

---

과목 명: 기초인공지능

담당 교수 명: 양 지 훈

< <AI\_assignment01> >

서강대학교 컴퓨터학과

[학번] 20171700

[이름] 최재원

# 목 차

1. 프로그램 개요	3
2. 프로그램 흐름도	3
2.1 stage1, stage2, stage3	3
3. 프로그램 설명	4
3.1 Stage1	4
3.1.1 출력 결과	4
3.1.2 사용 라이브러리	5
3.1.3 알고리즘 설명	5
3.2 Stage2	6
3.2.1 출력 결과	6
3.2.2 사용 라이브러리	7
3.2.3 알고리즘 설명(Heuristic Function)	7
3.3 Stage3	8
3.3.1 출력 결과	8
3.3.2 사용 라이브러리	9
3.3.3 알고리즘 설명(Heuristic Function)	9

## 1. 프로그램 개요

Uninformed Search 인 BFS(Bredth-First Search)와 Informed Search (Heuristic Search)인 A\* search 알고리즘을 이용해 미로 안에서 최단 거리를 구하는 것을 목표로 한다. Stage 1에서는 출발점 T로부터 single dot 까지의 최단거리를, Stage 2에서는 4 개의 corner 에 존재하는 dots 를 모두 지나는 최단거리를, 마지막으로 Stage 3 에선 multiple dots 를 모두 지나는 최단거리를 구한다. Stage2 와 Stage3 에서의 heuristic function 은 manhattan 이 아닌 다른 함수를 작성하였다.(manhattan distance 를 활용하여 작성.)

## 2. 프로그램 흐름도

### 2.1 stage1, stage2, stage3

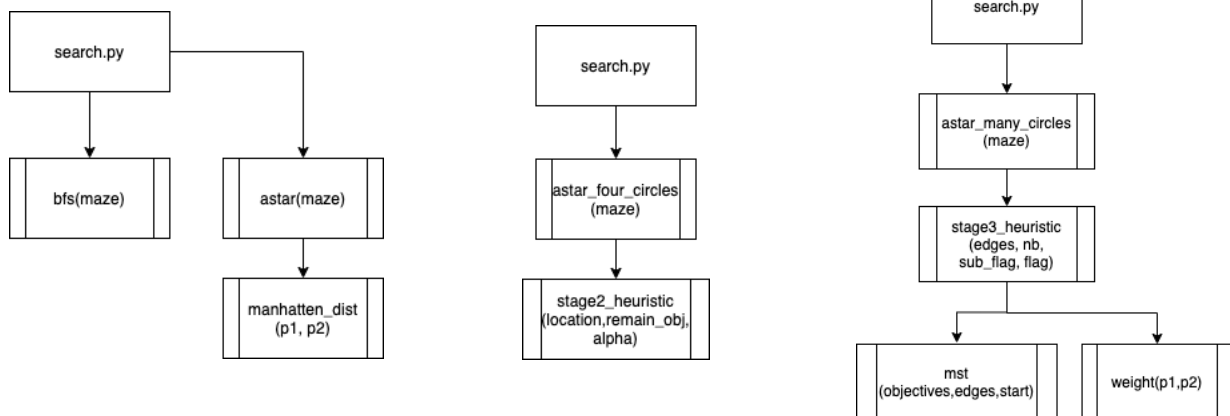


그림 1> 프로그램 흐름도 (좌에서 우순으로 각 stage1,2,3)

Stage1 에선, bfs 와 aster function 을 이용하며, 이 때 aster 알고리즘에 대한 heuristic function 은 manhattan 거리로 정의되어 있다. Stage2 에선 직접 정의한 heuristic function 을 사용하고, Stage3 에서 또한 직접 정의한 함수를 사용한다. 추가적으로 stage3 의 aster\_many\_circles 에선 mst 를 활용한 heuristic function 을 사용한다. 모든 aster 함수에서는 maze 클래스 객체 내의 함수인 maze.neighborPoints()를 사용하여 인접해있는 좌표들을 얻는다. stage2 에서의 A\* 알고리즘을 stage1 에 적용시키거나, stage3 에서의 A\* 알고리즘을 stage1,2 에 적용시켰을때 정상적으로 작동함을 확인하였고, 따라서 최종적인 stage3 에서의 A\*알고리즘이 보다 범용적인 알고리즘이라고 할 수 있다.


