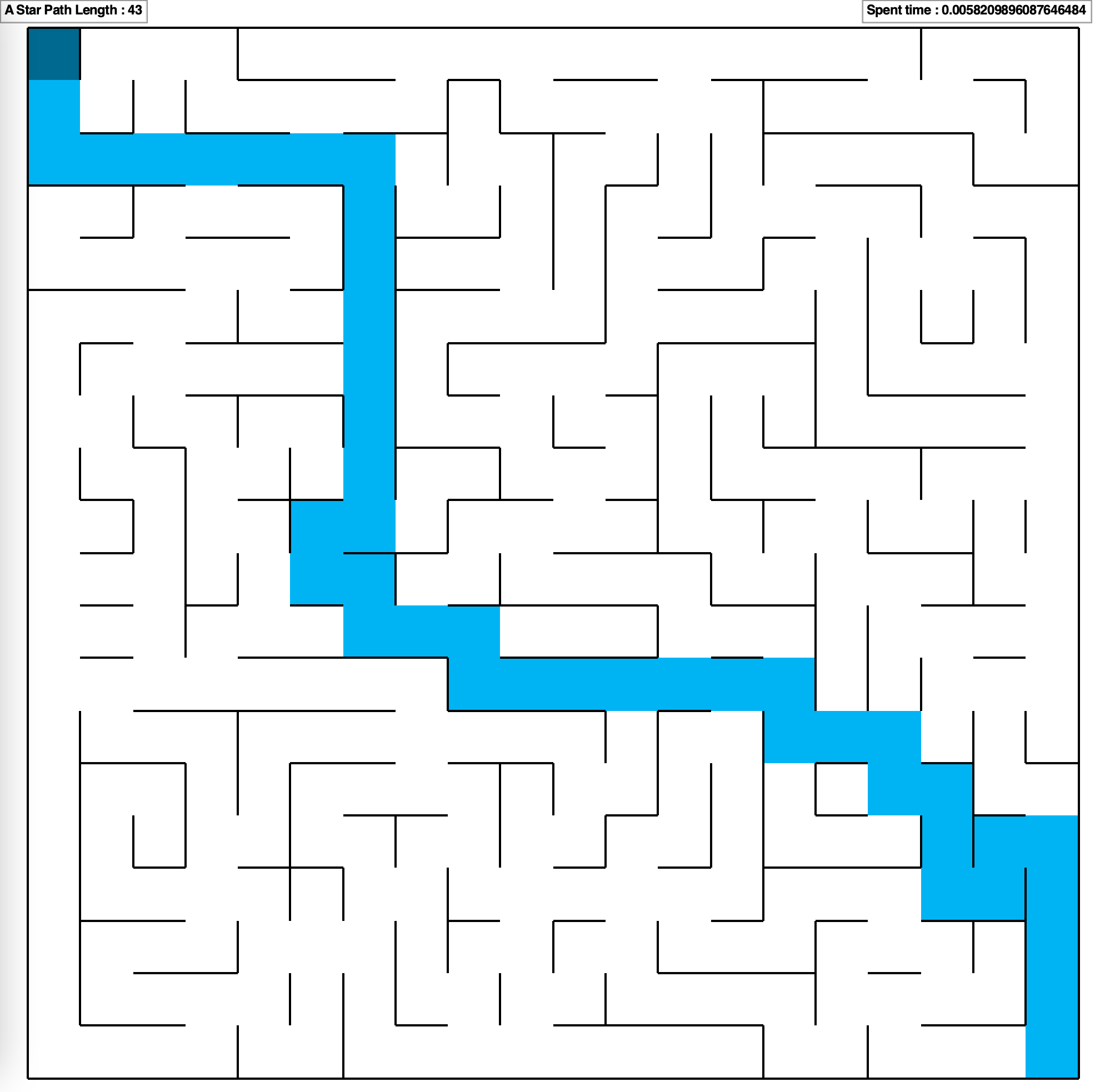
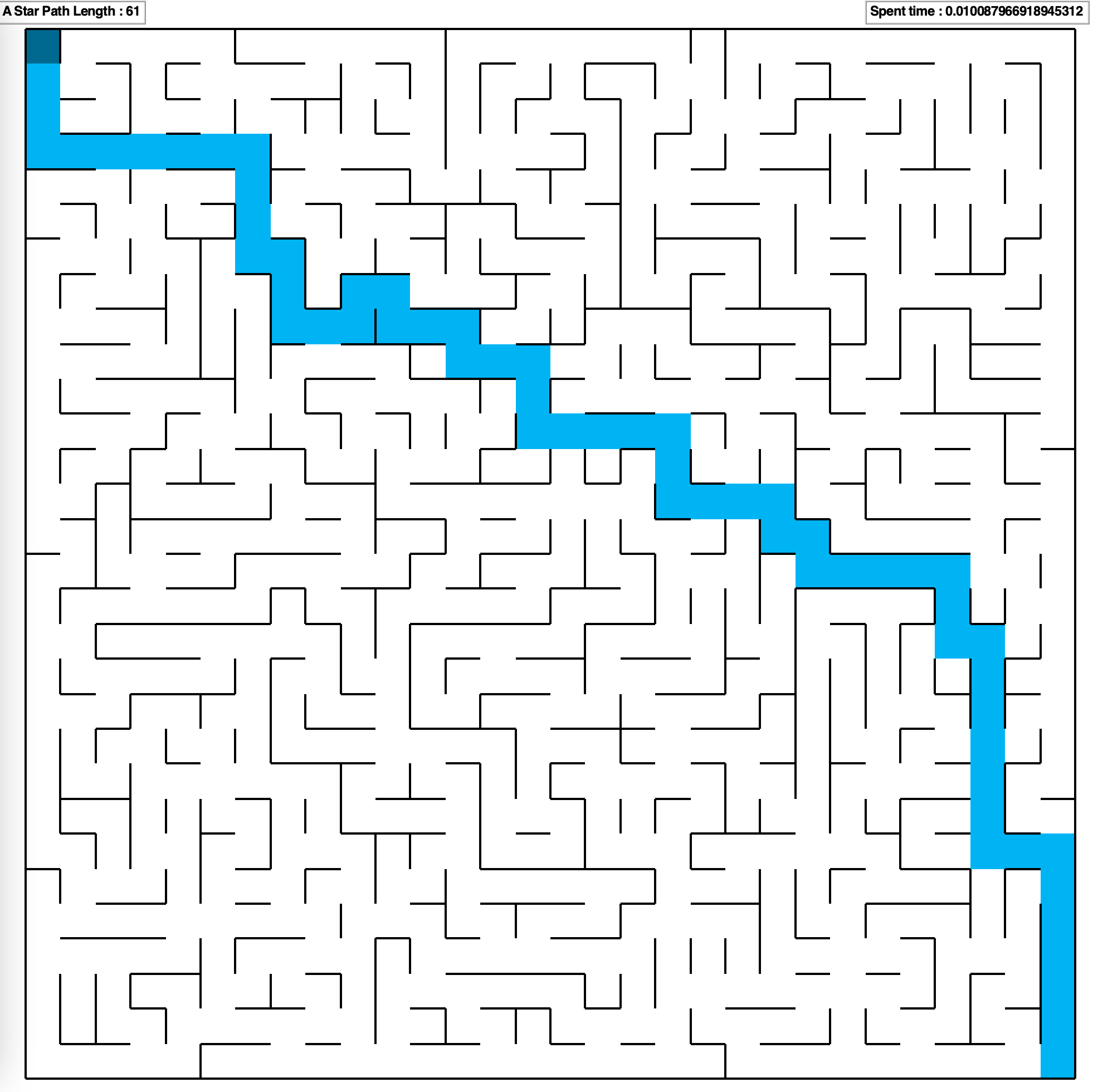
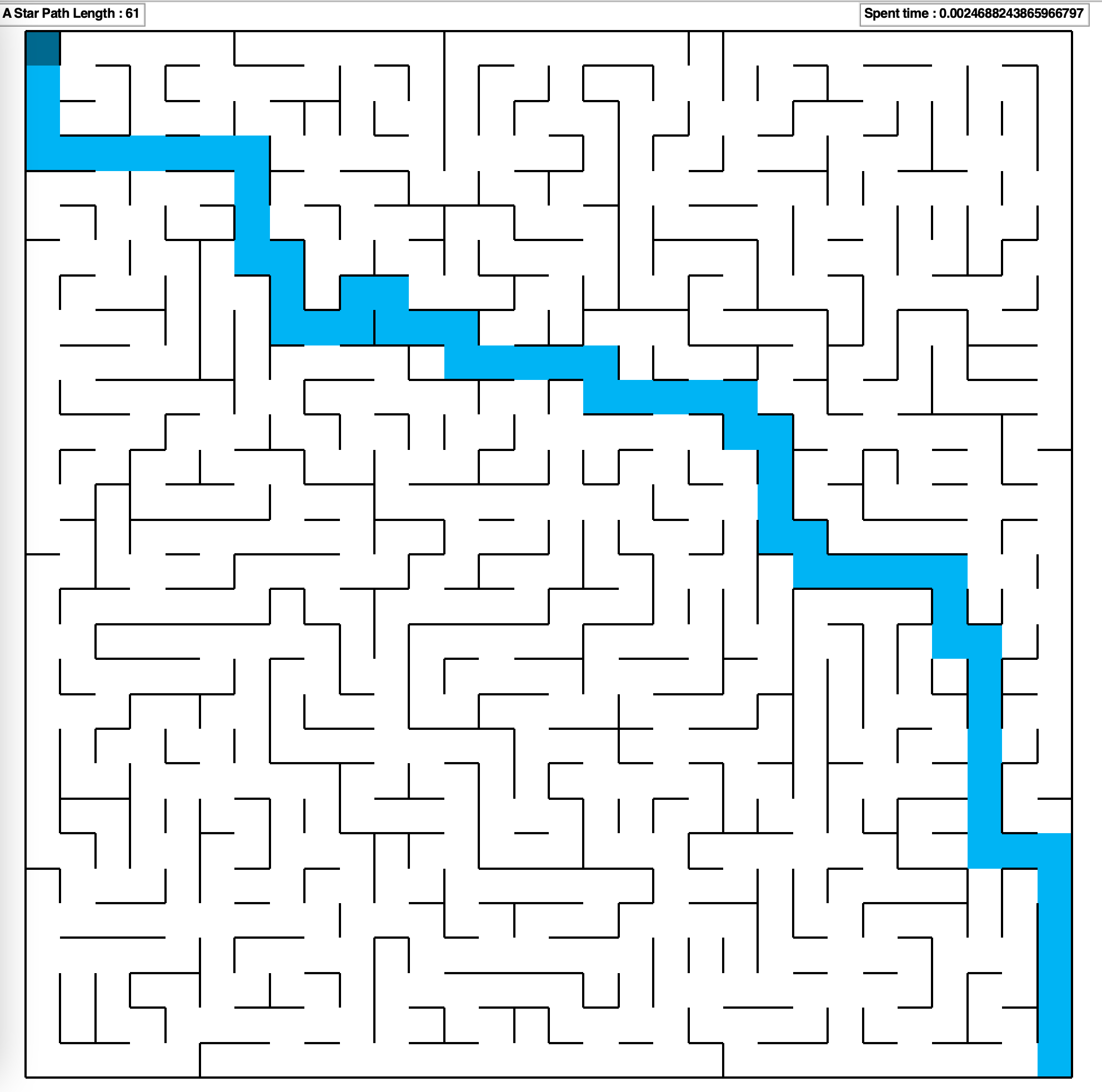
1. BFS



