기초 인공지능 : Assignment01(BFS, DFS, A* Algorithm을 활용한 최단 경로 찾기)

전공: 컴퓨터공학 학년: 3학년 학번: 20201635 이름: 전찬

0. 목차

1. 구현 목표

이번 과제에서는, BFS, DFS, A* Algorithm을 활용하여 주어진 미로에서 최적의 결과 path을 알아내야 한다. 이를 위해 주어진 maze object에 대해 적절하게 path finding을 수행할 수 있는 BFS, DFS, A* Algorithm을 구현해야 한다.

2. 사용한 라이브러리와 각 알고리즘마다 구현한 방법

이번 과제에서 추가적으로 사용한 라이브러리는 다음과 같다.

- from collections import deque : python에서 list 형태로 queue를 구현했을 때 발생할 수 있는 오버헤드를 줄이기 위한 deque 이다. BFS 방법을 구현할 때 사용한다.
- from queue import PriorityQueue : python에서 priority queue를 사용하기 위한 라이브러리이다. A* Algorithm의 구현에서 f value 기준으로 정렬된 queue를 위해 사용한다.

또한 각 알고리즘마다 구현한 방법은 아래와 같다.

- path finding

모든 Algorithm에서 동일한 형태로 path finding을 구현했다. parent_node 라는 dictinary를 통해 현재 Position : 이전 Position을 저장했다. 이 dictionary를 이용해 path를 간단하게 파악할 수 있다.

- BFS method

BFS 알고리즘은 queue에서 가장 앞에 있는 position(node)을 pop 하며 해당 position이 목적지이면 path를 return 하며, 그렇지 않으면 maze.neighborPoints() method을 통해 주변에 이동할 수 있는 position을 파악하고, 해당 position에 이전에 방문한 적이 없다면, 해당 position을 queue에 넣는 과정을 반복한다. 이때 해당 position을 이전에 방문했다는 것은, 해당 position까지 도달하는 데에 더 짧은 길이를 갖는 path가 존재하는 것과 동일하다.

- DFS method

DFS 알고리즘은 stack을 활용하여 가장 최근에 삽입된 position을 기준으로 목적지인지 확인하며, 목적지가 아니라면 maze.neighborPoints() method을 통해 주변 position을 주변에 쌓는 형식으로 수행한다. 이 경우에도 해당 주변 position을 이미 방문한 경우, 해당 주변 position은 stack에 넣지 않아도 된다. DFS method의 경우에는 항상 최적(가장 짧은 path)을 보장하지는 않는다.

- A* method

A* 알고리즘은 g value, h value을 바탕으로 계산한 f value을 기준으로 position을 priority queue에 정렬하는 형태로 구현했다. 목적지인지를 확인하며, 목적지가 아닌 경우, maze.neighborPoints() method로 주변 position을 파악하며, 주변 position이 이전에 방문하지 않았거나, 방문했지만 구한 주변 position의 f value 값보다 큰 값을 갖고 있었던 경우에는 f value를 update 해주며 priority queue 에 삽입하는 과정을 반복한다.

- 여러 목적지(circlePoint)를 지나는 A* method

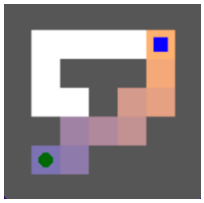
여러 목적지를 지나는 A* method는 하나의 목적지를 지나는 A* Algorithm을 활용해서 구현할 수 있다. 하나의 목적지를 지나는 A* Algorithm에서 현재 state는 position이라고 할 수 있다. 하지만 여러 목적지를 지나는 A* Algorithm은 현재 state를 (position, 각 목적지 도달 정보) 의 tuple로 정의한다. 예를 들어, 1~4의 목적지 중 1, 3 번째 목적지에 이미 도달한 경우, 각 목적지 도달 정보 = (1, 0, 1, 0) 와 같이 표현할 수 있다. 이 이외에는 위에서 구현한 하나의 목적지를 지나는 A* algorithm과 동일하게 작동하며, 목적지를 확인하는 부분도 단순히 position으로 판별하는 것이 아닌, position = 목적지, 도달 정보 = (1, 1, ….. , 1) 와 같이 두 정보로 확인해야 한다.