### 3. 프로그램 설명

#### 3.1 Stage1

##### 3.1.1 출력 결과

1) Small.txt

```
=====
[ bfs results ]
(1) Path Length: 9
(2) Search States: 15
(3) Execute Time 0.0000910759 seconds
=====
```

A small maze visualization with a green start point at the bottom left and a blue goal point at the top right. The path is highlighted in yellow, and the search area is highlighted in orange.

```
=====
[ astar results ]
(1) Path Length: 9
(2) Search States: 14
(3) Execute Time 0.0002279282 seconds
=====
```

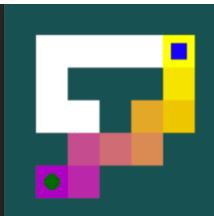
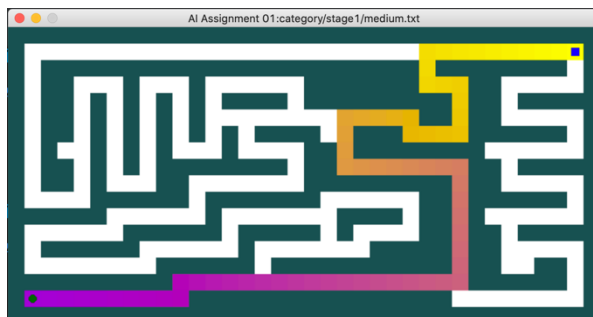
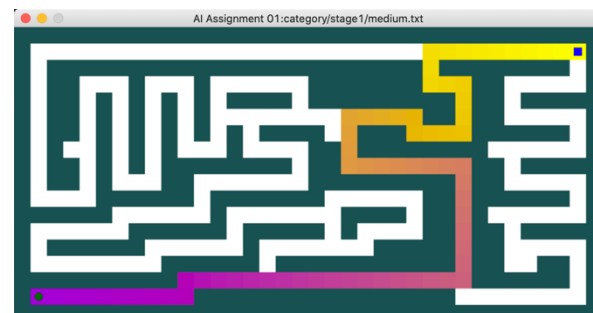
A small maze visualization with a green start point at the bottom left and a blue goal point at the top right. The path is highlighted in yellow, and the search area is highlighted in orange.

그림 2> BFS (상) A\*(하) for small.txt

2) medium.txt



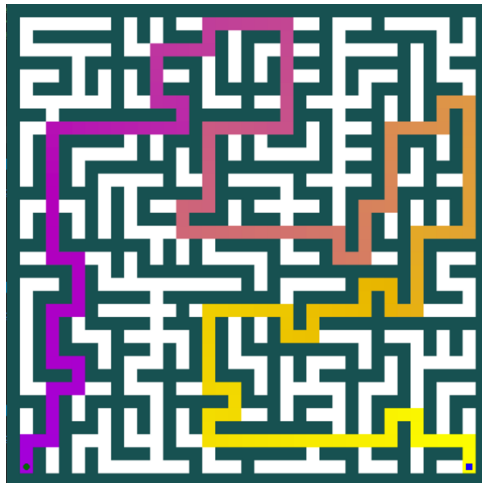
```
=====
[ bfs results ]
(1) Path Length: 69
(2) Search States: 269
(3) Execute Time 0.0013220310 seconds
=====
```



```
=====
[ astar results ]
(1) Path Length: 69
(2) Search States: 224
(3) Execute Time 0.0089719296 seconds
=====
```

그림 3> BFS (상) A\*(하) for medium.txt

3) big.txt



```
=====
[ bfs results ]
(1) Path Length: 211
(2) Search States: 622
(3) Execute Time 0.0028948784 seconds
=====
```

```
=====
[ astar results ]
(1) Path Length: 211
(2) Search States: 549
(3) Execute Time 0.0412189960 seconds
=====
```

