기초 인공지능 Assignment01 – Maze Game 보고서

2021.9.25 11:59 PM

20181202 김수미

1. 사용한 Python 라이브러리 설명

1) Pygame Library

Pygame은 Python 언어 라이브러리로 2D 게임 개발에 사용되며, Python 모듈 세트를 사용하여 게임을 개발할 수 있는 플랫폼이다. 인공지능 및 머신러닝 분야에서는 성능 실험을 위해 게임을 설계하는 경우가 많아 Pygame이 유용하게 사용될 수 있다. 해당 과제에서는 Pygame 라이브러리 이외 추가적인 라이브러리는 사용하지 않았다.

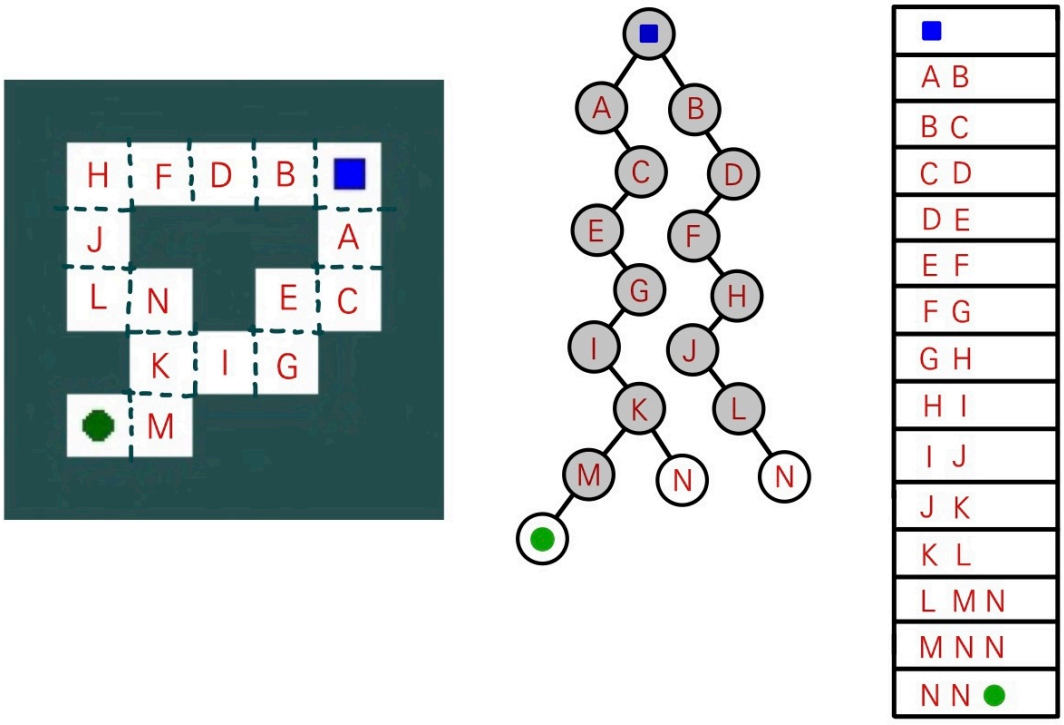
2. 알고리즘 구현 방법 설명

1) Stage1 : BFS 알고리즘

첫번째 단계에서는 BFS 알고리즘을 이용해 시작 지점부터 Goal 지점까지의 최단 경로를 계산하는 코드를 작성했다. BFS 알고리즘이란 시작 정점으로부터 가까운 정점을 먼저 방문하고 멀리 떨어져 있는 정점을 나중에 방문하는 순회 방법으로 깊게(deep) 탐색하기 전에 넓게(wide) 탐색하는 방법이다. 미로의 최단 탈출 경로 탐색에서 Breath First Search(BFS) 알고리즘을 사용하기 위해서는 Queue와 Tree 자료구조가 필요하다. 이 알고리즘이 미로의 최단 탈출 경로를 탐색하는 과정은 다음과 같다.

