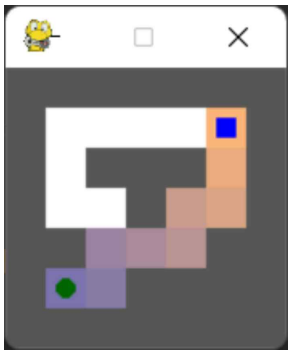
기초인공지능 과제 1번 보고서

20181650 안도현

1. BFS with stage 1

–가장 일반적인 bfs 구현을 사용했다. python이 제공하는 queue 라이브러리의 Queue 클래스를 사용하여 open list를 구현했고, 2차원 리스트 v를 통해 방문 여부를 체크하며 graph search로 진행하였다. 그래프 서치이므로 같은 노드에 재방문을 하지 않기 때문에 각 노드마다 부모를 기억하도록 하기 위해 2차원 리스트 prev 또한 정의하였다. 큐에서 뺀 노드의 주변 노드에 대해 방문하지 않은 노드를 방문 체크와 함께 큐에 집어넣었다. 그리고 목표 지점 도달 시 while 루프 내에서 prev를 반복적으로 따라가 path를 완성했다.

1) small

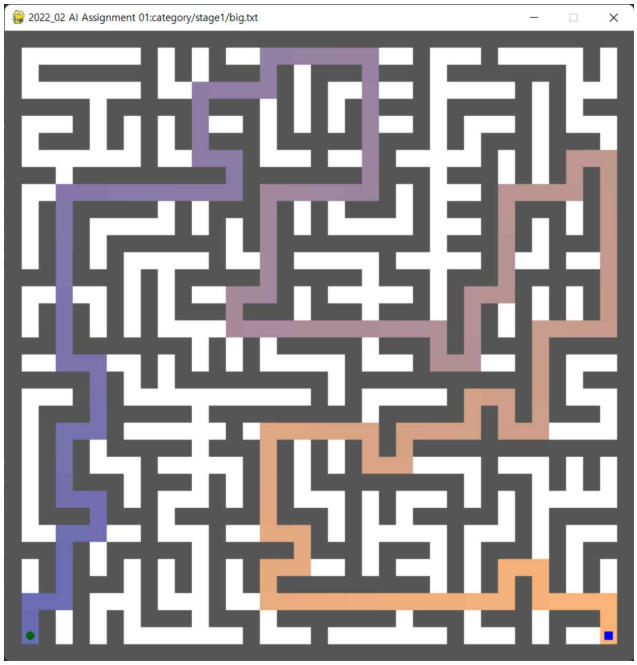


```
==========================================
[ bfs results ]
(1) Path Length: 9
(2) Search States: 16
(3) Execute Time: 0.0009994507 seconds
==========================================
```

2) medium


```
==========================================
[ bfs results ]
(1) Path Length: 69
(2) Search States: 220
(3) Execute Time: 0.0011415482 seconds
==========================================
```

3) big

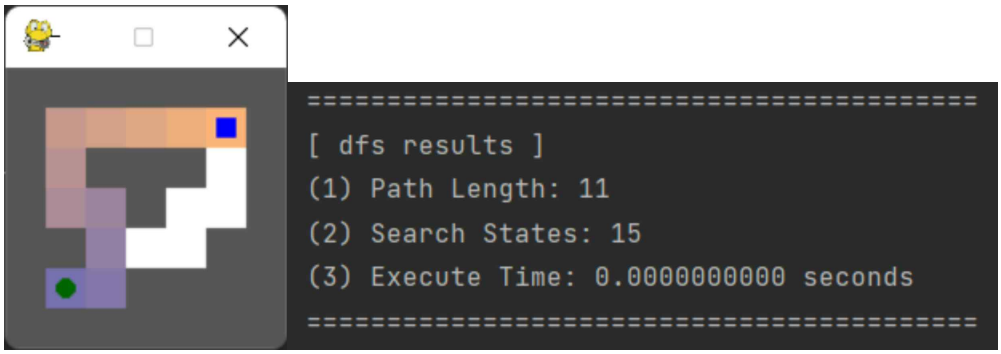



```
==========================================
[ bfs results ]
(1) Path Length: 211
(2) Search States: 620
(3) Execute Time: 0.0040125847 seconds
==========================================
```