위와 같은 형태로 각각의 Algorithm을 구현할 수 있다. 추가로 DFS의 경우에도 IDS, 혹은 DFS에서 첫 번째 path 발견이 아닌, stack에 원소가 존재하지 않을 때까지 검색을 수행하며 최단 거리를 구하는 형태의 Algorithm 도 생각해 보았는데, 이 경우에 단순한 DFS보다 time / space complexity가 상당히 커질 것 같아 단순한 DFS 형태로 구현했다. 따라서 DFS Algorithm의 경우에는 maze path가 최단 거리임을 보장하지는 않는다. BFS, A* Algorithm에서는 구한 path가 최단 거리임을 보장한다.

3-1. BFS method 출력 화면

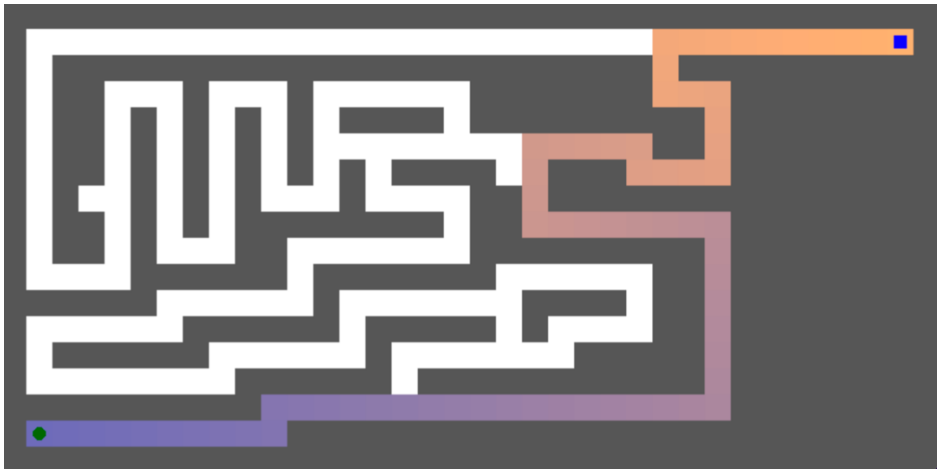
각각의 실행 화면과 결과는 아래와 같다.

-small.txt



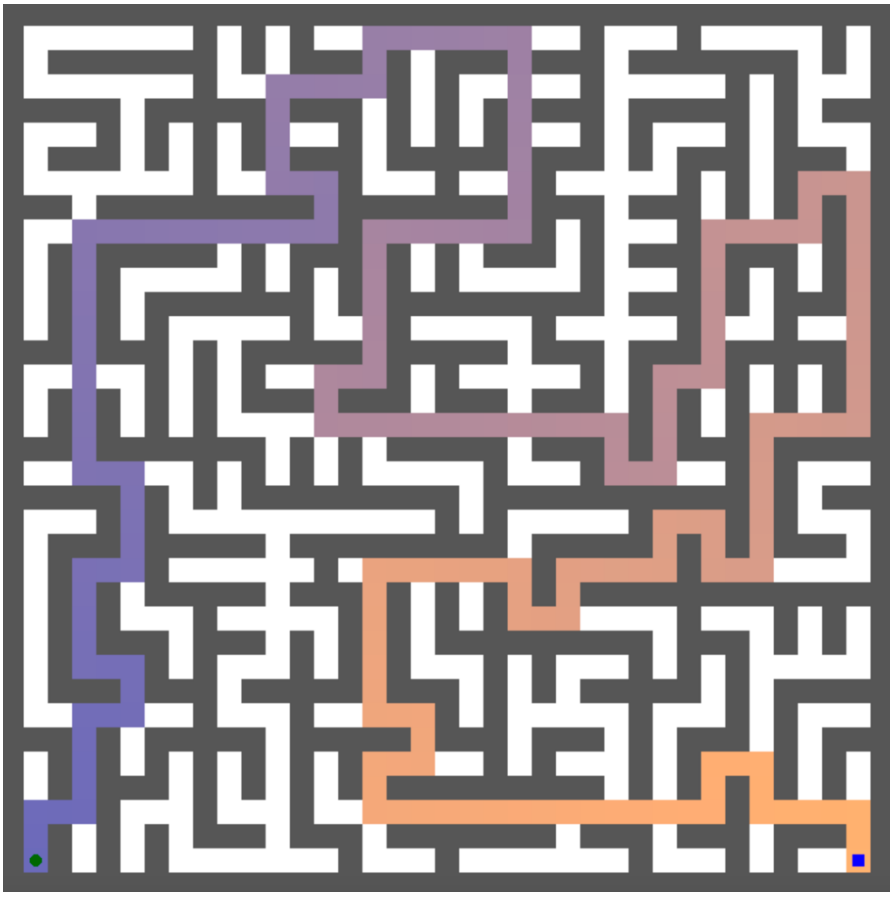
```
========================================
[ bfs results ]
(1) Path Length: 9
(2) Search States: 15
(3) Execute Time: 0.0000679493 seconds
========================================
```