먼저 미로의 각 칸은 노드로 표현되며, 각 노드는 현재 위치 및 parent 노드 정보, 방문한 미로의 칸 정보를 저장한다. 시작지점 노드는 BFS Tree의 root 노드가 되며, 해당 지점에서 이동할 수 있는 모든 칸이 노드의 child 노드가 된다. 새로운 노드가 tree에 추가될 때 마다 해당 노드는 동시에 queue에 저장된다. queue에서 pop되는 노드들의 child 노드가 차례대로 BFS tree와 queue에 저장되며 계속 BFS tree를 확장해 나가며 방문한 미로의 위치는 노드의 2차원 배열에 기록하여 지나간 길을 다시 방문하는 것을 방지한다. 즉 미로에서 시작점을 기준으로 나아갈 수 있는 경로가 모든 방향으로 가지를 뻗어 나가며 탐색 되고, 그 과정에서 goal 지점이 발견되면 탐색을 종료하고 goal 지점 노드에서부터 부모 노드를 따라가며 탈출 경로 path를 생성한다.

stage1의 small.txt를 예시로 보면, BFS tree와 queue는 아래와 같은 과정을 통해 생성된다.



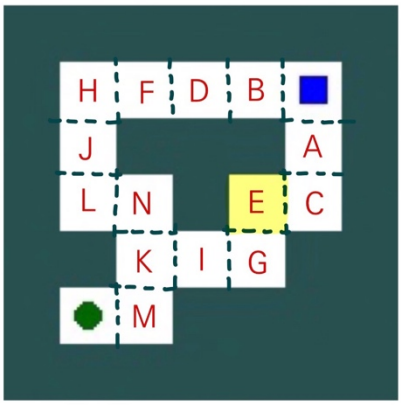
〈 그림1. 왼쪽부터 주어진 미로, BFS tree, queue / tree의 회색 노드는 expand된 노드이다 〉

2) Stage1 : A* 알고리즘

두번째 단계에서는 A* 알고리즘을 이용해 시작 지점부터 Goal 지점까지의 최단 경로를 계산하는 코드를 작성했다. A* 알고리즘이란 f(n) = g(n) + h(n)을 평가 함수로 사용하는 Best First Search 알고리즘이다. 이 때 g(n)은 초기 상태에서 노드 n으로의 경로 비용이고 h(n)은 n에서 목표로의 최단 경로의 추정 비용이다. 과제에서 h(n)값을 계산하기 위한 heuristic function으로 manhatten distance를 사용했다. 미로의 최단 탈출 경로 탐색에서 A* 알고리즘을 사용하는 경우, Tree와 배열 자료구조를 사용할 수 있다. 이 알고리즘이 미로의 최단 탈출 경로를 탐색하는 과정은 다음과 같다.