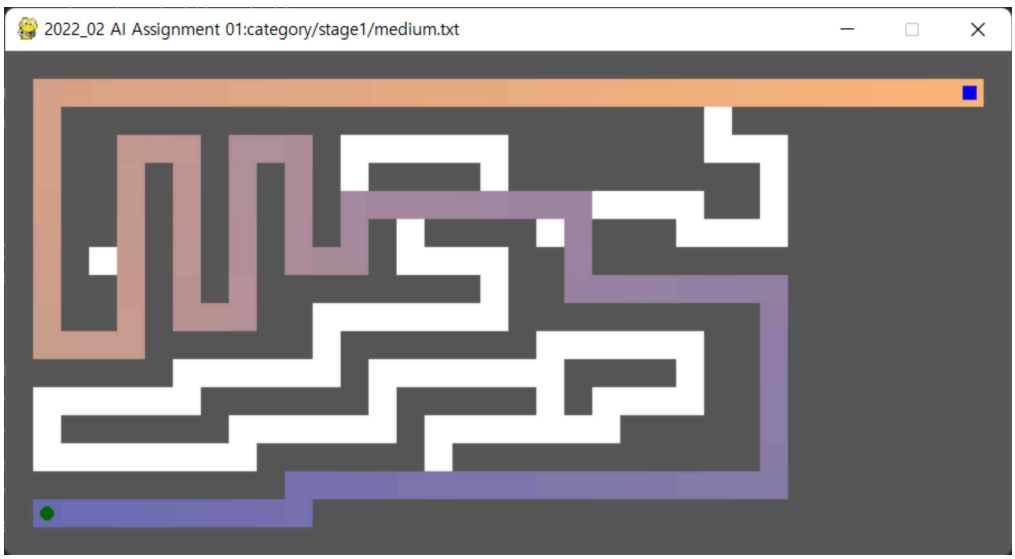
2. DFS with stage 1

-DFS는 코드 자체만 볼 때 BFS와 유사하다. 다만 Open list로 사용하던 queue 대신 stack의 성질을 갖는 open list를 도입하기 위해 list를 선언하고 append와 pop(-1)을 이용해 끝에서 노드를 넣고 빼는 식으로 코드를 작성했다. 처음에는 방문한 노드라도 이전에 방문했을 때 누적 코스트가 크다면 다시 방문이 가능하도록 해 최단 경로를 탐색하도록 구현했는데, 교수님께서 그래프 서치로 구현하라고 하셔서 최단 경로를 보장하지 못하는 대신 더 빠른 탐색이 가능한 그래프 서치로 구현했다. 다만 여기서는 bfs와 달리 방문 여부를 boolean형 리스트가 아니라 탐색을 하며 누적 계산된 코스트를 저장하는 식으로 하고 이 값이 -1이면 아직 방문하지 않은 것으로 간주했다. 이렇게 한 이유는 목적지에 도달하는 순간 탐색을 종료하는게 아니라 완전히 다른 경로지만 더 적은 코스트로 도착할 시 그 경로를 채택하기 위함이다.(물론 이렇게 해도 최적은 보장하지 못한다)