-medium.txt



```
========================================
[ bfs results ]
(1) Path Length: 69
(2) Search States: 219
(3) Execute Time: 0.0008161068 seconds
========================================
```

-big.txt

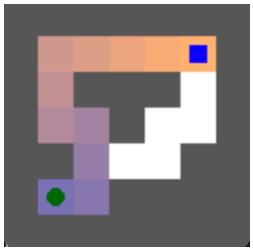


```
==========================================
[ bfs results ]
(1) Path Length: 211
(2) Search States: 619
(3) Execute Time: 0.0023000240 seconds
==========================================
```

3-2. DFS method 출력 화면

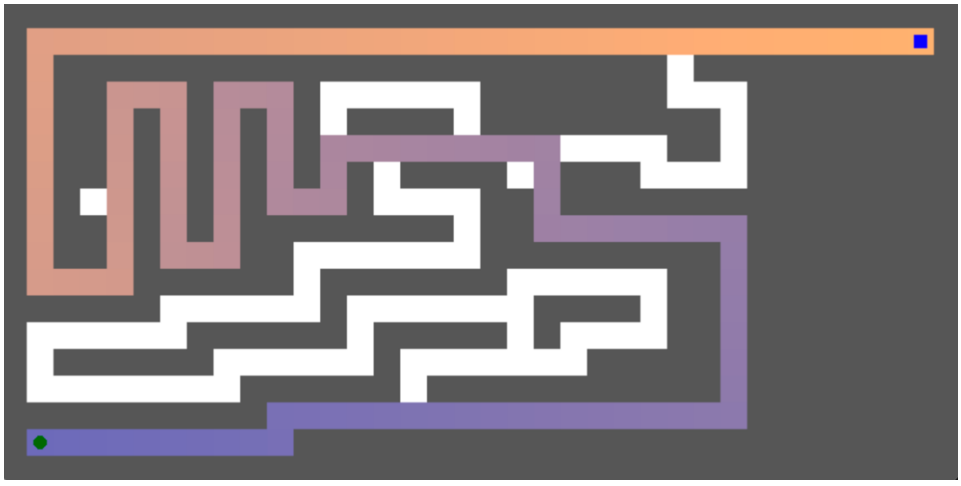
각각의 실행 화면과 결과는 아래와 같다.