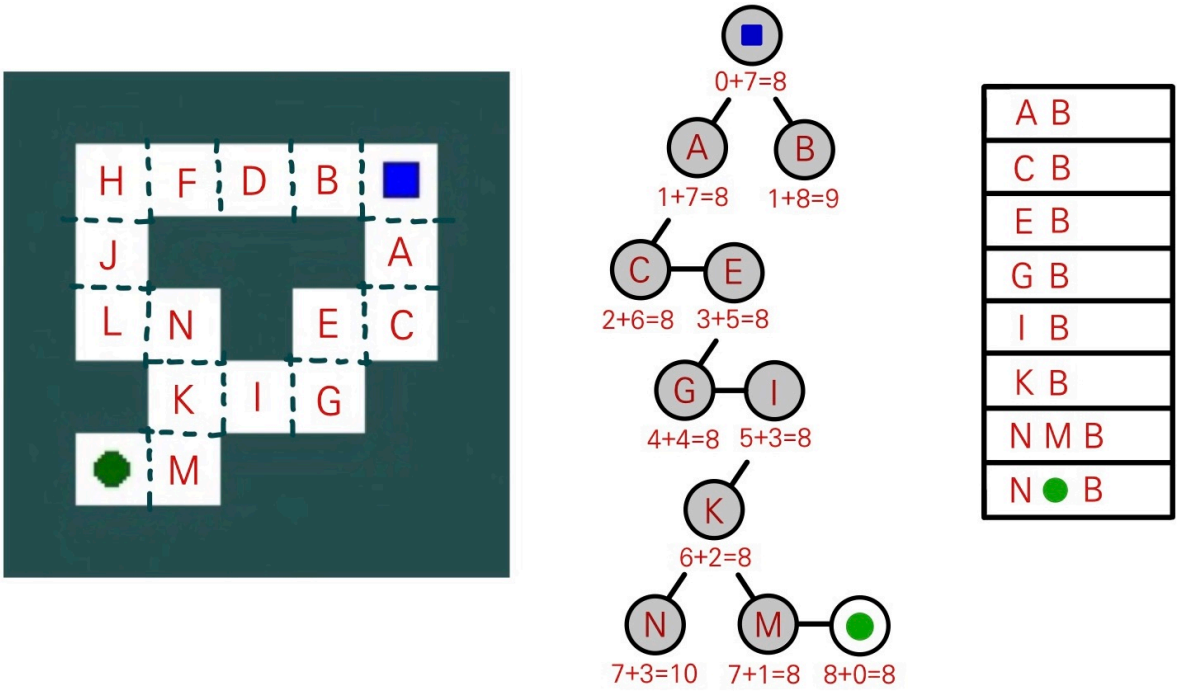
먼저 미로의 각 칸은 알고리즘에서 노드로 표현되며, 각 노드는 현재 위치 및 parent 노드 정보, f(n), g(n), h(n) 값을 저장한다. 시작지점 노드는 tree의 root 노드가 되며 미로의 특정 칸에서 이동할 수 있는 칸이 해당 노드의 child 노드가 된다. A* 알고리즘에서 tree가 가지를 뻗어나가기 위해서는 open_list 와 closed_list가 필요하다. child가 없는 노드는 모두 open_list에 저장되며, 특정 노드가 open_list 선택되어 child 노드를 가지게 되면 해당 노드는 open_list에서 closed_list로 옮겨진다. open_list에서 어떤 값이 선택되는가? f(n)의 값이 가장 작은 노드가 선택된다. 선택된 노드는 child 노드를 가지게 되며 closed_list로 옮겨지고, 다시 새롭게 생성된 child 노드가 open_list에 추가되게 된다.

다음으로 f(n)이 어떻게 계산되는지 살펴보자. open_list에서 노드를 선택하는 지표는 f(n)의 값이다. f(n)은 g(n)과 h(n)의 합인데, n이 특정 노드를 가리킨다고 하면 g(n)은 시작 지점부터 특정 노드가 가리키는 미로의 칸 까지의 실제 거리값이고, h(n)은 특정 노드가 가리키는 미로의 칸에서 도착 지점 까지의 직선 거리값이다. 예를 들면 오른쪽 그림에서 E칸의 g(n) 값은 시작 지점에서 E까지의 실제 거리인 3이고, h(n) 값은 E에서 도착 지점까지의 직선 거리이다. E의 좌표가 (3,4) 이고 도착 지점의 좌표가 (5,1) 이므로 h(n) = 5 이다. 미로의 한 칸의 길이는 1로 간주하기 때문에 g(n)값은 parent노드의 g(n)값에 1을 더하면 되며, h(n)은 좌표값과 manhatten distance 공식을 이용해서 연산할 수 있다.



〈 그림2. small.txt 〉

tree를 확장시켜 나가면서 도착 지점 노드에 도달하는 경우 탐색은 종료된다. stage1의 small.txt를 예시로 보면, tree와 open_list, closed_list 는 아래와 같은 과정을 통해 생성된다.



〈 그림3. 왼쪽부터 주어진 미로, A* tree, open_list / tree의 회색 노드는 expand된 노드이다 〉

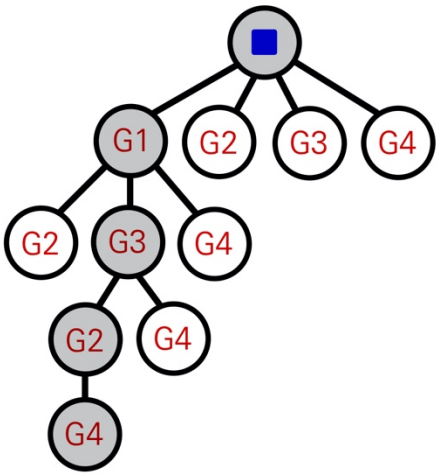
3) Stage2 : A* 알고리즘 with multiple goals

Stage2 에서는 A* 알고리즘을 이용해 Goal이 총 4개인 미로에서 모든 Goal을 경유하는 최단 경로를 계산하는 코드를 작성했다. Minimum Spanning Tree 알고리즘을 활용하여 Heuristic function을 정의했으며, MST 구현을 위해 Kruskal 알고리즘을 사용했다.