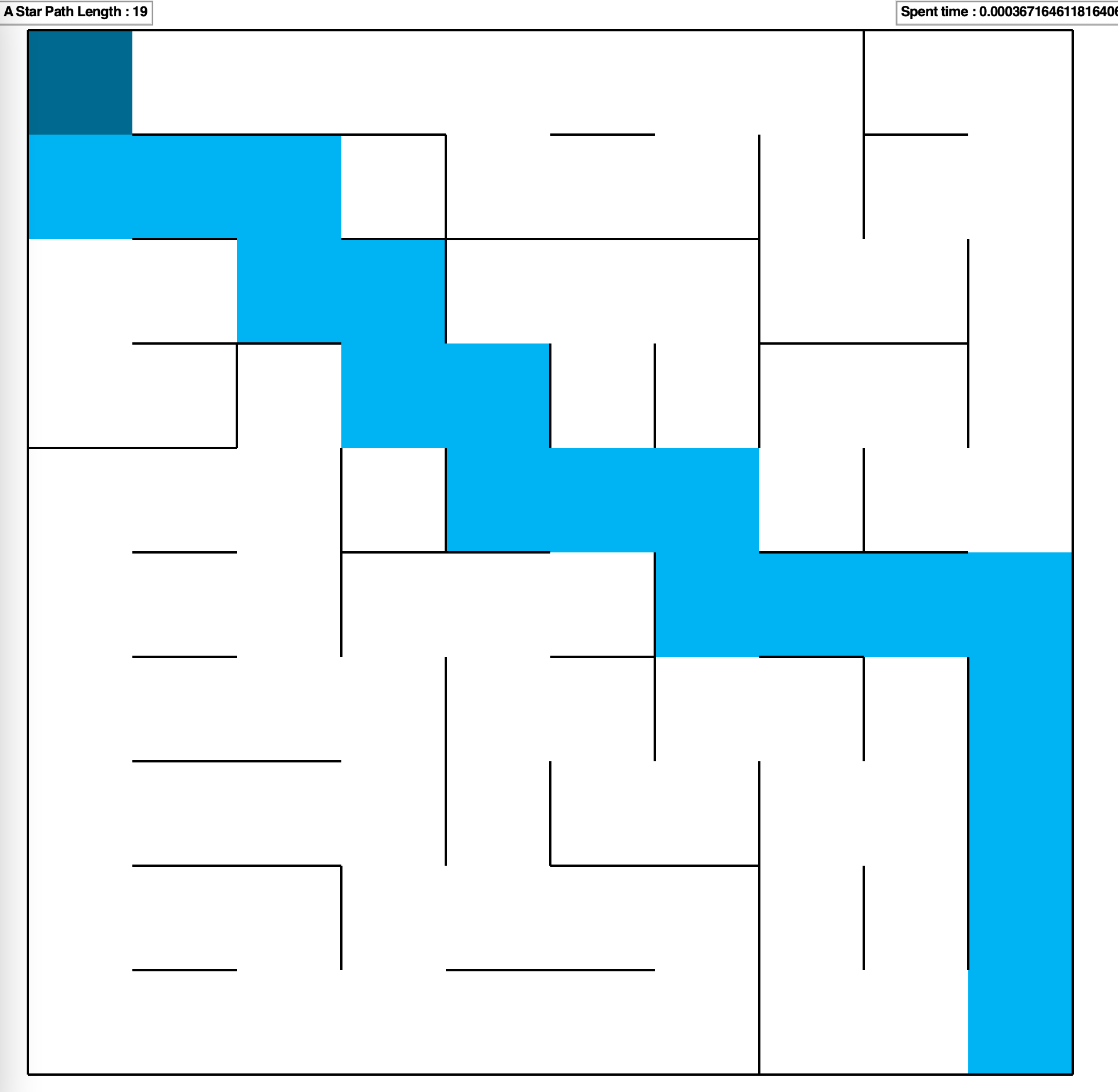
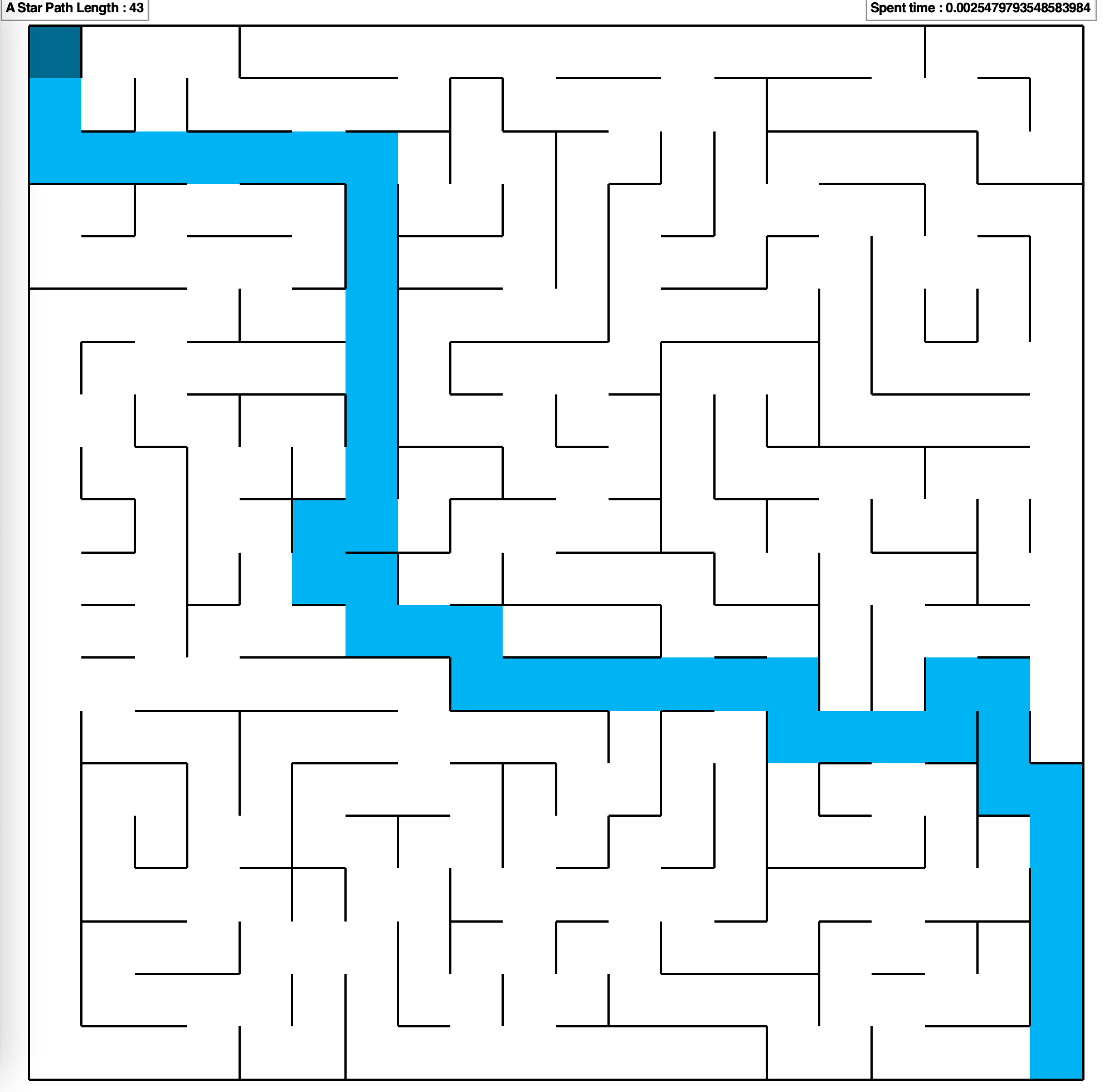
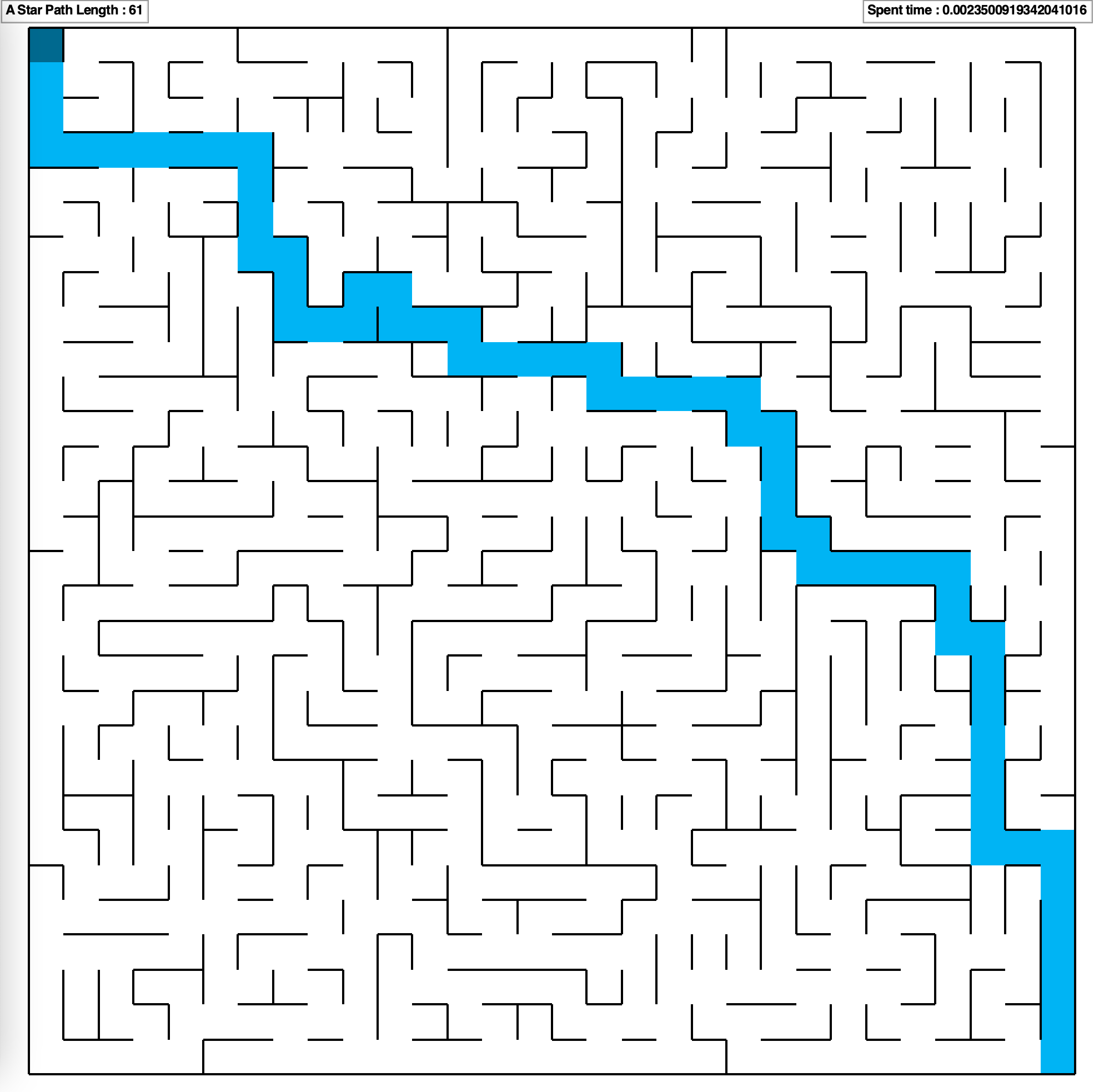
1. DFS



