Homework#1: Implementing Uniform Cost Search and A* Search Algorithms

Name : 임용성

Student ID [redacted]

[목차]

Ⅰ. Search Algorithm 설명 및 구현

■ Uniform_Cost_Search (at my.solver.py)

```
class UniformCostSearch():

    """brute-force는 무작위로 이웃 셀을 주는 method이고, fancy는 목표와 가까운 이웃 셀을 주는 method이다."""
    def __init__(self):
        logging.debug('Class uniform_cost_search ctor called')

        self.name = "Uniform Cost Search Algoritm"


    def uniform_cost_search(self, maze):

        logging.debug("Class uniform_cost_search solve called")


        priority_queue = PriorityQueue()                        # 우선순위 큐로 구현 (O(logn)의 시간복잡도를 가짐. (cf. list는 O(n)))
        priority_queue.put((0, maze.entry_coor))                # 시작 위치의 cell을 큐에 입력, 시작위치는 g=0
        path = list()                                           # To track path of solution cell coordinates

        print("\nSolving the maze with Uniform-Cost search...")
        time_start = time.perf_counter()

        while True:                                             # Loop until return statement is encountered
            while priority_queue:                               # While still cells left to search on current level
                stack, (k_curr, l_curr) = priority_queue.get()  # Search one cell on the current level
                maze.grid[k_curr][l_curr].visited = True        # Mark current cell as visited
                path.append(((k_curr, l_curr), False))          # Append current cell to total search path

                if (k_curr, l_curr) == maze.exit_coor:          # Exit if current cell is exit cell
                    if True:
                        print("Number of moves performed: {}".format(len(path)))
                        time_cost = time.perf_counter() - time_start
                        print("Execution time for algorithm: {:.4f}".format(time_cost))
                    return len(path),path,time_cost             # Return cost,path,time

                neighbour_coors = maze.find_neighbours(k_curr, l_curr)  # Find neighbour indicies
                neighbour_coors = maze.validate_neighbours_solve(neighbour_coors, k_curr,
                                                                 l_curr, maze.exit_coor[0],
                                                                 maze.exit_coor[1], "brute-force")

                if neighbour_coors is not None:
                    for coor in neighbour_coors:
                        priority_queue.put((stack+1,coor))      # 이웃 셀을 우선순위 큐에 추가한다. (시작위치까지의 거리가 증가하므로 stack +1)

        logging.debug("Class uniform_cost_search leaving solve")
```

[Fig. Uniform Cost Search 알고리즘 코드]

Uniform Cost Search 알고리즘이란, Uninformed Search의 한 방법이다. Uniform Cost Search는 각 셀마다 비용을 부여하여 다음 셀을 선택한다, 즉 현재까지의 비용을 고려하여 가장 Cost가 낮은 다음 셀을 선택한다.

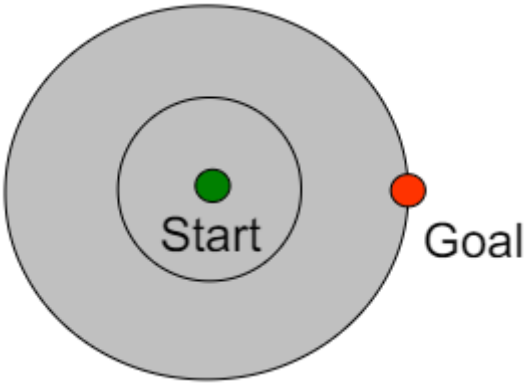
이 미로 예제에서 움직이는데 필요한 Cost가 모두 1로 고정되어 있으므로,

Uniform Cost Search와 Breadth First Search와의 차이가 없다고 할 수 있다.

이 알고리즘은 Completeness하고 Optimal하다는 장점이 있지만,

목표 위치에 대한 정보가 없기 때문에(Uninformed Search) 다음 그림과 같이 **'모든 방향'으로 확장해나가기 때문에 느리다는 단점**이 있다.

UCS에 특성 상 cost가 낮은 모든 방향으로 expand하기 때문에 경로가 넓고 깊지 않은 경우에 강점이 있다.