Stage1의 A* 알고리즘과 Stage2 에서 구현한 A* 알고리즘은 미로의 각 칸을 tree의 노드로 사용하지 않고, goal points들을 tree의 노드로 사용하여 모든 goal 지점을 최단거리로 경유하기 위해 어떤 순서로 goal points들을 방문해야 하는지를 먼저 계산한 다음 계산 결과를 바탕으로 미로의 탈출 경로를 만들었다.

각 노드의 g(n)값은 시작점에서부터 해당 노드까지의 실제 거리 값을 저장하며, 노드가 확장될 때 마다 c(n)값은 해당 노드의 parent 노드에서 해당 노드까지의 거리가 된다. 즉 특정 노드의 g(n)값은 parent 노드에서 특정 노드까지의 거리 + parent 노드의 g(n) 값 이다.

Heuristic function은 Minimum Spanning Tree알고리즘을 사용한다. 특정 노드가 만들어진 시점에서 선택된 지점들의 집합이 있을 것이다(해당 노드와 그 노드의 조상 노드들로 이루어진 집합). 선택된 지점들에서 모든 goals 그리고 시작지점까지의 거리를 계산하면 그 거리는 그래프의 edge 들이 된다. 이 방법으로 각 노드에서 그래프를 생성하고, Kruskal 알고리즘을 이용해 그래프의 MST를 계산한다. MST의 계산이 종료되면 해당 MST의 모든 edge값의 합인 cost_sum을 반환하며, 이는 노드의 h(n)값이 된다. 이렇게 구현하는 경우 A* 알고리즘의 tree는 아래와 같은 모양으로 생성된다.



〈 그림4. Stage2 A* 알고리즘 tree 예시. G1~G4는 4개의 goal 지점을 의미한다 〉

Tree 상에서 모든 goal 노드가 선택되면 탐색은 종료되고, seach function은 최단 경로로 모든 goal을 방문하기 위해서 어떤 순서로 방문해야 하는지를 list로 반환한다. 해당 list를 가지고 2단계에서 작성한 A* 알고리즘을 이용하면 빠르게 shortest path를 완성할 수 있다.

4) Stage3 : A* 알고리즘 using Minimum Spanning Tree

Stage3 에서는 A* 알고리즘을 이용해 Goal이 총 10개 이상인 미로에서 모든 Goal을 경유하는 최단 경로를 계산하는 코드를 작성했다. Minimum Spanning Tree 알고리즘을 활용하여 Heuristic

function을 정의했으며, MST 구현을 위해 Prim 알고리즘을 사용했다.

Stage2 에서는 goal 지점의 개수가 4개로 고정되어 있었기 때문에 goal 지점들을 A* search tree 의 노드로 사용할 수 있었지만, Stage3 에서는 goal 지점의 개수가 10개가 넘기 때문에 goal 지점들을 노드로 사용하는 경우 하나의 노드가 최대 9개의 child 노드를 가지게 되어 시간 및 공간복잡도의 효율성이 좋지 않다. 따라서 Stage3 에서는 Stage1과 동일하게 미로의 각 칸을 A* search tree의 노드로 사용했다. 이 방법은 Stage2에도 똑같이 적용할 수 있지만 위에서 실제로 Stage2에 적용한 방법 보다는 시간이 오래 걸릴 것이다.

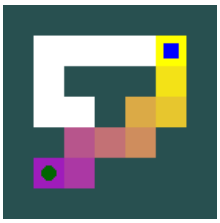
각 노드의 g(n)값은 시작점에서부터 해당 노드까지의 실제 거리 값이며, 노드가 확장될 때마다의 c(n) 값은 해당 노드의 parent 노드에서 해당 노드까지의 거리, 즉 1이다. 따라서 특정 노드의 g(n)값은 parent 노드의 g(n)+1 이다. 이는 Stage1과 동일하다.