1) small

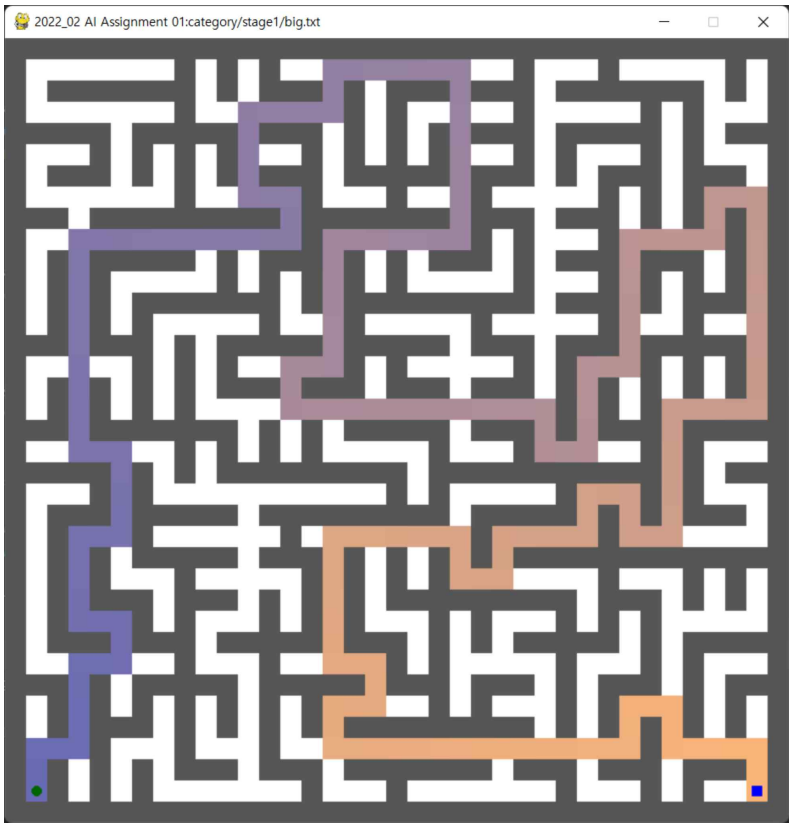


2) medium



```
==========================================
[ dfs results ]
(1) Path Length: 131
(2) Search States: 223
(3) Execute Time: 0.0010168552 seconds
==========================================
```

3) big

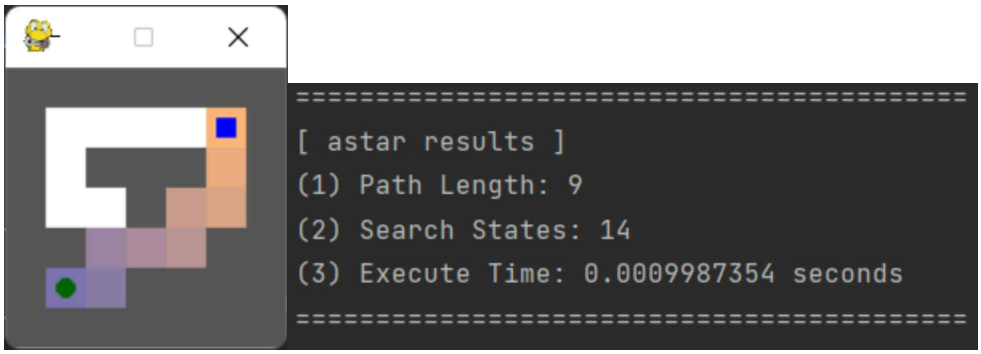



```
==========================================
[ dfs results ]
(1) Path Length: 211
(2) Search States: 646
(3) Execute Time: 0.0021295547 seconds
==========================================
```