[Fig. 이 미로 문제에서 UCS는 BFS와 같고, '모든 방향'으로 확장한다는 단점이 있다]

<시간 복잡도>

모든 Cost가 1인 Uniform Cost Search 알고리즘의 시간 복잡도는 $O(b^d)$이다. (b : 이동가능한 방향의 개수, d : 미로의 깊이)

리스트를 이용하지 않고, 우선순위 큐를 이용하기 때문에 (O(n) → O(log(n))), 움직일 때마다, 큐에 있는 모든 요소를 확인하면서 가장 작은 비용의 경로를 찾기 때문에 **시간 복잡도는 최악의 경우 $O(b^d * logn)$**이다.

<공간 복잡도>

Open List는 탐색을 해야하는 곳을 등록하는 곳으로 위의 코드에서는 `priority_queue = PriorityQueue()`이다.

Closed List는 탐색이 완료된 곳을 등록하는 곳으로 위의 코드에서는 `path = list()`이다.

따라서 이 알고리즘의 공간 복잡도는 O(N)이다. (N은 미로 안에 모든 셀의 수)

<Optimal 과 Completeness>

이 상황에서 Uniform Cost는 모든 이동 cost는 1이기 때문에, Uniform Cost Search는 Optimal Path를 찾을 수 있고, completeness는 d가 유한 하기 때문에, 완전하다.

- **Optimal O**
- **Completeness O**

■ A_Star_Search (at my.solver.py)

```python
def heuristic(self, coor, exit_coor) :
    """
    목표 위치의 시작 위치와 비용. 휴리스틱 기능을 위해 유클리드 거리를 사용하십시오.
    Return = dist_to_target : (k_n, l_n) 과 END cell 까지의 거리
    """
    (k_n, l_n) = coor
    (k_end, l_end) = exit_coor
    dist_to_target = math.sqrt((k_n - k_end) ** 2 + (l_n - l_end) ** 2)          #유클리드 거리 계산을 위해 Math 함수의 sqrt를 사용한다.
    return dist_to_target                                                       #coor에서 출구까지의 유클리드 거리를 반환한다.

def a_star_search(self, maze):

    logging.debug("Class a_star_search solve called")

    priority_queue = PriorityQueue()                                            # 우선순위 큐로 구현 (O(logn)의 시간복잡도를 가짐. (cf. list는 O(n)))
    g = 0
    h = self.heuristic(maze.entry_coor, maze.exit_coor)                         # h는 휴리스틱 함수로 '현재 위치에서 출구까지의 유클리드 거리이다.'
    priority_queue.put((g + h, g, maze.entry_coor))                             # 큐의 구조 (g + h, g, (현재위치))
    path = list()                                                               # To track path of solution cell coordinates
    print("\nSolving the maze with A_star_search...")
    time_start = time.perf_counter()                                            # Error발생으로 인해 time.perf_counter을 사용하였다.
    while True:                                                                 # Loop until return statement is encountered
        while priority_queue:                                                   # While still cells left to search on current level

            g , (k_curr, l_curr) = priority_queue.get()[1:]
            maze.grid[k_curr][l_curr].visited = True                            # Mark current cell as visited
            path.append(((k_curr, l_curr), False))                              # Append current cell to total search path

            if (k_curr, l_curr) == maze.exit_coor:                              # Exit if current cell is exit cell
                if True:
                    print("Number of moves performed: {}".format(len(path)))
                    time_cost = time.perf_counter() - time_start
                    print("Execution time for algorithm: {:.4f}".format(time_cost))
                return len(path),path,time_cost                                 # Return COST, PATH, TIME

            neighbour_coors = maze.find_neighbours(k_curr, l_curr)              # Find neighbour indicies
            neighbour_coors = maze.validate_neighbours_solve(neighbour_coors, k_curr,
                                                             l_curr, maze.exit_coor[0],
                                                             maze.exit_coor[1], "brute-force")

            if neighbour_coors is not None:
                for coor in neighbour_coors:
                    h = self.heuristic(coor, maze.exit_coor)                    # 다음 셀의 휴리스틱 함수를 계산한다.
                    priority_queue.put((g + 1 + h, g+1, coor))                  # 이웃 셀을 우선순위 큐에 추가한다. 구조는 (f,g,현재위치)이다.
```

[Fig. AStarSearch 클래스에서 구현한 A*search 알고리즘]

A Star Search 알고리즘은, Informed Search의 한 방법이다. Uninformed Seach인 UCS와 다르게 목표 위치에 대한 정보를 토대로 탐색하기 때문에, Heuristic Func이라는 함수를 사용하여 노드가 목표 노드로부터 얼마나 가까이 있는지를 측정하여 이용한다.