Heuristic function은 Stage2와 동일하게 Minimum Spanning Tree알고리즘을 사용했지만 Stage3 에서는 MST를 구하기 위해 Kruskal 대신 Prim 알고리즘을 사용했다. h(n) 값은 현재 노드에서 가장 가까운 goal 노드까지의 거리 + 현재 노드와 모든 goal 노드를 포함하는 graph에서의 MST(edge의 길이(wheight)는 Stage1에서 사용한 A* 알고리즘 사용하여 계산)의 cost_sum이다. 해당 알고리즘을 사용하는 경우 small.txt와 medium.txt 미로의 경우 모든 goal 지점을 경우하는 최단 경로를 잘 계산 했지만, big.txt의 경우 시간이 지나치게 오래 걸려(1시간 이상) weighted A* 알고리즘을 확인하여 최단 경로의 근사값만 확인해 보았다.

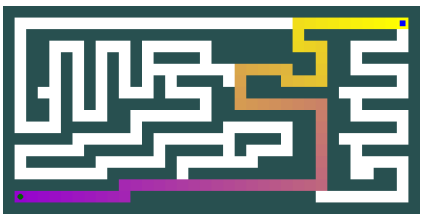
Stage1~3의 탐색 결과는 아래 코드 실행 결과 캡처 이미지에서 확인할 수 있다. Search Staes에는 데이터 전처리 과정에서까지의 모든 경우에 대한 노드 expand 여부가 포함되어 있어 값이 자칫 크게 느껴질 수 있다.

3. 코드 실행 결과 캡처 이미지

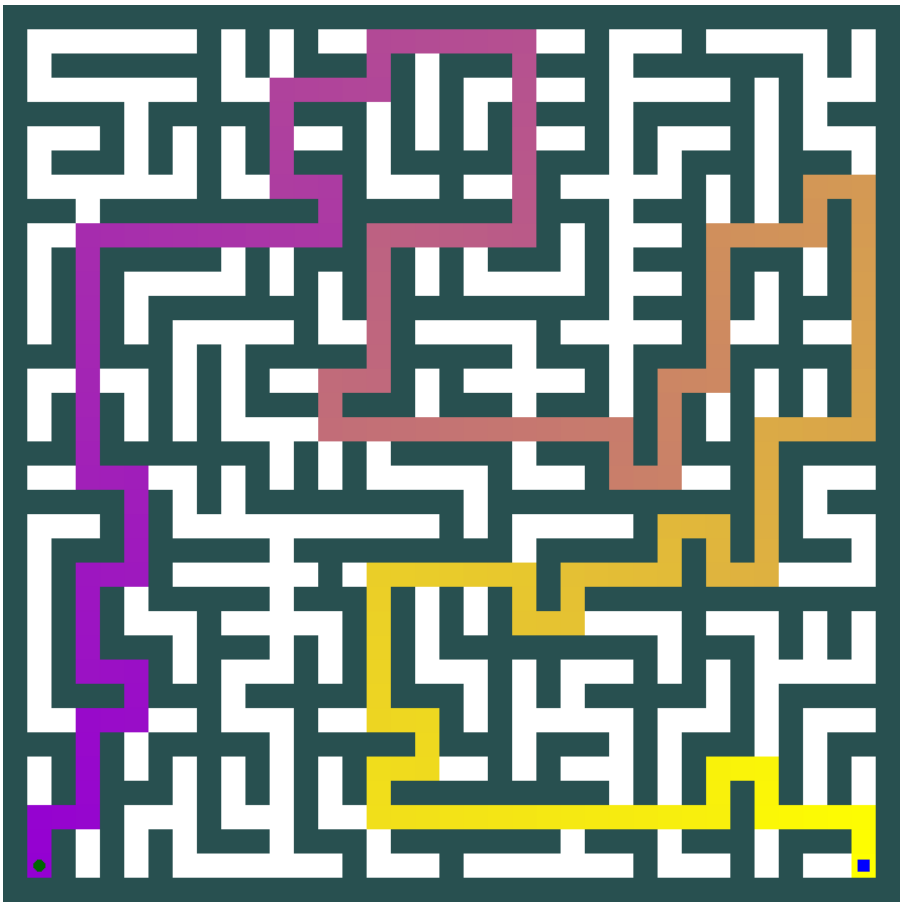
1) Stage1 : BFS method



```
==========================================
[ bfs results ]
(1) Path Length: 9
(2) Search States: 14
(3) Execute Time 0.0002107620 seconds
==========================================
```