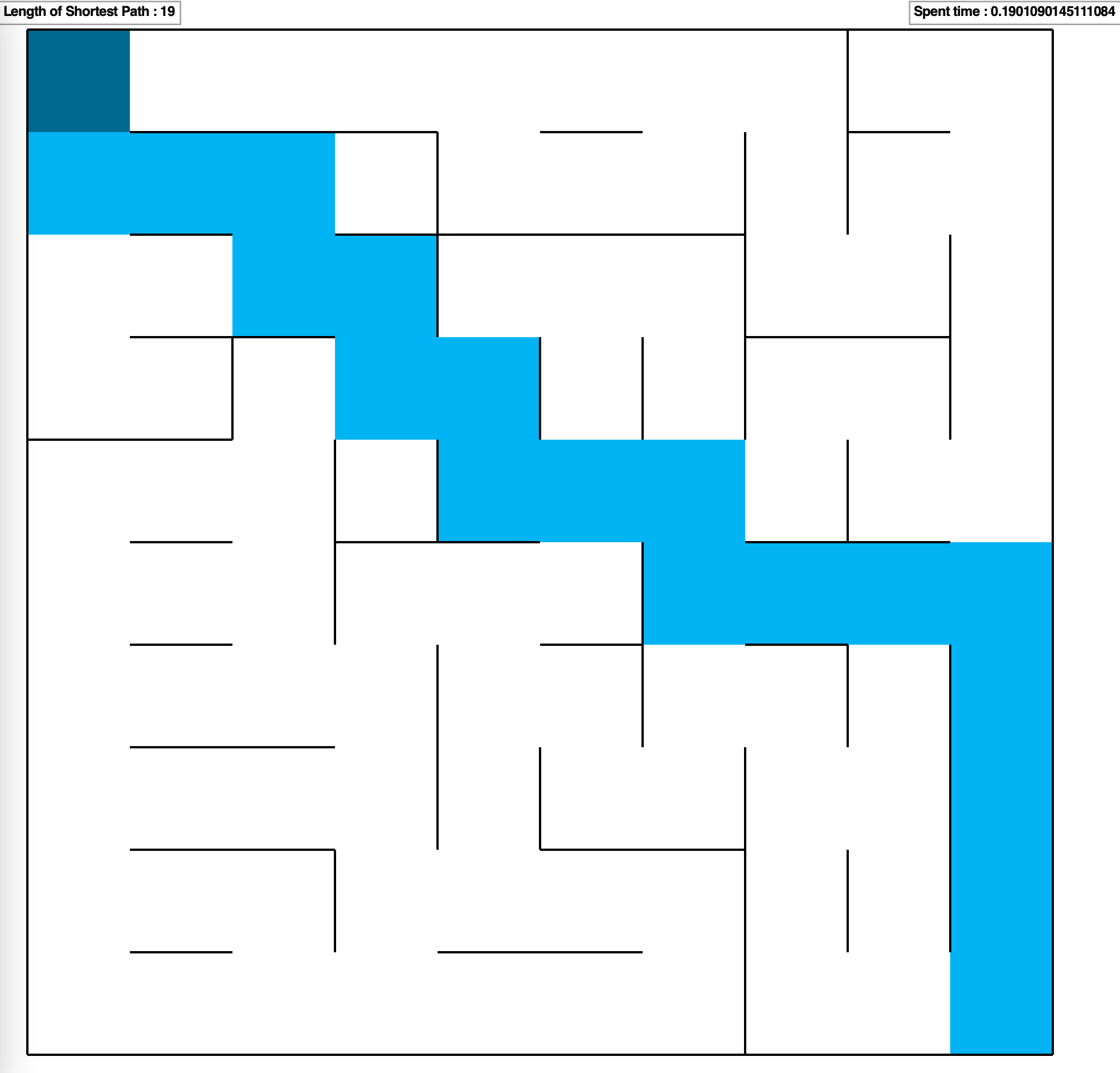
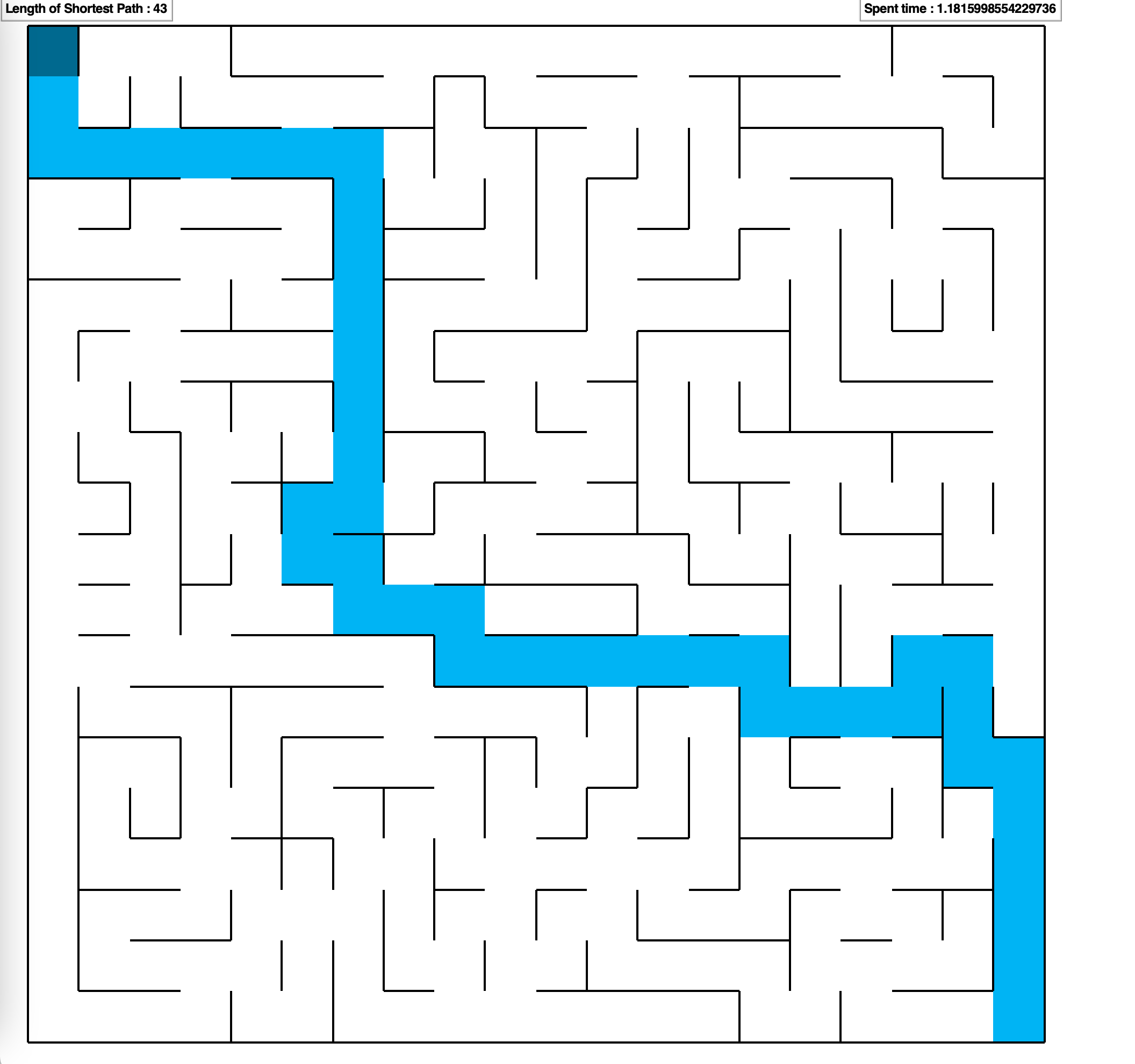