A Star Search의 동작방법은 F(n) = G(n) + H(n) 이 최소가 되는 곳으로 움직인다.

각각에 대해 설명하면,

F(n) = 해당 노드 n에 대한 가중치들의 합

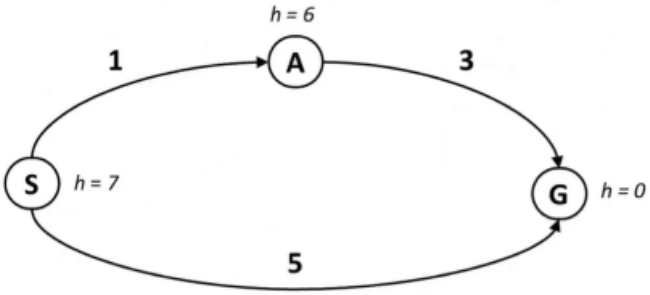
G(n) = 시작 노드에서 해당 노드까지 이동한 Cost (Backword Cost)

H(n) = 휴리스틱함수로, 해당 노드에서부터 도착지점까지의 예상 가중치이다. (Forword Cost)

이번 과제에서는 H(n)을 유클리드 거리로 정의하기로 하였다.

A Star Search는 Goal을 Enquene했을 때, Stop하지 않고, Dequene 했을 때 Stop한다.

항상 Optimal 한 Path를 찾는 UCS와 달리 이 알고리즘은 Optimal 하지 않다.



[Fig. 이 예제에서는 A * search 함수가 Optimal하지 않다.]

위의 그림과 마찬가지로 우리의 미로 문제에서도 **A* Search는 completenss하지만 Optimal하지 않다는 단점**이 있다.

아래 그림에서 볼 수 있듯이, **대체로 A star Search가 Uniformed Cost Search보다 Cost가 낮다.**

[Uniform Cost Search] **[A* Search]**

[Fig. Uniform Cost Search와 A*Search 비교]

따라서, 이 알고리즘은 UCS에 비해서,

Optimal하지 않을 수 있다는 단점이 존재하지만,

휴리스틱함수를 통해서 **UCS에 비해 대체로 더 적은 Cost로 목표를 찾는다는 장점이 있다.**

<시간 복잡도>

A* Search 알고리즘의 시간 복잡도는 $O(b^d)$이다. (b : 이동 가능한 방향의 개수, d : 목표 노드 까지의 최대 깊이) 이 코드에서는 우선순위 큐를 사용하므로 각 셀을 방문할 때마다, O(logn) 시간이 소요된다.

따라서, **시간복잡도는 $O(b^d * \log n)$** 이다.

<공간 복잡도>

Open List는 탐색을 해야하는 곳을 등록하는 곳으로 위의 코드에서는 `priority_queue = PriorityQueue()` 이다.

Closed List는 탐색이 완료된 곳을 등록하는 곳으로 위의 코드에서는 `path = list()` 이다.

미로 예제에서 closed list는 전체 노드의 셀 수와 동일할 수 있다.

따라서 이 알고리즘의 공간 복잡도는 O(N)이다. (N은 미로 안에 모든 셀의 수)

<Optimal 과 Completeness>

A*Search는 이 미로 예제에서 위에서 설명한 것과 **같이 Optimal하지 않다.**

반면 Completeness은 미로의 시작과 끝이 연결되어있다는 가정 하에 무조건 **Completeness하다.**

- **Optimal X**
- **Completeness O**

Ⅱ. 결과 및 분석

```
|-------------------------------------------------------------|
|----------------------Solution Cost Tabel--------------------|
|---++-----------------------------++-----------------------------|
|IDX||-------------COST----------- ||-------------TIME---------- |
|---++--------------+--------------++--------------+-------------|
|IDX|| Uniform_Cost | A_STAR_COST || Uniform_TIME | A_STAR_TIME |
|---++--------------+--------------++--------------+-------------|
|  1||          210 |          184 ||       0.0038 |      0.0057 |
|  2||           16 |           13 ||       0.0008 |      0.0010 |
|  3||          142 |          113 ||       0.0037 |      0.0060 |
|  4||          395 |          394 ||       0.0096 |      0.0092 |
|  5||          177 |          165 ||       0.0039 |      0.0052 |
|  6||          103 |           64 ||       0.0026 |      0.0034 |
|  7||           17 |           11 ||       0.0007 |      0.0008 |
|  8||          297 |          282 ||       0.0079 |      0.0108 |
|  9||          394 |          394 ||       0.0062 |      0.0109 |
| 10||           23 |           14 ||       0.0009 |      0.0009 |
|---++--------------+--------------++--------------+-------------|
|AVG||       177.40 |       163.40 ||       0.0040 |      0.0054 |
|---++--------------+--------------++--------------+-------------|
|-------------------------------------------------------------|

xidid@DESKTOP-KOT8OOT MINGW64 /c/WorkSpace/AI_HW/pymaze-master
```