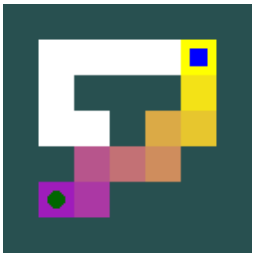


```
==========================================
[ bfs results ]
(1) Path Length: 69
(2) Search States: 267
(3) Execute Time 0.0050740242 seconds
==========================================
```

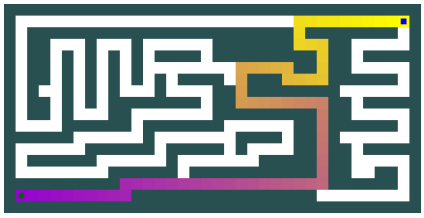


```
==========================================
[ bfs results ]
(1) Path Length: 211
(2) Search States: 616
(3) Execute Time 0.0079898834 seconds
==========================================
```

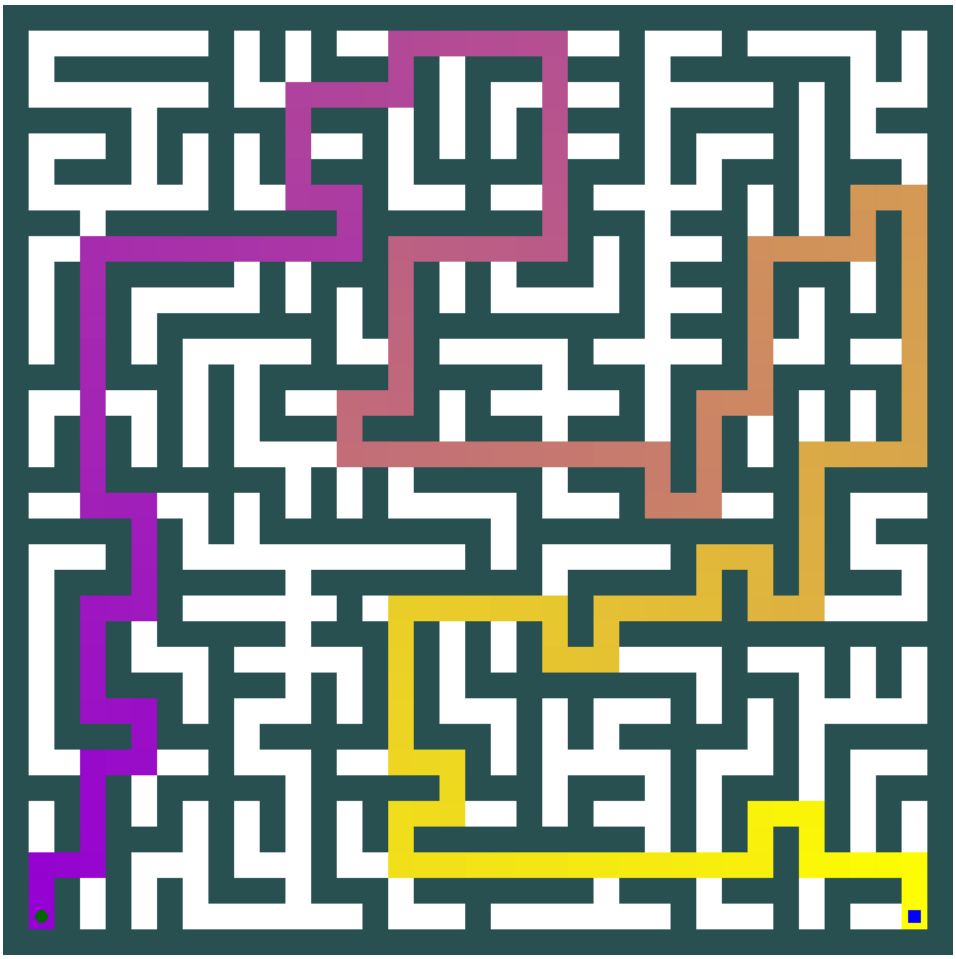
2) Stage1 : Astar method



```
==========================================
[ astar results ]
(1) Path Length: 9
(2) Search States: 10
(3) Execute Time 0.0002369881 seconds
==========================================
```