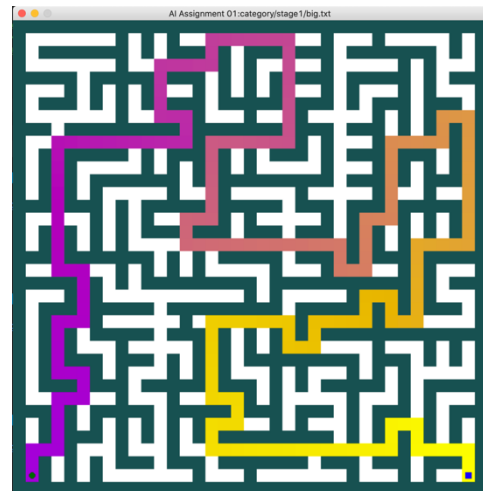


그림 4> BFS (상) A\*(하) for big.txt

### 3.1.2 사용 라이브러리

1) <From collections import deque>

BFS 알고리즘에서 좌표를 queue 자료구조에 저장하기 위해 사용한다. Deque 는 양방향 큐이지만, 일반적인 단방향 큐로 사용하였다.

### 3.1.3 알고리즘 설명

1) BFS algorithm

BFS 는 uninformed search 중 하나이다. 이를 구현하기 위해 2 차원 visited 배열, queue, 그리고 path 를 저장하기 위한 route(Node 객체)의 자료구조를 선언하였다. 반복문 안에서 maze.neighborPoints 함수를 호출하여 시작점으로부터 인접한 동서남북 4 방향을 조회한다. Wall 이 아닌 경우 해당 좌표들을 list 로 받아와 neighbor 에 저장하고, 이 좌표들 중 아직 방문하지 않은 좌표를 queue 에 삽입한다. 이 후 해당 queue 에 들어있는 좌표들을 pop 하여 목적지 좌표(dot)이 나오면 종료한다. Path 는 Node 객체인 route 의 클래스 변수 obj 에 저장하였다.

## 2) A\* algorithm

Informed search 알고리즘으로써, heuristic 함수를 사용하며 이에 따라 알고리즘의 시간 복잡도 및 공간 복잡도가 영향을 받는다. Open list 와 Closed list 의 자료구조는 각각 frontier, reached 의 이름을 가진 list 를 사용하였다. BFS 에서와 마찬가지로 주변 좌표를 탐색할 때 maze.neighborPoints 함수를 사용한다. 해당 좌표들을 Location 으로 하는 Node 객체를 생성하고, 이 node 들에 대해서 reached (closed\_list)에 존재하지 않는 경우에만  $G(n)$ ,  $H(n)$ ,  $F(n)$ 값을 계산하여, frontier 에 저장한다. 이 후 frontier 에서 가장  $f$  값이 작은 node 를 반복문을 통해 구하고, 해당 노드를 parent 로 하는 child 노드들을 앞서 설명한 maze.neighborPoints 함수로 부터 좌표를 얻어와 생성한다. Frontier 에서 pop 한 node 의 location 이 도착지점일시에 현재 방문한 node 로부터 parent 로 거슬러 올라가면서 path[]에 그 Location 값을 저장한다. heuristic function 으로는 manhattan\_dist 를 사용하며, 이는 실제 거리값  $h(n)$ '에 비해 한참 이상적인 값이라고 할 수 있다. 하지만, 작거나 같음은 만족하므로 admissible 하고 따라서 최단경로를 찾을수 있다.