-small.txt



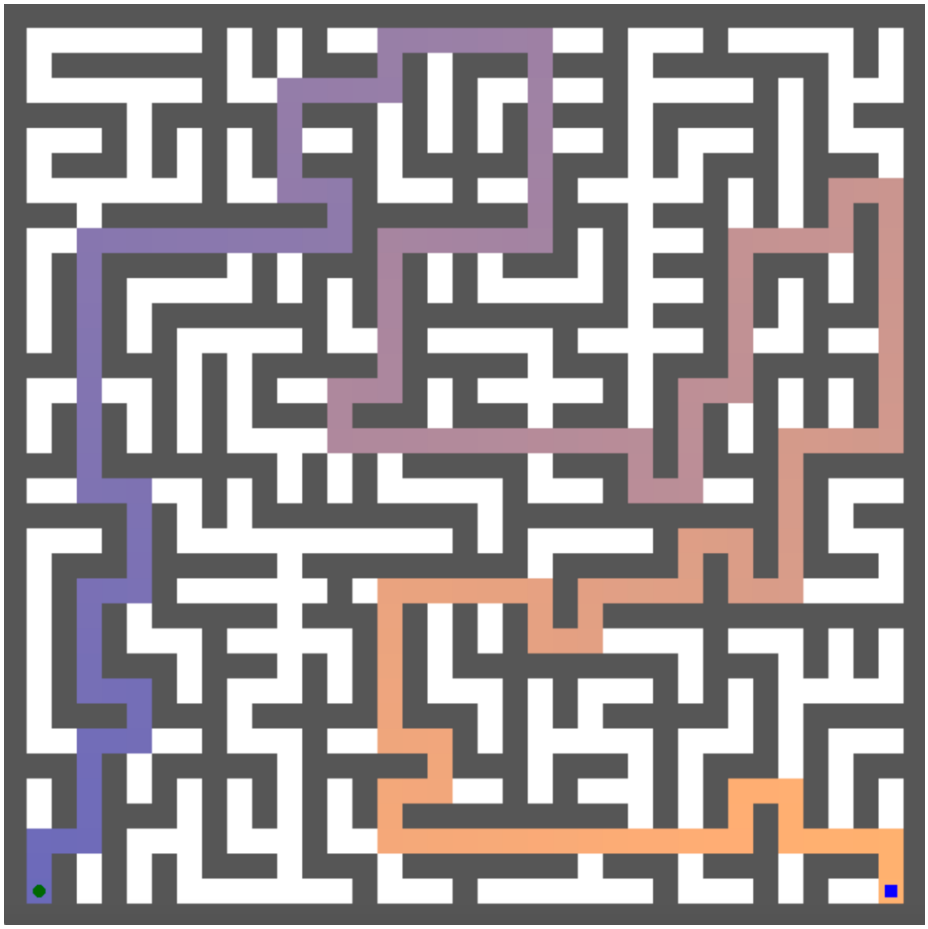
```
==========================================
[ dfs results ]
(1) Path Length: 11
(2) Search States: 14
(3) Execute Time: 0.0000729561 seconds
==========================================
```

-medium.txt



```
==========================================
[ dfs results ]
(1) Path Length: 131
(2) Search States: 144
(3) Execute Time: 0.0005819798 seconds
==========================================
```

-big.txt

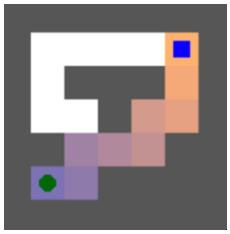


```
==========================================
[ dfs results ]
(1) Path Length: 211
(2) Search States: 426
(3) Execute Time: 0.0016226768 seconds
==========================================
```

3-3. A* method 출력 화면

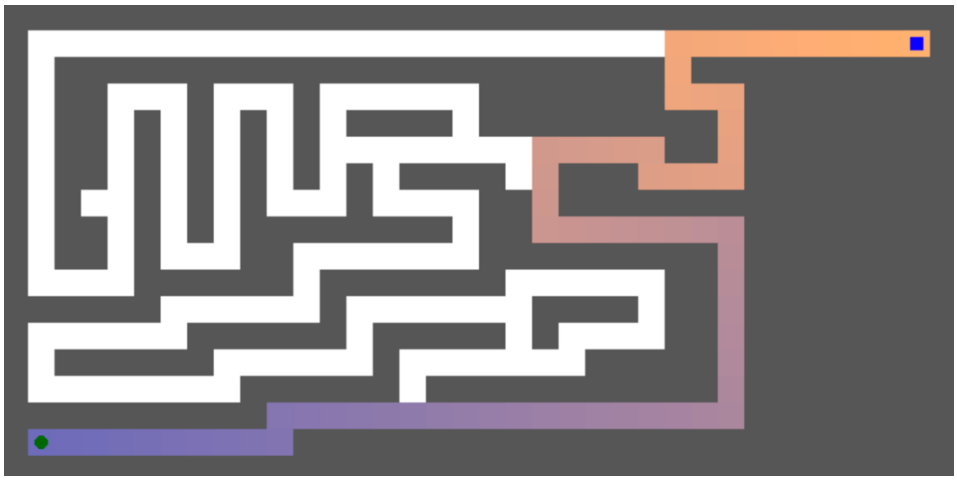
각각의 실행 화면과 결과는 아래와 같다.