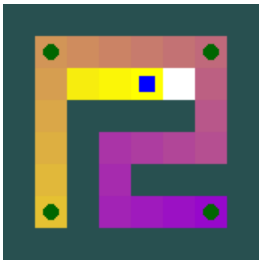


```
==========================================
[ astar results ]
(1) Path Length: 69
(2) Search States: 226
(3) Execute Time 0.0049300194 seconds
==========================================
```

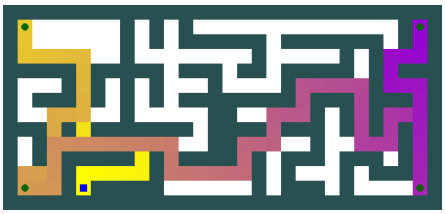


```
==========================================
[ astar results ]
(1) Path Length: 211
(2) Search States: 552
(3) Execute Time 0.0113470554 seconds
==========================================
```

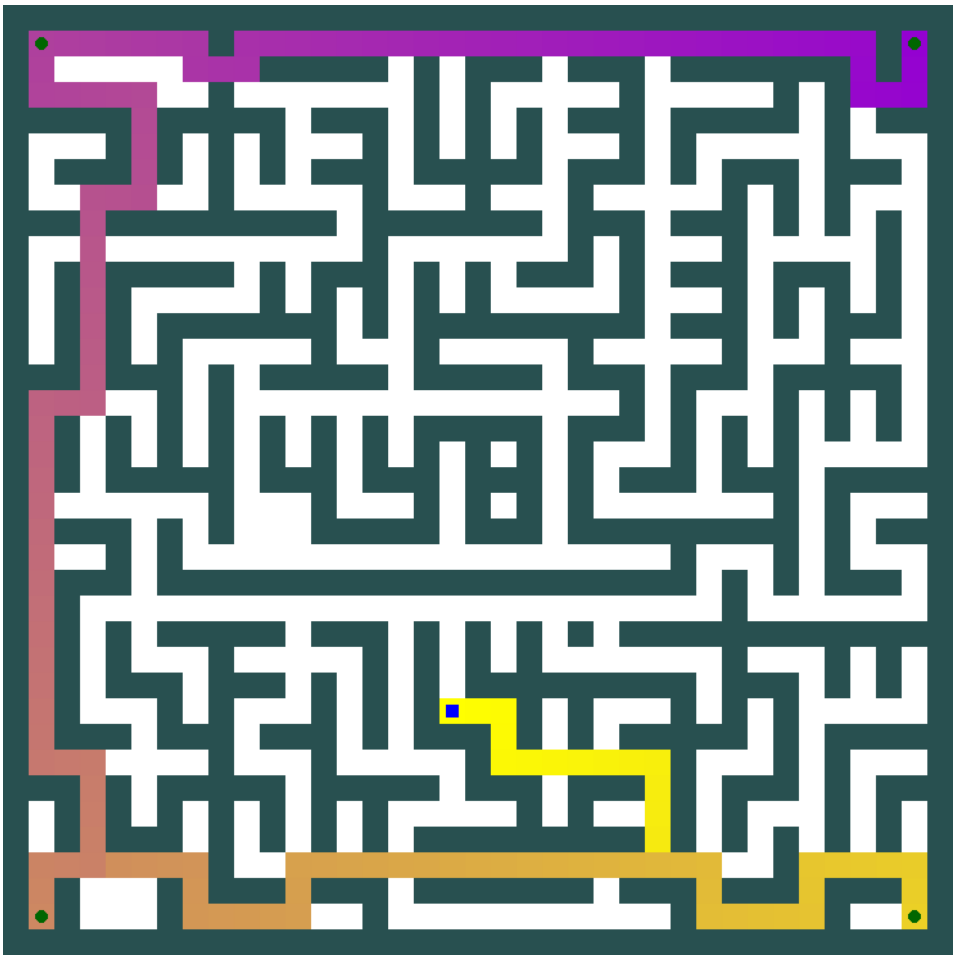
3) Stage2 : Astar four circles method



```
==========================================
[ astar_four_circles results ]
(1) Path Length: 29
(2) Search States: 318
(3) Execute Time 0.0063288212 seconds
==========================================
```