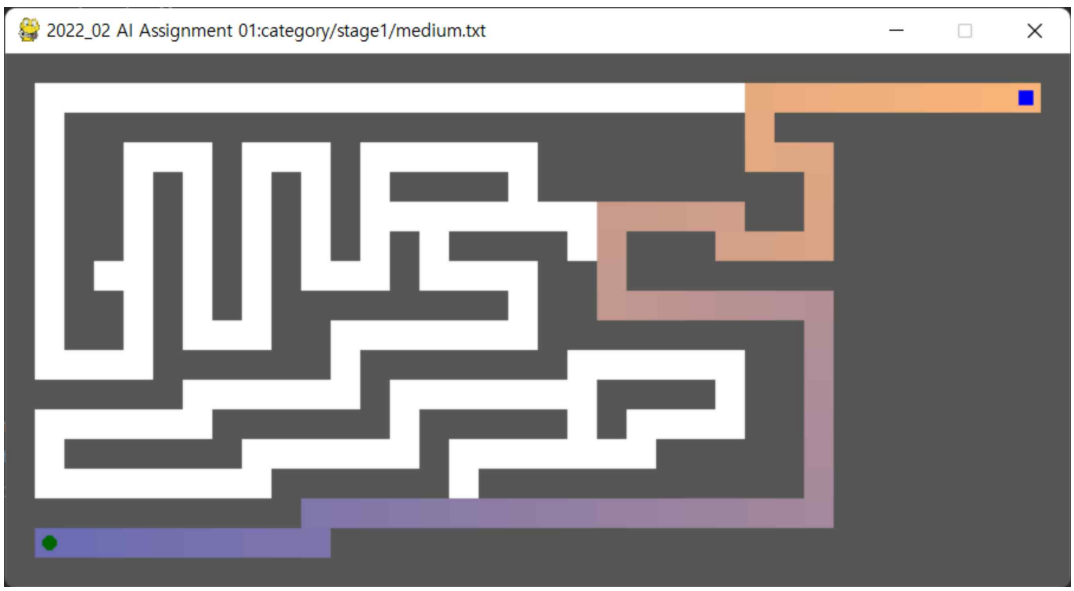
3. A* with stage 1

-휴리스틱 함수로 맨해튼 거리를 사용한 에이스타 알고리즘이다. 에이스타 알고리즘은 오픈 리스트에 들어있는 노드 중 f(=g+h)값이 가장 작은 노드부터 뽑아 탐색하는 알고리즘이므로 오픈 리스트를 python의 queue가 제공하는 PriorityQueue 클래스를 사용해 관리했다. stage1은 목적지가 하나이므로 특별히 트리 서치를 사용하지는 않았다. 맨해튼 거리가 admissible한 휴리스틱이므로 최적은 보장되겠지만 시간이 너무 많이 걸릴 것으로 예상되기 때문이었다. 따라서 BFS와 유사하게 방문 노드를 저장할 v 리스트를 사용했고, DFS와 유사하게 이웃 노드를 오픈 리스트에 추가할 때 반영되는 g값을 1씩 갱신했으며, 루프를 진행할 때 g 값 자체는 어디에 저장하지 않고 부모 노드의 f값에서 h값을 빼는 식으로 얻었다. 물론 그래프 서치로 최적 경로가 보장되는 건 휴리스틱이 consistent할 때인데, g값이 1 늘어날 때 h값이 1보다 더 작아질 수는 없으므로 f 값은 감소하지 않는 것으로 이해할 수 있었다.