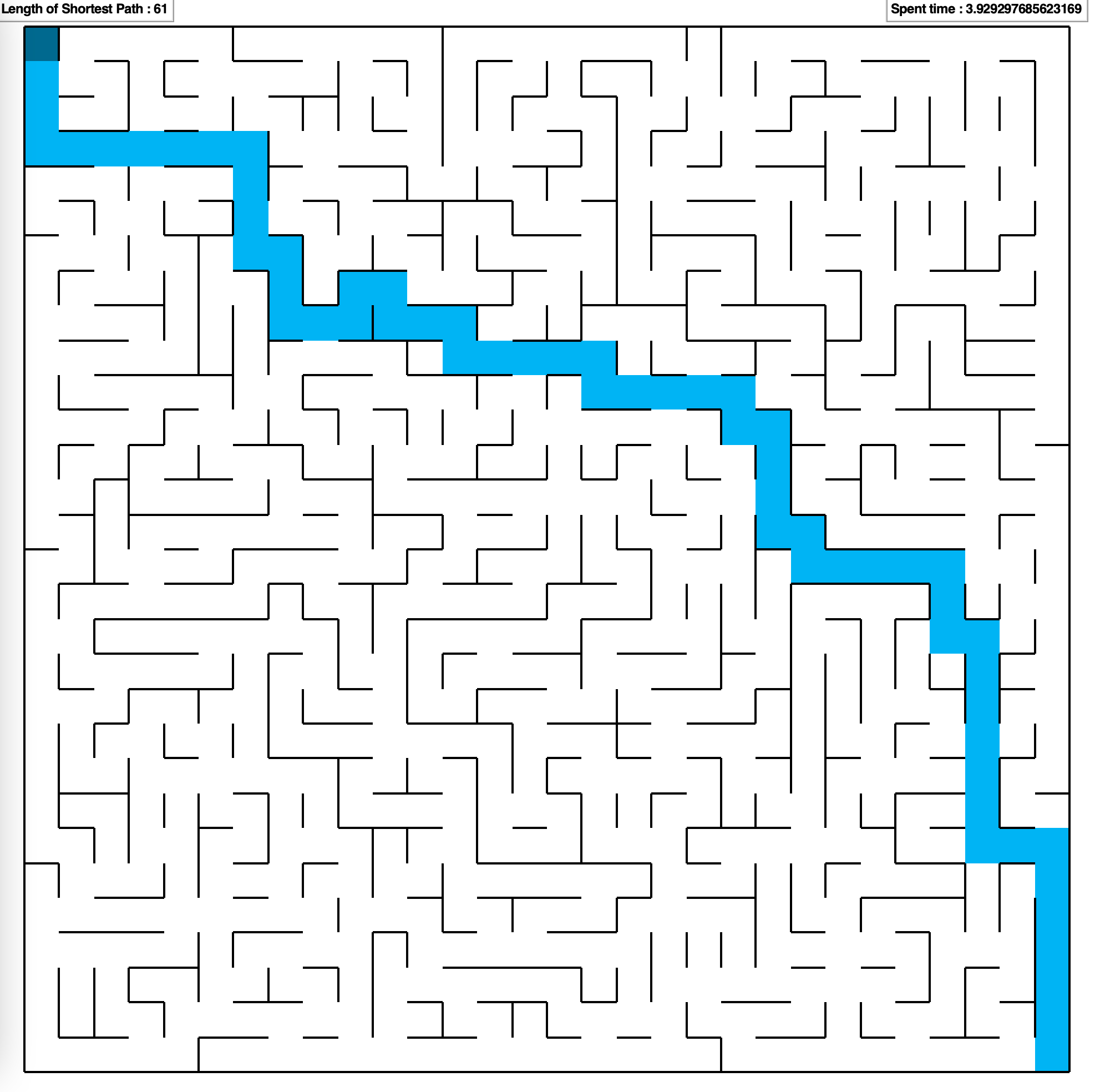
1. A\* Euclidean distance