[Fig. 미로 10개 UCS와 A* search 실행 결과]

위에 결과 표를 보았을 때 모든 경우에서, A Star Search 알고리즘이 Uniform Cost Search 알고리즘에 비해 더 적거나 같은 Cost를 소모하는 것을 확인할 수 있다. 하지만, 알고리즘이 동작하는 동안의 총 소모 시간을 보면 A Star Search의 시간이 더 많은 것을 확인할 수 있다.

이는, 휴리스틱 함수의 계산으로 인해 발생한 문제로,

```python
def heuristic(self, coor, exit_coor) :
    """
    목표 위치의 시작 위치와 비용. 휴리스틱 기능을 위해 유클리드 거리를 사용하십시오.
    Return = dist_to_target : (k_n, l_n) 과 END cell 까지의 거리
    """
    (k_n, l_n) = coor
    (k_end, l_end) = exit_coor
    dist_to_target = math.sqrt((k_n - k_end) ** 2 + (l_n - l_end) ** 2)
    return dist_to_target
```

[Fig. 사용한 휴리스틱 함수]

Math.sqrt를 반복적으로 호출하므로, 인해서 유클리드 거리를 계산하는 시간이 증가했고,

따라서, cost는 A star search 알고리즘이 Uniform Cost Search보다 적지만, 더 많은 시간을 소모하는 것이다.

개선 방법으로는, math.sqrt를 사용하지 않고, 정수 계산을 사용하여 대략적인 유클리드 거리를 사용하는 방식의 휴리스틱 알고리즘을 사용하여 개선할 수 있다.

하지만 이번 과제에서는 유클리드 거리를 사용하라 명시되어 있으므로 위와 같이 사용했다.

Ⅲ. 부가적으로 과제 수행을 위해 추가한 코드 설명

solve_2019310649.py : 커맨드에서 실행할 파일. Python examples/solve_2019310649.py

mysolver.py : Uniform Cost Search와 A Star Search 알고리즘이 구현된 파일

<maze.py>

```
26    def __init__(self, num_rows, num_cols, id=0, algorithm = "dfs_backtrack"):
27        """Creates a gird of Cell objects that are neighbors to each other.
28
29            Args:
30                    num_rows (int): The width of the maze, in cells
31                    num_cols (int): The height of the maze in cells
32                    id (id): An unique identifier
33
34        """
35        self.num_cols = num_cols
36        self.num_rows = num_rows
37        self.id = id
38        self.grid_size = num_rows*num_cols
39        self.entry_coor = self._pick_random_entry_exit(None)
40        self.exit_coor = self._pick_random_entry_exit(self.entry_coor)
41        self.generation_path = []
42        self.solution_cost1 = -1          #ADD Cost를 Return 하려고 추가.
43        self.solution_time1 = 0.0         #ADD uniform cost search 알고리즘의 소모 시간 저장을 위해 추가.
44        self.solution_path = None
45        self.solution_cost2 = -1          #ADD Cost를 Return 하려고 추가.
46        self.solution_time2 = 0.0         #ADD A* search 알고리즘의 소모 시간 저장을 위해 추가.
47        self.initial_grid = self.generate_grid()
48        self.grid = self.initial_grid
49        self.generate_maze(algorithm, (0, 0))
```

[Fig. cost, time을 반환하기 위해서 추가하였다.]

Maze 객체 내에서 각각의 알고리즘을 사용하여 만들어진 Cost와 Time(소모 시간)을 추가적으로 저장하기 위해서 위와 같이 추가하였다.