## 3.2 Stage2

### 3.2.1 출력 결과

#### 1) Small.txt

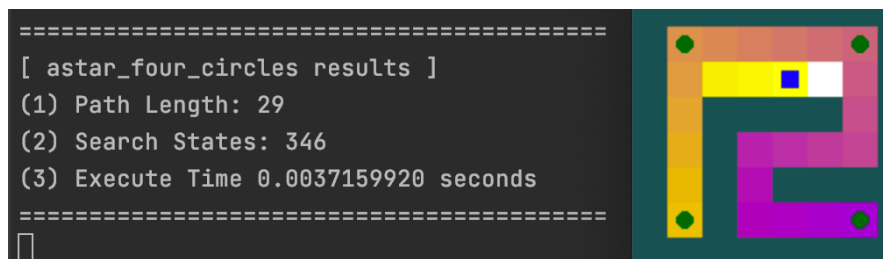


그림 5> A\*\_four\_circles for small.txt

#### 2) medium.txt

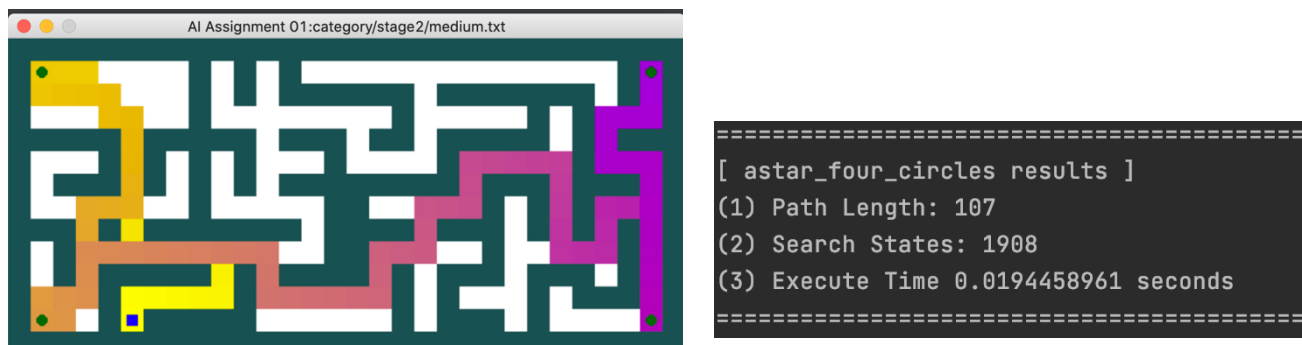


그림 6> A\*\_four\_circles for medium.txt

### 3) big.txt

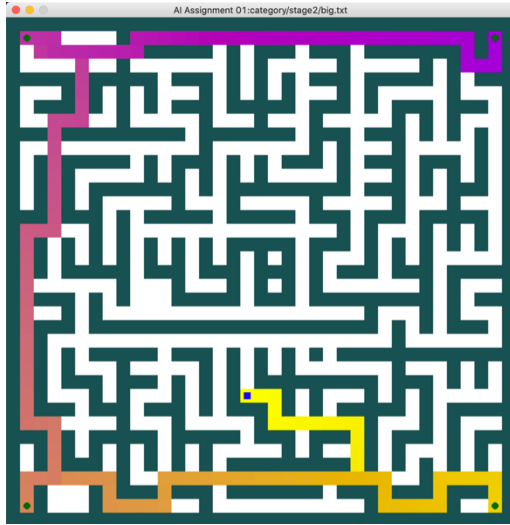


그림 7> A\*\_four\_circles for big.txt

```
=====
[ astar_four_circles results ]
(1) Path Length: 163
(2) Search States: 6857
(3) Execute Time 0.0729680061 seconds
=====
```

### 3.2.2 사용 라이브러리

#### 1) <from heapq import \*>

목적지가 하나인 앞선 stage 와 다르게 stage2 에서는 openlist 역할을 하는 frontier 를 heap 으로 사용하고, 이를 통해 항상 f 가 최소인 node 를 확인할때 사용하였다.

### 3.2.3 알고리즘 설명(Heuristic Function)

해당 stage 에서는 목적지가 총 4 군데이다. 따라서 이전에 썼던 manhattan dist 방식의 heuristic function 을 사용하지 않고, 새로 정의하였다. stage2\_heuristic() 함수에서 h(n)값을 계산한다. 해당 좌표에서 가장 가까운 목적지 dot 까지의 거리를 manhattan 거리로 구하고, 거기에 해당 dot 을 제외한 나머지 dots 들끼리의 거리를 더한값을 h(n)으로 정의하였다. 이는 최종적으로 모든 목적지를 방문해야하기 때문에, 나머지 dots 들까지 고려한 값이고, 실제거리 h'(n)보다 항상 작거나 같으므로 admissible 하다. 한편, node 를 순회하는 방식을 heap 으로 사용하고 graph 방식(repeated state 를 closed list 인 reached 를 사용하여 없앤다. repeated state 를 방지하기 위해 defaultdict 를 사용하여 파생된 node 의 location 과 obj 를 key 값으로 하는 dictionary 가 존재하면 해당 노드를 openlist 인 heap 에 삽입하지 않는다.)이며, heuristic function 은 f(n)이 Path 내에서 nondecreasing 하고 삼각형 정리에 따라서 heuristic function 은 consistent 함을 만족한다.