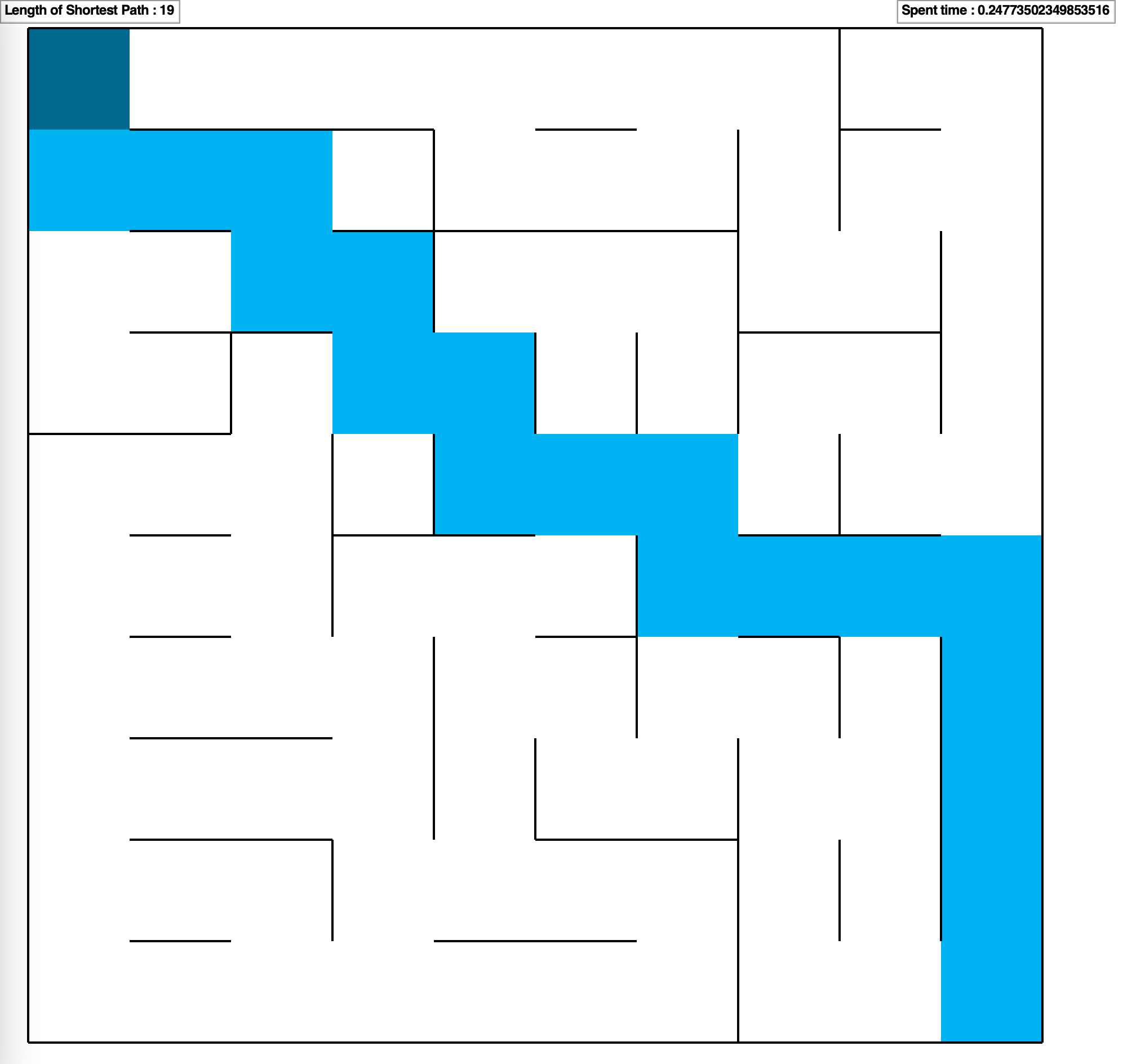
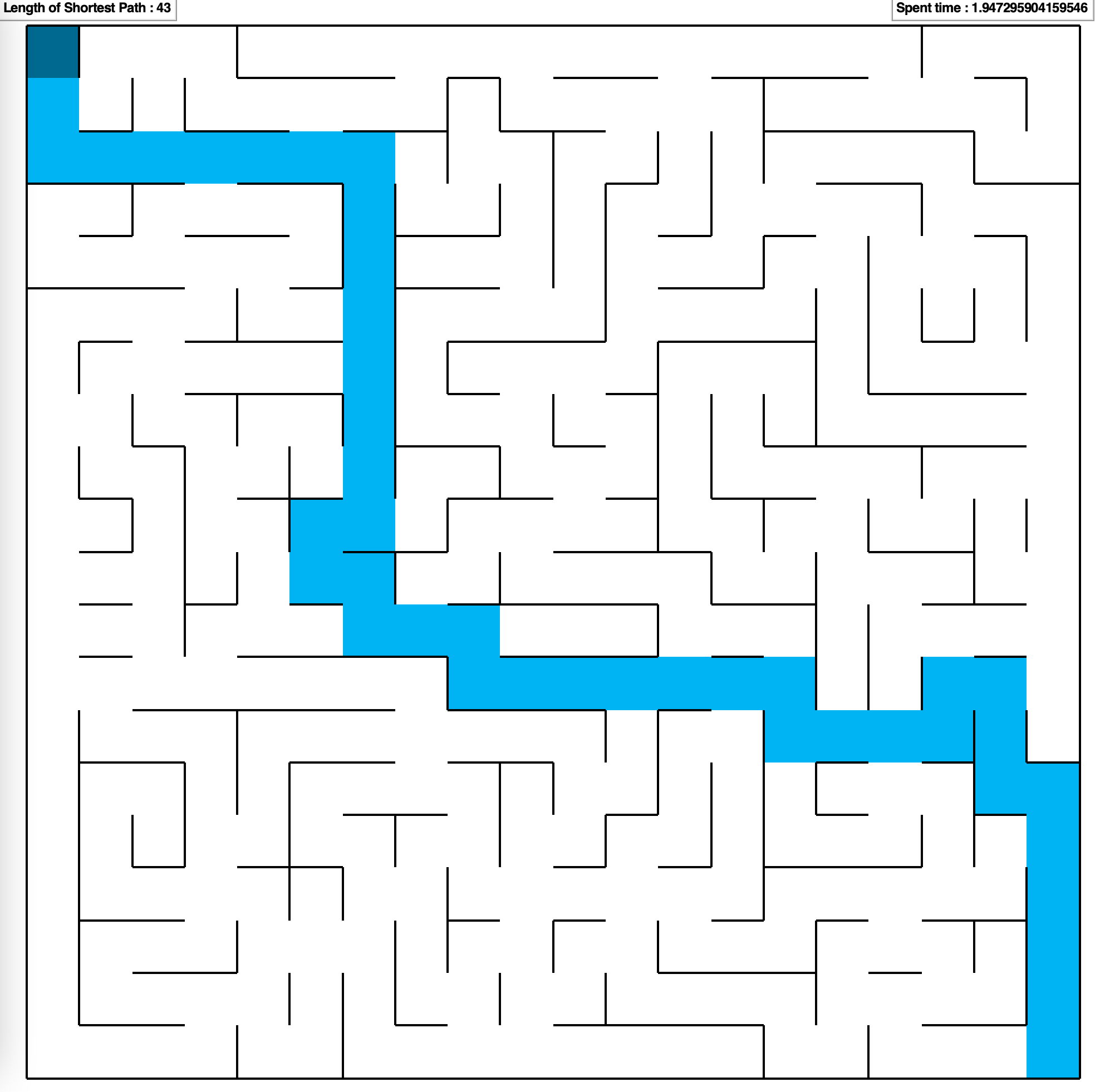
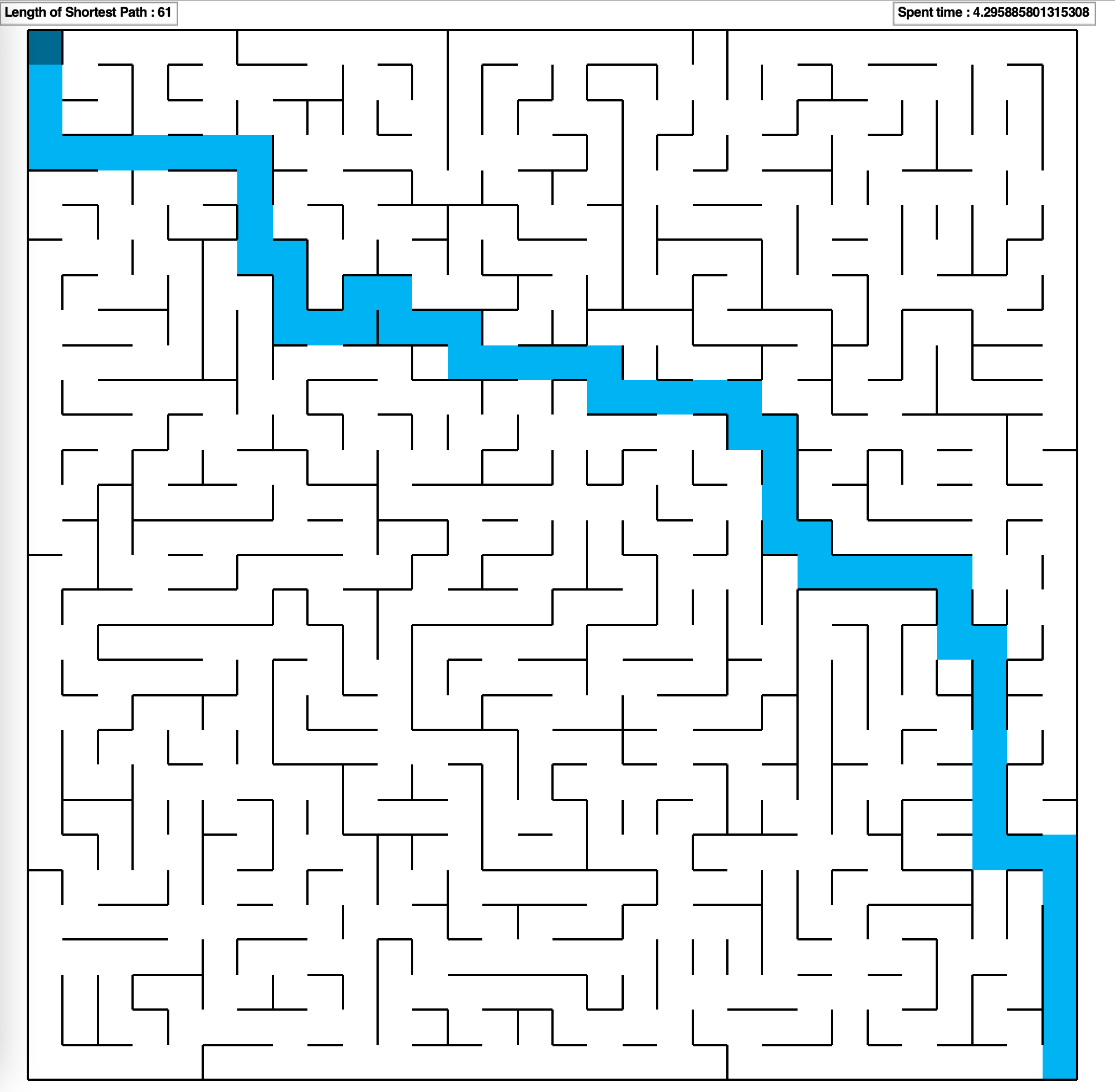
1. A\* Manhattan distance