1)small

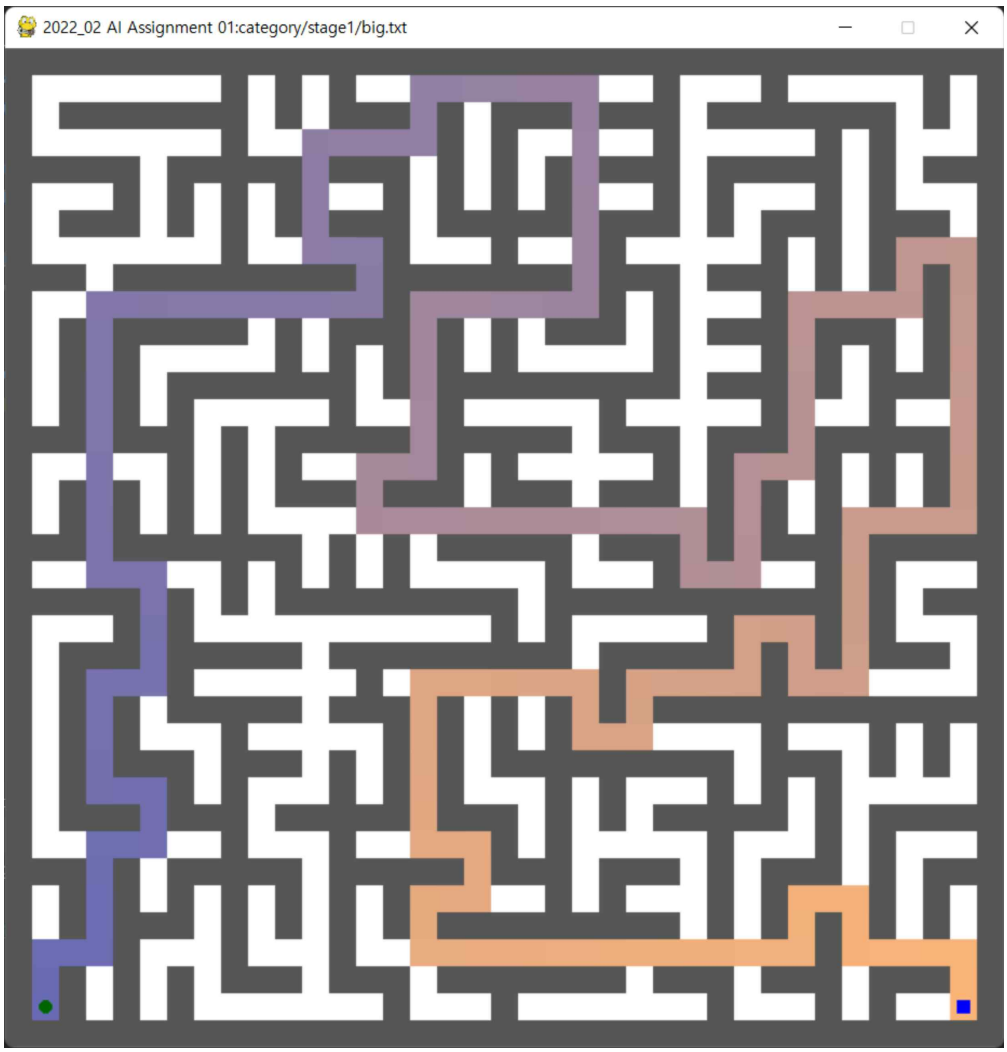


2)medium



```
==========================================
[ astar results ]
(1) Path Length: 69
(2) Search States: 182
(3) Execute Time: 0.0010006428 seconds
==========================================
```

3) big




```
==========================================
[ astar results ]
(1) Path Length: 211
(2) Search States: 548
(3) Execute Time: 0.0043013096 seconds
==========================================
```

4. A* with stage 2

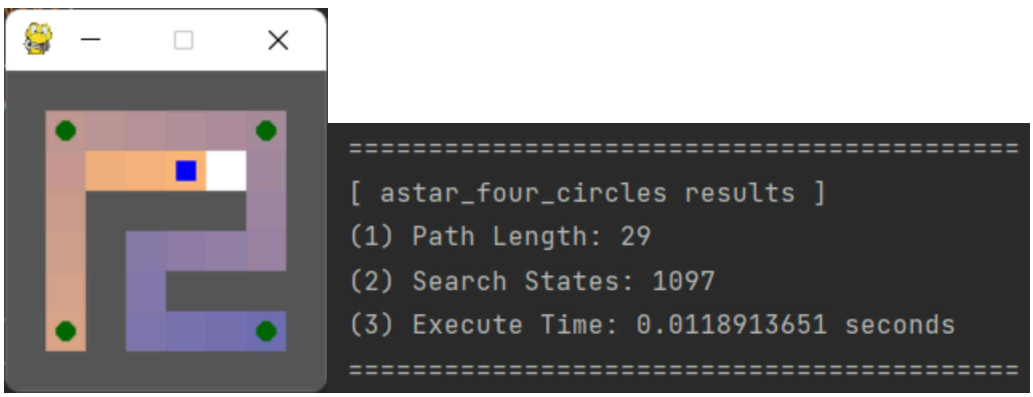
-stage 2부터는 코드가 길어져서 단순히 노드를 좌표만 사용해서 접근하는 것보다 클래스를 별도로 만들어 사용했다. 멤버 변수로 y,x,prev(부모),f,g,h가 있고, PriorityQueue에서 우선순위를 판단하기 위한 대소 비교 함수도 오버로딩했다.

-stage2는 목표 지점이 4개로 제한된다는 점에 착안하여 그냥 itertools의 permutation을 사용해 어느 순서로 목표 지점을 방문하는지에 따라 도출되는 경로의 길이를 모두 구해 최솟값을 취하는 식으로 코드를 작성했다. 분명 시간이 조금 더 걸리긴 하지만 가장 확실한 방법이다. Maze 클래스에 정의되어 있는 circlePoints()메소드로 목적지 리스트를 받아와 그걸 순서를 바꿔가며 같은 플로우를 반복하는 형식이다. 그리고 tmppath에 저장된 경로의 길이가 min_path_len보다 작다면 min_path_len을 갱신하고 path를 tmppath로 assign한다. 이런 식으로 탐색 종료시 최단거리를 갖게 된다.