-small.txt



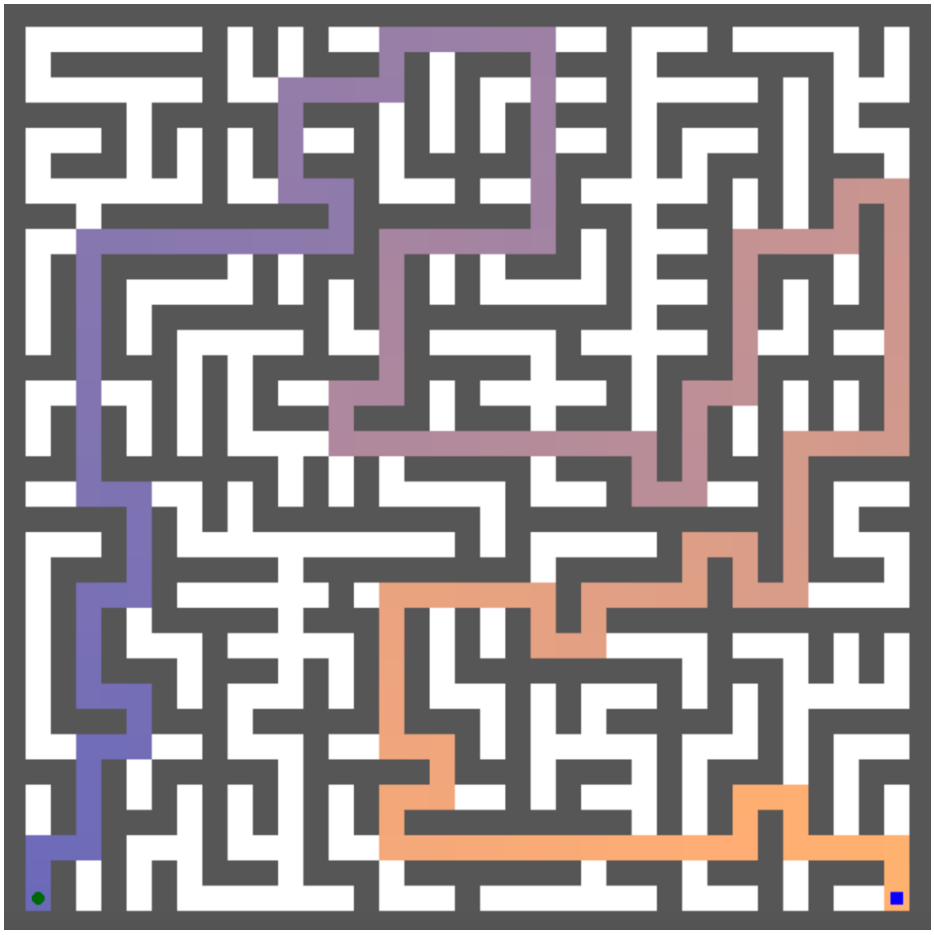
```
==========================================
[ astar results ]
(1) Path Length: 9
(2) Search States: 14
(3) Execute Time: 0.0003728867 seconds
==========================================
```

-medium.txt



```
==========================================
[ astar results ]
(1) Path Length: 69
(2) Search States: 183
(3) Execute Time: 0.0017409325 seconds
==========================================
```

-big.txt

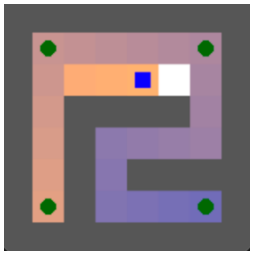


```
==========================================
[ astar results ]
(1) Path Length: 211
(2) Search States: 548
(3) Execute Time: 0.0049488544 seconds
==========================================
```

3-4. 4개의 목적지를 갖는 A* method 출력 화면

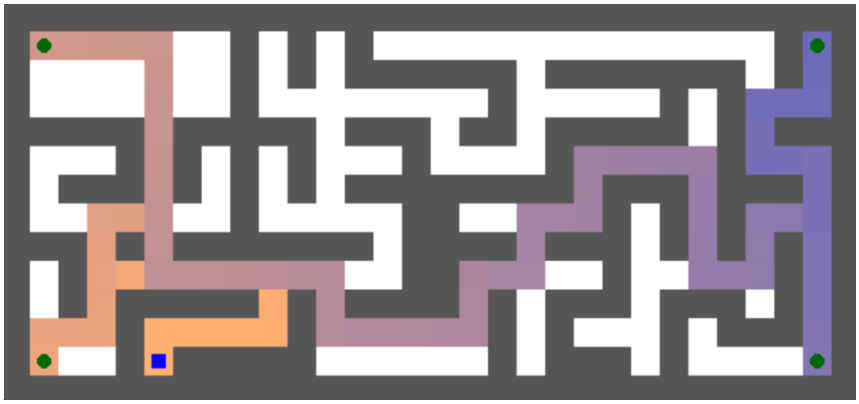
각각의 실행 화면과 결과는 아래와 같다.