1. MDP Value Iteration



6) MDP Policy Iteration



Secondly, the table below shows the running time of each algorithm on maps of different sizes and the length of the shortest path selected. Because the space complexity is mainly determined by the size of the map, the space complexity of the data structures used by each algorithm is not much different, so we mainly compare the execution time of the algorithms.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | BFS | DFS | A\*Euclidean distance | A\*Manhattan distance | MDP Value Iteration | MDP Policy Iteration |
| 10x10 | 0.00049 | 0.00011 | 0.0017 | 0.00037 | 0.19 | 0.25 |
| 20x20 | 0.0057 | 0.00056 | 0.0058 | 0.0025 | 1.18 | 1.94 |
| 30x30 | 0.0304 | 0.00062 | 0.01 | 0.0025 | 3.93 | 4.29 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | BFS | DFS | A\*Euclidean distance | A\*Manhattan distance | MDP Value Iteration | MDP Policy Iteration |
| 10x10 | 19 | 21 | 19 | 19 | 19 | 19 |
| 20x20 | 43 | 55 | 43 | 43 | 43 | 43 |
| 30x30 | 61 | 79 | 61 | 61 | 61 | 61 |

1. **Comparison of the Algorithms**
2. Comparison between BFS, DFS and A\*

On three mazes of different sizes, except that A\* uses Manhattan distance as the evaluation function, DFS always finds the path to solve the maze with the smallest time. However, the path length found by DFS is always longer, because DFS is a depth-first search algorithm, and the search priorities in different directions are different. If a path successfully finds the end point, the result will be returned directly, so the speed is very fast, but the result is not necessarily the best.