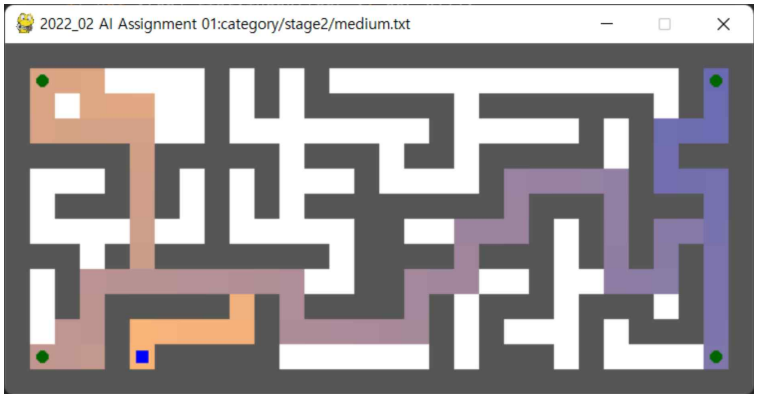
-휴리스틱은 현재 목적지 리스트에 저장된 목적지 순서대로, 그 목적지가 아직 방문하지 않은 상태면 거기까지의 맨해튼 거리를 휴리스틱으로 삼는다. 어차피 이 문제를 아무리 단순히 생각해도 출발 지점에서 목적지 중 한 점, 그리고 그 점에서 다른 한 점, 이런 식으로 네 점을 돌아야 하는 건 같으므로, 그 각각을 별개의 탐색으로 설정하면 적절한 휴리스틱으로 그래프 탐색을 하는 것으로 판단할 수 있다.

-목적지를 배치한 순서들 각각에 대해, 4번 루프를 도는데, 이는 한 목적지에 도착할 때마다 새로운 탐색을 시작하기 위해 open list와 close list를 초기화하며 방문 노드 체크를 전부 초기화하기 위함이다. 이렇게 진행된 경로는 tmppath에 저장되고, 이전 탐색의 도착 지점이 새로운 탐색에서 중복되어 저장하는 것을 막기 위해 start_point를 먼저 tmppath에 넣은 뒤 각 탐색의 시작점은 tmppath에 집어넣지 않는 식으로 중복 집계를 예방했다.

1) small



2)medium




```
==========================================
[ astar_four_circles results ]
(1) Path Length: 107
(2) Search States: 8078
(3) Execute Time: 0.0753595829 seconds
==========================================
```

3) big


```
==========================================
[ astar_four_circles results ]
(1) Path Length: 163
(2) Search States: 15643
(3) Execute Time: 0.1526973248 seconds
==========================================
```

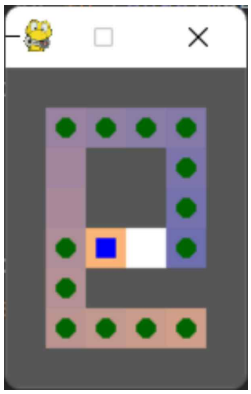
5. A* with stage 3(MST)

-stage3에서는 mst를 휴리스틱에 반영하기 위해 우선 현재 노드에서 아직 방문하지 않은 목적지 노드들을 잇는 MST를 구축하는 mst 함수를 만들었다. 즉 해당 점에서 미방문 목적지를 잇는 mst 간선 코스트 합을 휴리스틱으로 설정한 것이다. 우선 stage3_heuristic에서 아직 방문하지 않은 노드들의 리스트를 만들고 mst 함수를 호출한 값을 리턴하는 형식인데, mst 함수에서는 mst에 포함된 간선들의 가중치 합을 리턴하기에 이를 휴리스틱으로 삼았다. mst 함수는 모든 미방문 목적지와 현재 점들을 대상으로 모든 간선을 맨해튼 거리를 코스트로 하여 edges에 집어넣은 뒤, kruskal algorithm을 사용해 mstedges에 집어넣었다. mstedges가 완성되면 안에 포함된 간선들의 cost를 모두 더한 뒤 리턴하는 것이 mst의 역할이다.

-사이클이 만들어지는 경우 그 간선은 mst에 포함하지 않기 위해 disjoint set 자료구조를 사용했다. Disjoint 클래스를 정의하여 노드의 root를 찾는 set_find와 두 노드의 set을 결합하는 set_union함수를 만들었다.

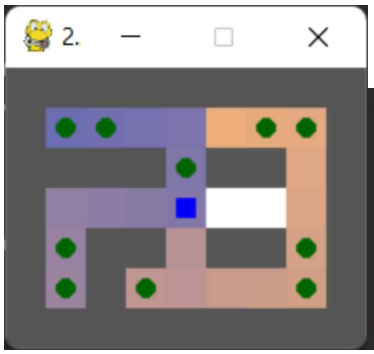
-다만 여기서 코스트의 판단 기준을 맨해튼 거리를 사용해서인지 mst가 분명 서로 같음에도 차이가 났다. 실제로 없는 경로이지만 있는 것으로 생각하여 그쪽으로 mst를 이을 수 있다고 생각한 것이다. 이 때문에 최단경로를 정확히 찾지는 못했고, 적당히 짧은 경로를 찾는 선에서 그쳤다.