<solve_2019310649.py>

```
3   import os
4   import sys
5   sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))
```

[Fig. src path 문제로 인해서 추가한 코드]

과제 코드를 실행할 때, src PATH error가 떠서, 위의 줄을 추가하였다.

```
# Create the manager
manager = MazeManager()

# Add a 10x10 maze to the manager
maze1 = manager.add_maze(20, 20)

"""--------------------Uniform_Cost_Search Soltion--------------------"""
manager.solve_maze(maze1.id, "UniformCostSearch")

uniform_sol_costs.append(maze1.solution_cost1)
uniform_sol_times.append(maze1.solution_time1)
uniform_sol_paths.append(maze1.solution_path)

# Display the maze
manager.show_maze(maze1.id)

# solution 애니메이션 visualize
manager.show_solution_animation(maze1.id)

# 결과 show
manager.show_solution(maze1.id)
```

[Fig. Uniform Cost Search를 실행하는 코드]

위의 코드는 Uniform Cost Search를 실행하는 코드이다.
MazeManager() 클래스의 객체를 만들고,
add_maze함수를 통해 20 x 20의 미로를 만든다.
solve_maze함수를 통해서 어떤 알고리즘으로 maze를 풀지 정하고, 맨 처음에 설명한 각 알고리즘의 풀이로 넘어가여 실질적인 solve가 일어나는 곳이다.

나머지 3개의 함수 같은 경우에는 시각화를 위해 사용된다.

A Star Search 알고리즘도 같은 방식으로 진행되는데, 한 가지 중요한 차이는 다음을 추가했다는 것이다.

```
# Place a Cell object at each location in the grid
for i in range(maze1.num_rows):
    for j in range(maze1.num_cols):
        maze1.grid[i][j].visited=False

maze1.solution_path=None
```

[Fig. 각 셀들을 방문했다는 표식을 없에는 코드]

이 코드를 통해서 Uniform Cost Search를 했을 때 지나갔던 Cell들의 표식을 품으로써, 같은 Maze를 통해 다시 A Star Search 알고리즘을 수행할 수 있게 해준다.

<maze_manager.py>

```
133         elif method == "UniformCostSearch":                                 #ADD
134             solver = UniformCostSearch()
135             maze.solution_cost1, maze.solution_path, maze.solution_time1 = solver.uniform_cost_search(maze)
136         elif method == "AStarSearch":                                       #ADD
137             solver = AStarSearch()
138             maze.solution_cost2, maze.solution_path, maze.solution_time2 = solver.a_star_search(maze)
```

[Fig. solve_maze 함수에서 Uniform_Cost_Search와 A_Star_Search를 하기 위해 추가한 코드]

과제의 양식에 맞게,

1. 각각의 함수의 이름은 uniform_cost_search와 a_star_search로 하였다.
2. maze객체를 입력받는다.
3. Cost와 path를 반환한다.

부가적으로 알고리즘을 실행한 소모 시간을 반환하도록 하였는데, 이는 결과를 나타낼 때, 각 알고리즘에서 소요하는 시간을 print하기 위함이다. 각각은 maze 객체에 저장된다.