BFS and A\* algorithms can always find the shortest path, but BFS is faster than A\* algorithm that uses Euclidean distance as the evaluation function. The reason is that the maze can only go horizontally and vertically, and the Euclidean distance is to calculate the geometric distance between two points, so it is not very suitable for the maze problem.

The A\* algorithm using Manhattan distance as the evaluation function is the fastest, and there is no significant increase in time when the maze becomes larger, which is in sharp contrast to the BFS and DFS algorithms. Because the time complexity of the BFS and DFS algorithms is O(V+E), V is the number of edges, and E is the number of points. When the number of edges and points increase at the same time, it will also increase. A\* is a heuristic algorithm, and the direction to choose to move is based on the evaluation of the evaluation function, which is not significantly changed by the change of the map size.

1. Comparison between MDP Value Iteration and MDP Policy Iteration

In the MDP algorithm, value iteration or policy iteration can be used to find the shortest path to solve the maze, but the value iteration algorithm takes less time than policy iteration. Two reasons are as follows.

1. Convergence speed: Value iteration usually converges faster than policy iteration. This is because policy iteration involves multiple iterations of policy evaluation and policy improvement, whereas value iteration combines these two steps in a single update.

2. Execution time: The execution time of policy iteration is usually longer than that of value iteration. This is because policy iteration requires multiple iterations of policy evaluation and policy improvement, which can be computationally expensive.

1. Comparison between search algorithm and MDP algorithm

The both of the MDP algorithms perform worse than the three search algorithms on three different mazes. They take about 100 times longer than the search algorithms. As the MDP algorithm needs to model the decision problem of a series of sequential decisions in an uncertain or random environment, but the search algorithm is used to find the shortest or best path from the start state to the goal state in a given search space. Because the maze problem is a given search space, the search algorithm performs better. But search algorithms cannot be applied to problems that involve a series of decisions and uncertain outcomes.

**Appendix**

1. BFS
2. **from** pyamaze **import** maze, agent, textLabel, COLOR
3. **import** time

6. **def** BFS(m):
7. st = time.time()
8. dir = 'ESWN'
9. destination = (1, 1)
10. start = (m.rows, m.cols)
11. vis = [start]
12. queue = [start]
13. path = {}
14. **while** len(queue) > 0:
15. cur = queue.pop(0)
16. **if** cur == destination:
17. **break**
18. **for** d **in** dir:
19. **if** m.maze\_map[cur][d]:
20. **if** d == 'E':
21. Next = (cur[0], cur[1] + 1)
22. **elif** d == 'S':
23. Next = (cur[0] + 1, cur[1])
24. **elif** d == 'W':
25. Next = (cur[0], cur[1] - 1)
26. **elif** d == 'N':
27. Next = (cur[0] - 1, cur[1])
28. **if** Next **in** vis:
29. **continue**
30. vis.append(Next)
31. queue.append(Next)
32. path[Next] = cur
33. bfs\_path = {}
34. cur = (1, 1)
35. **while** cur != start:
36. bfs\_path[path[cur]] = cur
37. cur = path[cur]
38. ed = time.time()
39. **return** bfs\_path, ed - st

42. **if** \_\_name\_\_ == '\_\_main\_\_':
43. m = maze(30, 30)
44. m.CreateMaze(loopPercent=50, theme=COLOR.light, saveMaze=True)
45. path, spent\_time = BFS(m)
46. **print**(path)
47. a = agent(m, footprints=True, filled=True)
48. m.tracePath({a: path})
49. l = textLabel(m, 'Length of the Path', len(path) + 1)
50. l = textLabel(m, 'Spent time', spent\_time)
51. m.run()
52. DFS
53. **from** pyamaze **import** maze, agent, textLabel, COLOR
54. **import** time
56. **def** DFS(m):
57. st = time.time()
58. dir = 'ESWN'
59. destination = (1, 1)
60. start = (m.rows, m.cols)
61. vis = [start]
62. queue = [start]
63. path = {}
64. **while** len(queue) > 0:
65. cur = queue.pop()
66. **if** cur == destination:
67. **break**
68. **for** d **in** dir:
69. **if** m.maze\_map[cur][d]:
70. **if** d == 'E':
71. Next = (cur[0], cur[1] + 1)
72. **elif** d == 'S':
73. Next = (cur[0] + 1, cur[1])
74. **elif** d == 'W':
75. Next = (cur[0], cur[1] - 1)
76. **elif** d == 'N':
77. Next = (cur[0] - 1, cur[1])
78. **if** Next **in** vis:
79. **continue**
80. vis.append(Next)
81. queue.append(Next)
82. path[Next] = cur
83. dfs\_path = {}
84. cur = (1, 1)
85. **while** cur != start:
86. dfs\_path[path[cur]] = cur
87. cur = path[cur]
88. ed = time.time()
89. **print**(st, ed)
90. **return** dfs\_path, ed - st

93. **if** \_\_name\_\_ == '\_\_main\_\_':
94. m = maze(20, 20)
95. m.CreateMaze(loopPercent=40, theme=COLOR.light, loadMaze='maze3.csv')
96. path, spent\_time = DFS(m)
97. a = agent(m, footprints=True, filled=True)
98. m.tracePath({a: path})
99. l = textLabel(m, 'Length of Shortest Path', len(path) + 1)
100. l = textLabel(m, 'spent\_time', spent\_time)
101. m.run()
102. A\*
103. **import** math
105. **from** pyamaze **import** maze, agent, textLabel, COLOR
106. **from** queue **import** PriorityQueue
107. **import** time
109. **def** h1(cur, goal):
110. **return** math.sqrt((cur[0]-goal[0])\*\*2 + (cur[1]-goal[1])\*\*2)

113. **def** h(cur, goal):
114. **return** abs(cur[0] - goal[0]) + abs(cur[1] - goal[1])

117. **def** aStar(m):
118. st = time.time()
119. dir = 'ESWN'
120. start = (m.rows, m.cols)
121. # The cost path from the start node to the current node
122. g\_score = {}
123. # The cost path from the current node to goal node
124. f\_score = {}
125. **for** cur **in** m.grid:
126. g\_score[cur] = float('inf')
127. **for** cur **in** m.grid:
128. f\_score[cur] = float('inf')
129. g\_score[start] = 0
130. f\_score[start] = h(start, (1, 1))
131. pq = PriorityQueue()
132. pq.put((h(start, (1, 1)), h(start, (1, 1)), start))
133. path = {}
134. **while** **not** pq.empty():
135. cur = pq.get()[2]
136. **if** cur == (1, 1):
137. **break**
138. **for** d **in** dir:
139. **if** m.maze\_map[cur][d]:
140. **if** d == 'E':
141. Next = (cur[0], cur[1] + 1)
142. **elif** d == 'S':
143. Next = (cur[0] + 1, cur[1])
144. **elif** d == 'W':
145. Next = (cur[0], cur[1] - 1)
146. **elif** d == 'N':
147. Next = (cur[0] - 1, cur[1])
148. next\_g = g\_score[cur] + 1
149. next\_f = next\_g + h(Next, (1, 1))
150. **if** next\_f < f\_score[Next]:
151. g\_score[Next] = next\_g
152. f\_score[Next] = next\_f
153. pq.put((next\_f, h(Next, (1, 1)), Next))
154. path[Next] = cur
155. astar\_path = {}
156. cur = (1, 1)
157. **while** cur != start:
158. astar\_path[path[cur]] = cur
159. cur = path[cur]
160. ed = time.time()
161. **return** astar\_path, ed - st