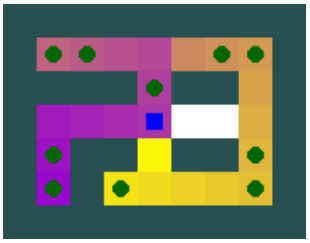


```
==========================================
[ astar_four_circles results ]
(1) Path Length: 107
(2) Search States: 2112
(3) Execute Time 0.0353951454 seconds
==========================================
```

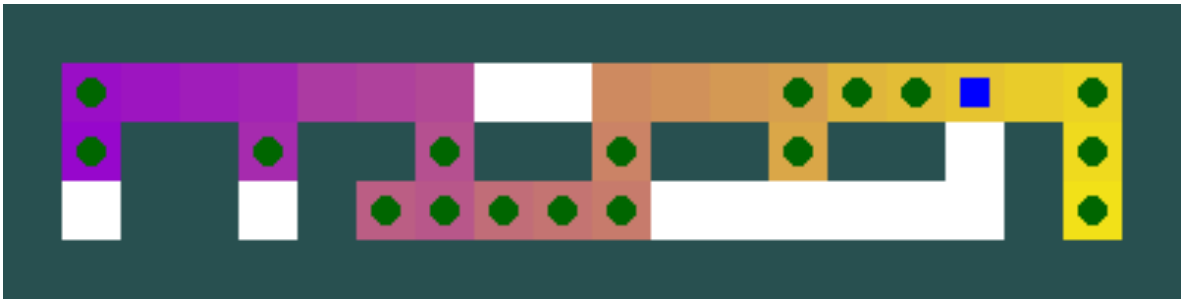


```
==========================================
[ astar_four_circles results ]
(1) Path Length: 163
(2) Search States: 3948
(3) Execute Time 0.0611240864 seconds
==========================================
```

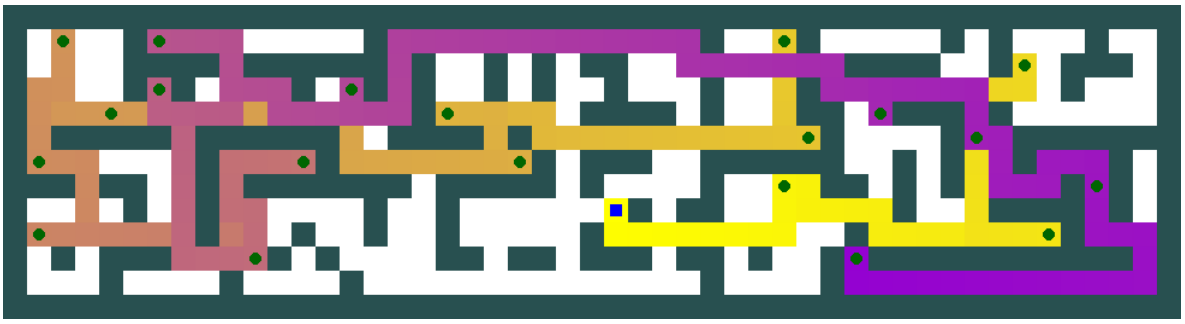
4) Stage3 : Astar many circles method



```
==========================================
[ astar_many_circles results ]
(1) Path Length: 28
(2) Search States: 1950
(3) Execute Time 0.0640740395 seconds
==========================================
```



```
================================================
[ astar_many_circles results ]
(1) Path Length: 35
(2) Search States: 4768
(3) Execute Time 0.2871286869 seconds
================================================
```



```
================================================
[ astar_many_circles results ]
(1) Path Length: 233
(2) Search States: 469833
(3) Execute Time 25.8440628052 seconds
================================================
```