1. tiny



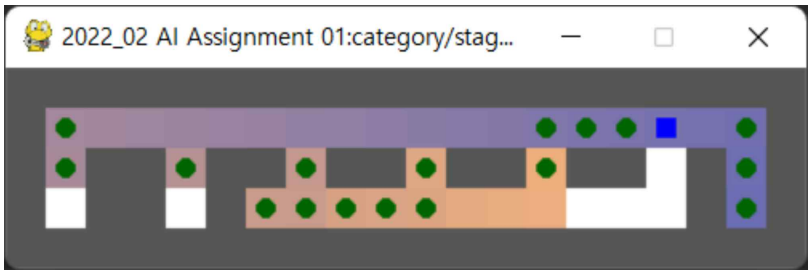
```
==========================================
[ astar_many_circles results ]
(1) Path Length: 21
(2) Search States: 22
(3) Execute Time: 0.0093886852 seconds
==========================================
```

2. small



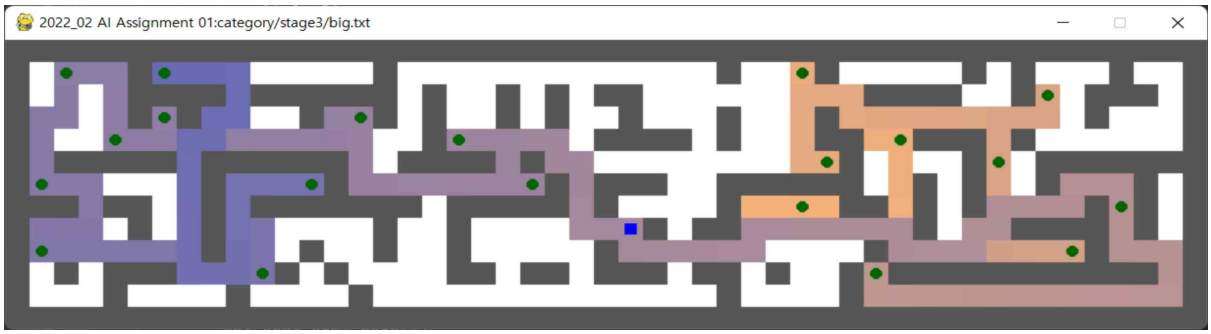

```
==========================================
[ astar_many_circles results ]
(1) Path Length: 32
(2) Search States: 35
(3) Execute Time: 0.0070018768 seconds
==========================================
```

3. medium



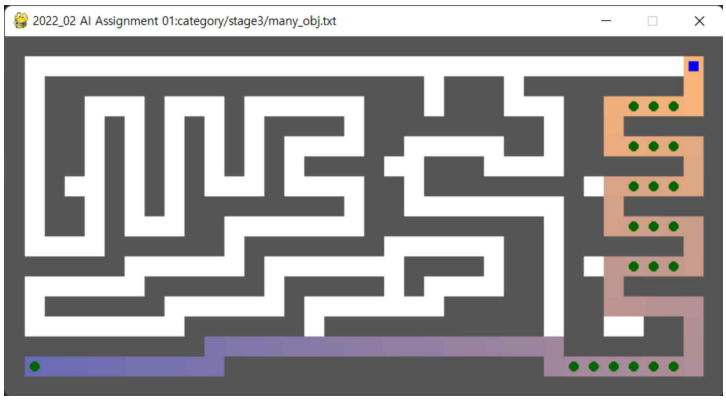

```
==========================================
[ astar_many_circles results ]
(1) Path Length: 49
(2) Search States: 75
(3) Execute Time: 0.0334854126 seconds
==========================================
```

4. big



```
==========================================
[ astar_many_circles results ]
(1) Path Length: 239
(2) Search States: 745
(3) Execute Time: 0.5801167488 seconds
==========================================
```

5. many_obj




```
==========================================
[ astar_many_circles results ]
(1) Path Length: 75
(2) Search States: 91
(3) Execute Time: 0.0679550171 seconds
==========================================
```