### 3.3 Stage3

#### 3.3.1 출력 결과

##### 1) Small.txt

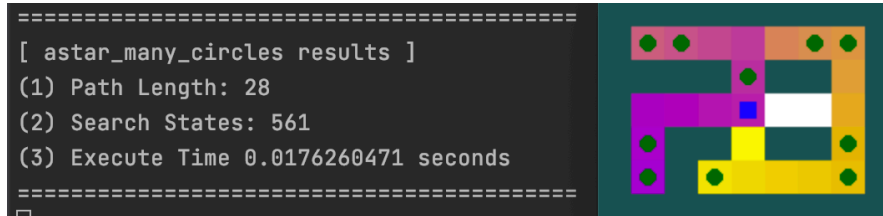


그림 8> A\*\_many\_circles for small.txt

##### 2) medium.txt

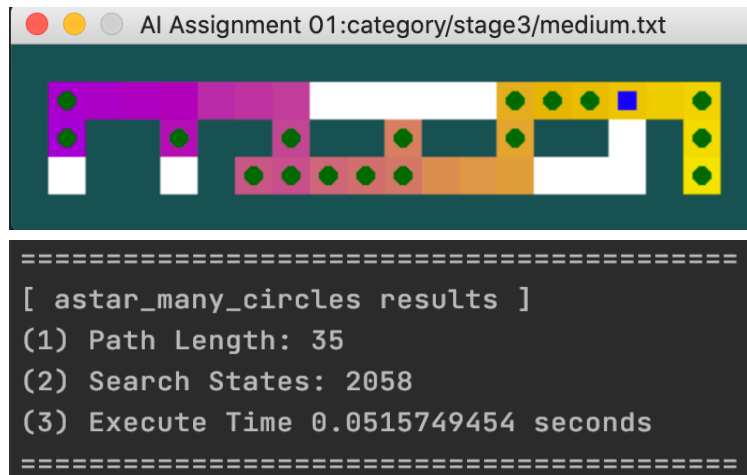


그림 9> A\*\_many\_circles for medium.txt

##### 3) big.txt

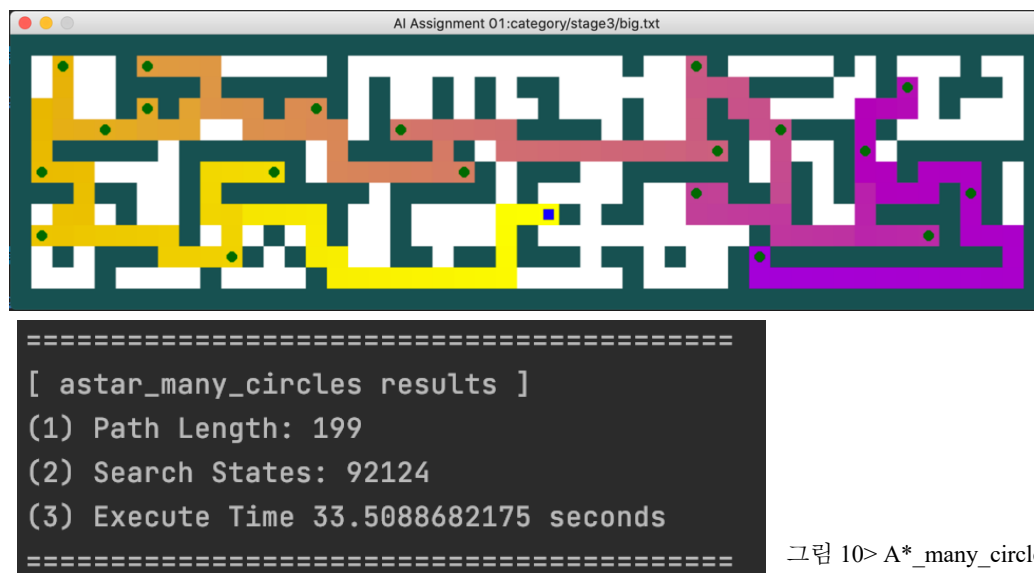


그림 10> A\*\_many\_circles for big.txt



### 3.3.2 사용 라이브러리

1) <from collections import defaultdict>

mst 에 사용할 edges 의 weight 들을 저장하기 위해 dictionary 를 사용하였다. 해당 weight 는 기존의 목적지가 하나인 A\* 알고리즘을 사용한다. 2 차원 dictionary 배열로 사용하며, 해당 key 값은 astar 에서의 시작점과 도착점이다. 또한 여러개의 목적지들에 대한 A\*알고리즘에서 repeated state 를 방지하기위한 수단으로 defaultdict 를 사용하였다.