-small.txt



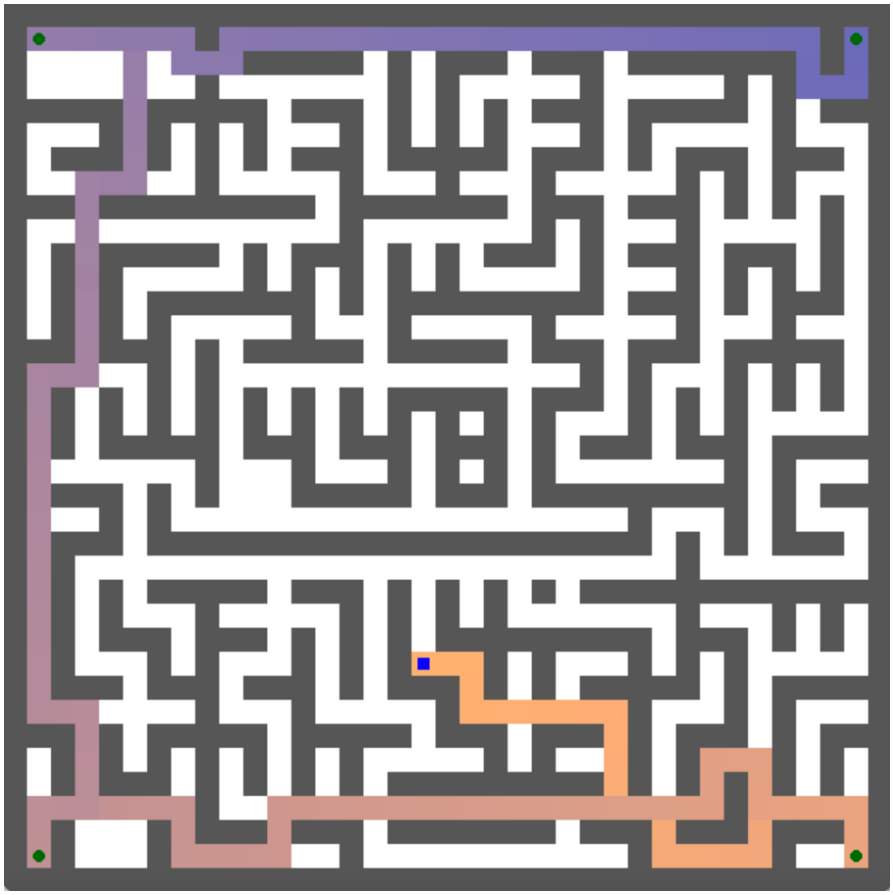
```
========================================
[ astar_four_circles results ]
(1) Path Length: 29
(2) Search States: 234
(3) Execute Time: 0.0047969818 seconds
========================================
```

-medium.txt



```
========================================
[ astar_four_circles results ]
(1) Path Length: 107
(2) Search States: 1546
(3) Execute Time: 0.0343170166 seconds
========================================
```

-big.txt

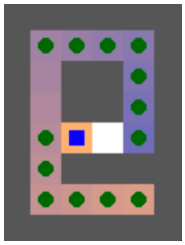


```
========================================
[ astar_four_circles results ]
(1) Path Length: 163
(2) Search States: 6090
(3) Execute Time: 0.1223440170 seconds
========================================
```

3-5. 여러 목적지를 갖는 A* method 출력 화면

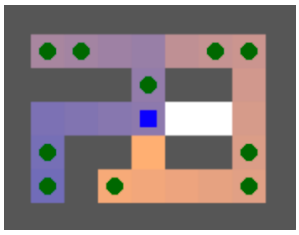
각각의 실행 화면과 결과는 아래와 같다.