164. **if** \_\_name\_\_ == '\_\_main\_\_':
165. m = maze(30, 40)
166. m.CreateMaze(loadMaze='maze3.csv',theme=COLOR.light)
167. path, spent\_time = aStar(m)
168. **print**(path)
169. a = agent(m, footprints=True, filled=True)
170. m.tracePath({a: path})
171. l = textLabel(m, 'A Star Path Length', len(path) + 1)
172. l = textLabel(m, 'Spent time', spent\_time)
173. m.run()
174. MDP Value Iteration
175. **import** numpy as np
176. **import** time
177. **from** pyamaze **import** maze, agent, textLabel, COLOR

180. **def** MDP\_value(m):
181. st = time.time()
182. actions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
184. dir = 'ESWN'
185. V = np.zeros((m.rows, m.cols))
186. **print**(m.rows, m.cols)
187. probabilities = np.zeros((m.rows, m.cols, 4, m.rows, m.cols))
188. rewards = np.zeros((m.rows, m.cols, 4, m.rows, m.cols))
189. gamma = 0.9
190. theta = 0.001
191. **for** i **in** range(m.rows):
192. **for** j **in** range(m.cols):
193. **for** u, d **in** enumerate(dir):
194. **if** m.maze\_map[(i+1, j+1)][d]:
195. next\_i = i + actions[u][0]
196. next\_j = j + actions[u][1]
197. probabilities[i, j, u, next\_i, next\_j] = 1
198. rewards[i, j, u, next\_i, next\_j] = 1 **if** (next\_i, next\_j) == (m.rows-1, m.cols-1) **else** 0
199. **else**:
200. probabilities[i, j, u, i, j] = 1
201. **while** True:
202. delta = 0
203. **for** i **in** range(m.rows):
204. **for** j **in** range(m.cols):
205. v = V[i, j]
206. qs = np.zeros(4)
207. **for** u, action **in** enumerate(actions):
208. qs[u] = np.sum(probabilities[i, j, u] \* (rewards[i, j, u] + gamma \* V))
209. V[i, j] = np.max(qs)
210. delta = max(delta, abs(v - V[i, j]))
211. **if** delta < theta:
212. **break**
214. policy = np.zeros((m.rows, m.cols))
215. **for** i **in** range(m.rows):
216. **for** j **in** range(m.cols):
217. q\_vals = []
218. **for** idx, action **in** enumerate(actions):
219. q = np.sum(probabilities[i, j, idx, :, :] \* (rewards[i, j, idx, :, :] + gamma \* V))
220. q\_vals.append(q)
221. policy[i, j] = np.argmax(q\_vals)
223. path = [(1, 1)]
224. i, j = 0, 0
225. **while** (i, j) != (m.rows-1, m.cols-1):
226. idx = int(policy[i, j])
227. i += actions[idx][0]
228. j += actions[idx][1]
229. path.append((i+1, j+1))
231. mdpValue\_path = {}
232. **for** i **in** range(0, len(path) - 1):
233. mdpValue\_path[path[i + 1]] = path[i]
234. ed = time.time()
235. **return** mdpValue\_path, ed - st

238. **if** \_\_name\_\_ == '\_\_main\_\_':
239. m = maze(20, 20)
240. m.CreateMaze(loadMaze='maze1.csv', theme=COLOR.light)
241. path, spent\_time = MDP\_value(m)
242. a = agent(m, footprints=True, filled=True)
243. m.tracePath({a: path})
244. l = textLabel(m, 'Length of Shortest Path', len(path) + 1)
245. l = textLabel(m, 'Spent time', spent\_time)
246. m.run()
247. MDP Policy Iteration
248. **from** pyamaze **import** maze, agent, textLabel, COLOR
249. **import** numpy as np
250. **import** time

253. **def** mdp\_policy(m):
254. st = time.time()
255. dir = 'ESWN'
256. n\_actions = 4
257. size = max(m.cols, m.rows)
258. n\_states = size\*\*2
259. actions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
260. gamma = 0.8
261. theta = 0.001
262. probabilities = np.zeros((n\_states, n\_actions, n\_states))
263. rewards = np.zeros((n\_states, n\_actions, n\_states))
264. **for** i **in** range(m.rows):
265. **for** j **in** range(m.cols):
266. **for** u, d **in** enumerate(dir):
267. **if** m.maze\_map[(i + 1, j + 1)][d]:
268. next\_i = i + actions[u][0]
269. next\_j = j + actions[u][1]
270. rewards[i\*size+j, u, next\_i\*size+next\_j] = 1 **if** (next\_i, next\_j) == (m.rows - 1, m.cols - 1) **else** 0
271. **else**:
272. **if** m.rows > i + actions[u][0] >= 0 **and** m.cols > j + actions[u][1] >= 0:
273. rewards[i\*size+j, u, next\_i\*size+next\_j] = -10
274. **for** s **in** range(n\_states):
275. probabilities[s, 0, s - size **if** s - size >= 0 **else** s] = 1.0  # Up
276. probabilities[s, 1, s + size **if** s + size < n\_states **else** s] = 1.0  # Down
277. probabilities[s, 2, s - 1 **if** s % size != 0 **else** s] = 1.0  # Left
278. probabilities[s, 3, s + 1 **if** (s + 1) % size != 0 **else** s] = 1.0  # Right
279. policy = np.zeros(n\_states).astype('int')
280. Value = np.zeros(n\_states).astype('float')
281. is\_convergence = False
283. **while** **not** is\_convergence:
284. **while** True:
285. delta = 0
286. **for** s **in** range(n\_states):
287. v = Value[s]
288. Value[s] = np.sum(probabilities[s, policy[s]] \* (rewards + gamma \* Value))
289. **print**(Value[s])
290. delta = max(delta, np.abs(v - Value[s]))
291. **if** delta < theta:
292. **break**
293. policy\_convergence = True
294. **for** s **in** range(n\_states):
295. old\_action = policy[s]
296. q\_values = np.zeros(n\_actions)
297. **for** a **in** range(n\_actions):
298. q\_values[a] = np.sum(probabilities[s, a] \* (rewards + gamma \* Value))
299. policy[s] = np.argmax(q\_values)
300. **if** old\_action != policy[s]:
301. policy\_convergence = False
302. **if** policy\_convergence:
303. is\_convergence = True
305. **for** i **in** range(m.rows):
306. **for** j **in** range(m.cols):
307. **if** policy[i][j] == 0:
308. **print**(dir[3], end=' ')
309. **elif** policy[i][j] == 1:
310. **print**(dir[1], end=' ')
311. **elif** policy[i][j] == 2:
312. **print**(dir[2], end=' ')
313. **elif** policy[i][j] == 3:
314. **print**(dir[0], end=' ')
315. **print**("")
317. path = [(1, 1)]
318. i, j = 0, 0
319. **while** (i < m.rows - 1) **and** (j < m.cols - 1):
320. idx = int(policy[i, j])
321. i += actions[idx][0]
322. j += actions[idx][1]
323. path.append((i + 1, j + 1))
325. mdpPolicy\_path = {}
326. **for** i **in** range(0, len(path) - 1):
327. mdpPolicy\_path[path[i + 1]] = path[i]
328. ed = time.time()
329. **return** mdpPolicy\_path, ed - st

332. **if** \_\_name\_\_ == '\_\_main\_\_':
333. m = maze(10, 10)
334. m.CreateMaze(loadMaze='maze1.csv', theme=COLOR.light)
335. path, spent\_time = mdp\_policy(m)
336. a = agent(m, footprints=True, filled=True)
337. m.tracePath({a: path})
338. l = textLabel(m, 'Length of Shortest Path', len(path) + 1)
339. l = textLabel(m, 'Spent time', spent\_time)
340. m.run()