2) <import copy>

다른 node 및 list 에 대해 node.obj, edges 의 정보를 복사할때 원본의 정보를 보존하기 위하여 copy 의 deepcopy 를 사용하였다.

3) <from heapq import \*>

목적지가 하나인 앞선 stage 와 다르게 stage2 에서는 openlist 역할을 하는 frontier 를 heap 으로 사용하고, 이를 통해 항상 f 가 최소인 node 를 확인할때 사용하였다. 또한 heuristic function 에서 사용하는 mst function 내부에서 prim 방식을 택하여, heap 구조를 사용한다.

### 3.3.3 알고리즘 설명(Heuristic Function)

astar\_many\_circles 함수는 크게 두 부분으로 나뉘어진다. 전반부에서는 mst 에서 사용할 edges 의 weight 를 A\* 알고리즘을 사용하여 미리 계산하여 2 차원 dictionary 배열에 저장한다. 그리고 후반부에서는 maze.neighborPoints 함수를 사용하여 주변 좌표들을 탐색해 나가는데, repeated state 를 방지하기 위해 defaultdict 를 사용하여 파생된 node 의 location 과 obj 를 key 값으로 하는 dictionary 가 존재하면 해당 노드를 openlist 인 heap 에 삽입하지 않는다. 이 방법은 stage2 에서와 같고, 다른 점은 Heuristic function 이다. 해당 stage 의 heuristic function 에서는 현재 node 의 location 에서 가장 가까운 end\_point 까지의 거리와 나머지 end\_points 들의 거리의 합을  $h(n)$ 값으로 하는것에는 stage2 에서와 의미는 같지만, 이를 구하는 방식은 다르다.

우선, 현재 node 의 location 에서 가장 가까운 end\_point 를 시작으로 하여 end\_points 들끼리의 거리를 MST function 을 이용하여 구하고, 이 때 prim 방식을 선택하였다. 그래프 내의 적은 숫자의 간선만을 가지는 희소 그래프의 경우에는 Kruskal 알고리즘이 적합하지만, 해당 문제에서는 end\_point 사이의 거리를 모두 구하고, 이는 간선이 많이 존재하는 밀집 그래프의 경우에 해당한다고 판단하여 primd 알고리즘을 택하였다.

다음으로 현재 node 의 location 에서 가장 가까운 end\_point 를 선택하는 과정이 이전과 좀 다른데, 현재 node 의 location 이 end\_point 인 경우 앞서 계산하였던 edges 를 활용하여 좀 더 admissible 하게 작성하였다. 추가적으로 end\_point 와 거리가 1 인 좌표들, 즉 end\_point 들의 neighborPoints 들에 대해서는, manhattan 거리를 사용하지 않고, 조금 더 admissible 한 거리를 적용하였다. 해당 좌표들은 다른 end\_point 로부터 parent 노드의 location 인 end\_point 까지의 실제거리에서 +1 이 되거나, -1 이 된다(벽이 사방에 존재하지는 않으므로). 따라서 이러한 경우를 분류하여 manhattan 거리와 edges 에 저장된 부모 node 의 location 까지의 거리중(A\*를 사용하여 구한

값.) 더 큰값을 채택하도록 하였고, 물론 end\_point 는 거리가 현재 node 로부터 가장 짧은 점을 선택한다. 이 때 edges 에 저장된 값을 선택할 경우 실제 거리인  $h'(n)$ 은 +1 이 되거나 -1 이 될 수 있는데, 이에 대한 평균치보다 작게 설정하여 최대한 admissible 한 조건을 만족시키며 값이 커질수 있도록 하였다. 그리고 end\_point 도 아니고, end\_point 에 바로 인접해 있는 좌표가 아닌 경우에는 manhattan 거리를 활용하여 현재 노드의 좌표에서 가장 가까운 end\_point 를 찾는데, 미로 찾기문제에서 manhattan 거리는 가장 간단하면서도 복잡하지 않은 미로에서는 꽤 admissible 함을 알 수 있었다. 최종적으로, 해당 heuristic function 에 의해  $f(n)$ 은 nondecreasing 하는 방식으로 선택되며 heuristic 은 consistent 하다.