-tiny.txt



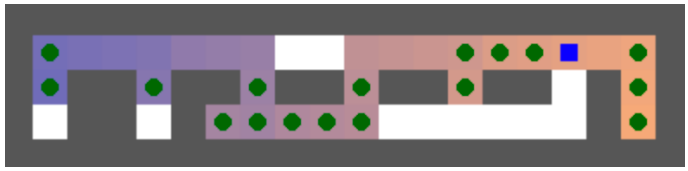
```
==========================================
[ astar_many_circles results ]
(1) Path Length: 21
(2) Search States: 722
(3) Execute Time: 0.0045819283 seconds
==========================================
```

-small.txt



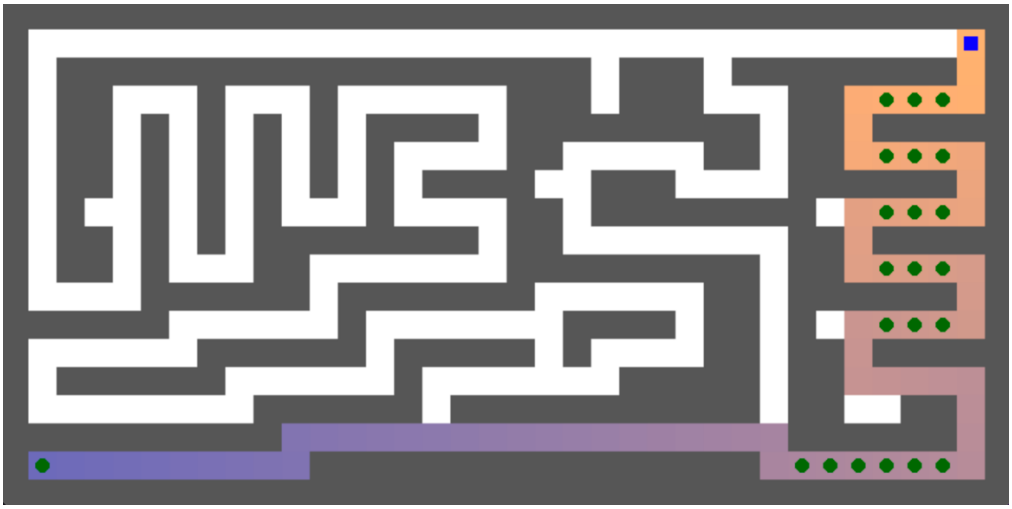
```
==========================================
[ astar_many_circles results ]
(1) Path Length: 28
(2) Search States: 719
(3) Execute Time: 0.0063638687 seconds
==========================================
```

-medium.txt



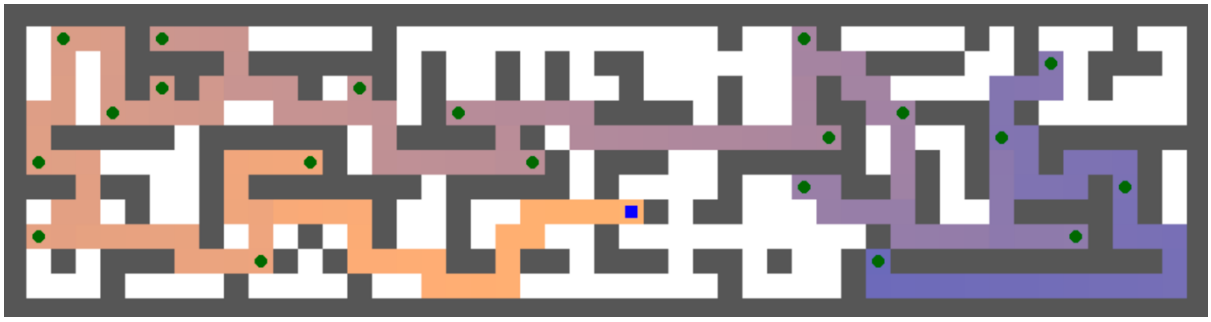
```
==========================================
[ astar_many_circles results ]
(1) Path Length: 35
(2) Search States: 2714
(3) Execute Time: 0.0127389431 seconds
==========================================
```

-many_obj.txt



```
========================================
[ astar_many_circles results ]
(1) Path Length: 75
(2) Search States: 11550
(3) Execute Time: 0.0520646572 seconds
========================================
```

-big.txt



```
========================================
[ astar_many_circles results ]
(1) Path Length: 199
(2) Search States: 46592
(3) Execute Time: 1.7873599529 seconds
========================================
```

4. Stage2와 Stage3에서 직접 정의한 Heuristic Function

Stage2와 Stage3에서는 각각 A* Algorithm을 통해 여러 목적지를 지나는 최단 거리를 구할 수 있어야 한다. 이를 위해 Heuristic function을 정의해야 한다.

Stage2의 Heuristic function은 비교적 간단하다. 2. 에서 설명한 도달한 목적지 tuple이 가질 수 있는 경우의 수가 2^4 = 16 가지로, 총 state가 최대로 많아도 한 목적지를 찾는 state의 16배라는 의미이다. 따라서 비교적 정확하지 않은, 간단한 Heuristic function을 사용하더라도 짧은 시간 내에 path을 얻어낼 수 있음을 파악할 수 있다. Stage2의 Heuristic function은 따라서 아직 도달하지 못한 목적지 중, 가장 가까운 목적지와 현재 position 사이의 manhattan distance을 사용하여 쉽게 표현할 수 있다. 구현은 아래와 같다.

```python
def stage2_heuristic(current_pos, destinations, dest_arrived):

    #  현재 위치에서 가장 가까운 목적지로 manhattan_dist 을 heuristic으로 이용한다.
    dest_left = [destinations[i] for i in range(4) if dest_arrived[i] == 0]

    if dest_left == []:
        return 0

    x1_diff = [abs(i[0] - current_pos[0]) for i in dest_left]
    x2_diff = [abs(i[1] - current_pos[1]) for i in dest_left]
    manhattan_value = min(x1_diff) + min(x2_diff)

    return manhattan_value
```

위 함수에서 current_pos는 h value를 계산하는 현재 position, destinations는 4개의 목적지 list, dest_arrived 는 위에서 이야기한 (1, 0, 1, 0) 와 같은 tuple을 parameter로 가진다.

Stage3에서는 도달한 목적지의 수가 정해져 있지 않다. 또한 n개의 목적지인 경우, 목적지 tuple이 가질 수 있는 경우의 수는 2^n 이라고 할 수 있다. 따라서 총 state 또한 기하급수적으로 늘어난다. 따라서 Stage2에서 구현한 Heuristic function 보다는 더욱 정밀한 함수가 필요하다. 이를 위해 사용할 수 있는 방법이 mst(Minimum Spanning Tree)로, 각 node를 모두 연결하는 최단 거리를 사용할 수 있다. 이러한 mst는 Kruskal, 혹은 Prim algorithm을 통해 구할 수 있는데, 과제에서는 Kruskal algorithm을 통해 mst의 값을 계산했다. 또한 Stage2에서 활용한 heuristic의 값 또한 활용했다. 따라서 Heuristic function은 mst(도달하지 않은 목적지) + manhattan distance 와 같이 구현했다.

또한 Kruskal Algorithm을 이용하여 mst 값을 구하기 위해, 모든 목적지가 각각의 node로 존재하는 graph 형태에서 각각의 목적지 사이의 edge 값 또한 파악해야 하며, 이는 2중 loop로 각각 i, j 값에 대해 destinations[i] -> destinations[j] 을 위에서 구현한 BFS Algorithm을 통해 edge 값을 구했다. 이렇게 계산한 값을 parameter로 전달해 주며 Stage3에서의 Heuristic function을 구현해 낼 수 있었다. 이를 토대로 구현한 함수는 아래와 같다.

```python
def stage3_heuristic(current_pos, destinations, dest_arrived, sorted_edge):

    # 2 가지의 value 를 이용한다.

    # 첫 번째는 현재 위치에서 가장 가까운 목적지로 manhattan_dist
    # 두 번째는 MST value, 즉 모든 남은 목적지를 연결하기 위한 value 를 사용한다.
    # 구한 두 값을 더한 값을 stage3_heuristic value 로 설정한다.
    dest_left = [destinations[i] for i in range(len(destinations)) if dest_arrived[i] == 0]
    if dest_left == []:
        return 0

    x1_diff = [abs(i[0] - current_pos[0]) for i in dest_left]
    x2_diff = [abs(i[1] - current_pos[1]) for i in dest_left]

    manhattan_value = min(x1_diff) + min(x2_diff)

    mst_value = mst(dest_arrived, sorted_edge)

    return manhattan_value + mst_value
```

위 함수에서 current_pos는 현재 position, destinations는 목적지 list, dest_arrived는 Stage2와 동일한 tuple 이다. 또한 sorted_edge는 astar_many_circles에서 구한 각 목적지간의 거리 data가 정렬된 값을 parameter로 받는 형태이다.