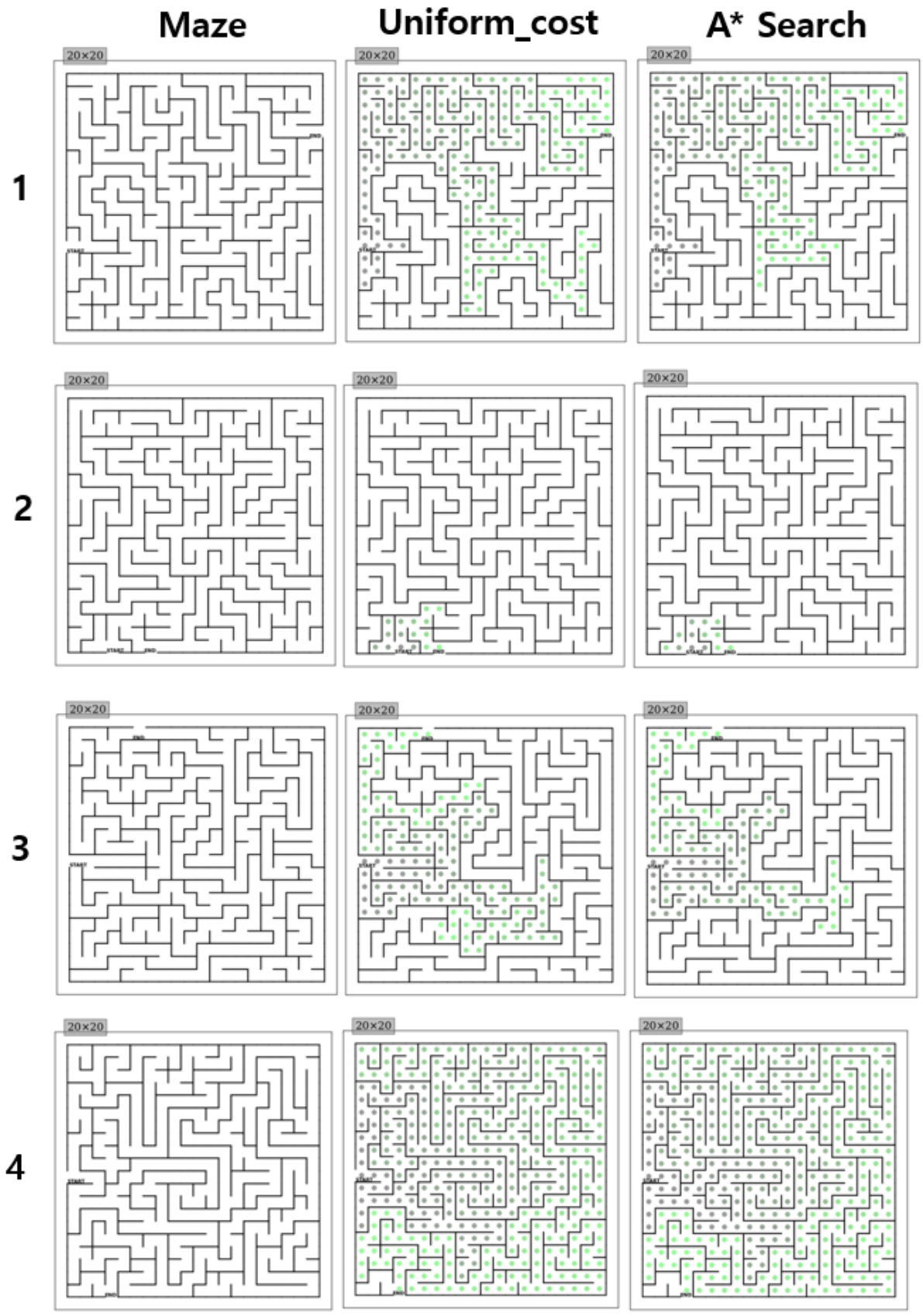
이를 이용하여 결과로

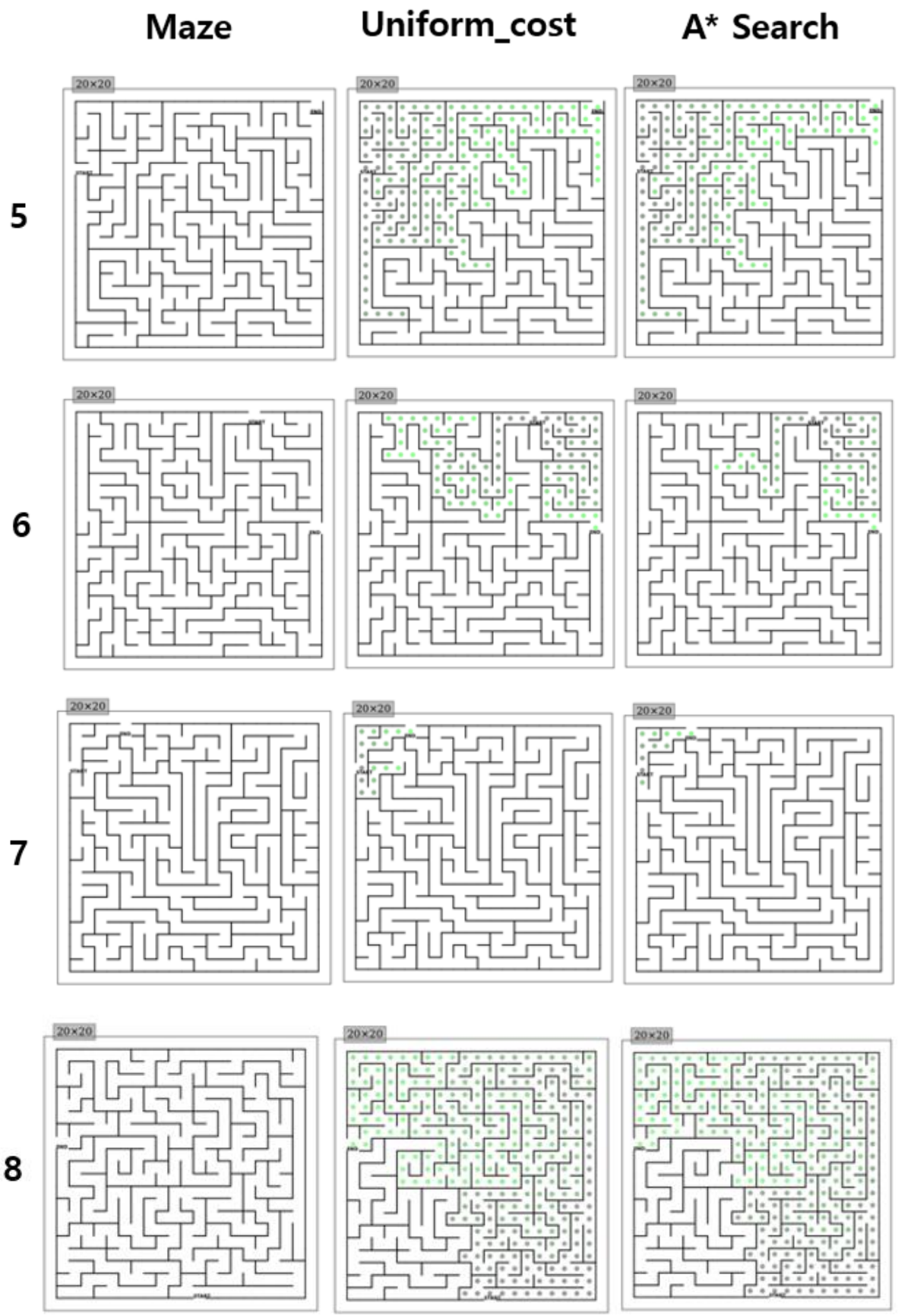
```
72      sum_uniform_sol_cost = 0
73      sum_a_star_sol_cost = 0
74      sum_uniform_sol_time = 0.0
75      sum_a_star_sol_time = 0.0
76
77
78      print("|-------------------------------------------------------------|")
79      print("|----------------------Solution Cost Tabel--------------------|")
80      print("|---++-----------------------------++-------------------------|")
81      print("|IDX||------------COST---------- ||------------TIME---------- |")
82      print("|---++-------------+-------------++-------------+-------------|")
83      print("|IDX|| Uniform_Cost | A_STAR_COST || Uniform_TIME | A_STAR_TIME |")
84      print("|---++-------------+-------------++-------------+-------------|")
85      for i in range(len(uniform_sol_costs)) :
86          print('|{:>3d}|| {:>12d} | {:>11d} || {:>12.4f} | {:>11.4f} |'.format(i+1,uniform_sol_costs[i], a_star_sol_costs[i],uniform_sol_times[i], a_star_sol_times[i]))
87          sum_uniform_sol_cost += uniform_sol_costs[i]
88          sum_a_star_sol_cost  += a_star_sol_costs[i]
89          sum_uniform_sol_time += uniform_sol_times[i]
90          sum_a_star_sol_time  += a_star_sol_times[i]
91      print("|---++-------------+-------------++-------------+-------------|")
92      avg_uniform_sol_cost = sum_uniform_sol_cost / len(uniform_sol_costs)
93      avg_a_star_sol_cost = sum_a_star_sol_cost/ len(uniform_sol_costs)
94      avg_uniform_sol_time = sum_uniform_sol_time/ len(uniform_sol_costs)
95      avg_a_star_sol_time = sum_a_star_sol_time/ len(uniform_sol_costs)
96      print('|AVG|| {:>12.2f} | {:>11.2f} || {:>12.4f} | {:>11.4f} |'.format(avg_uniform_sol_cost, avg_a_star_sol_cost,avg_uniform_sol_time, avg_a_star_sol_time))
97      print("|---++-------------+-------------++-------------+-------------|")
98      print("|-------------------------------------------------------------|")
```

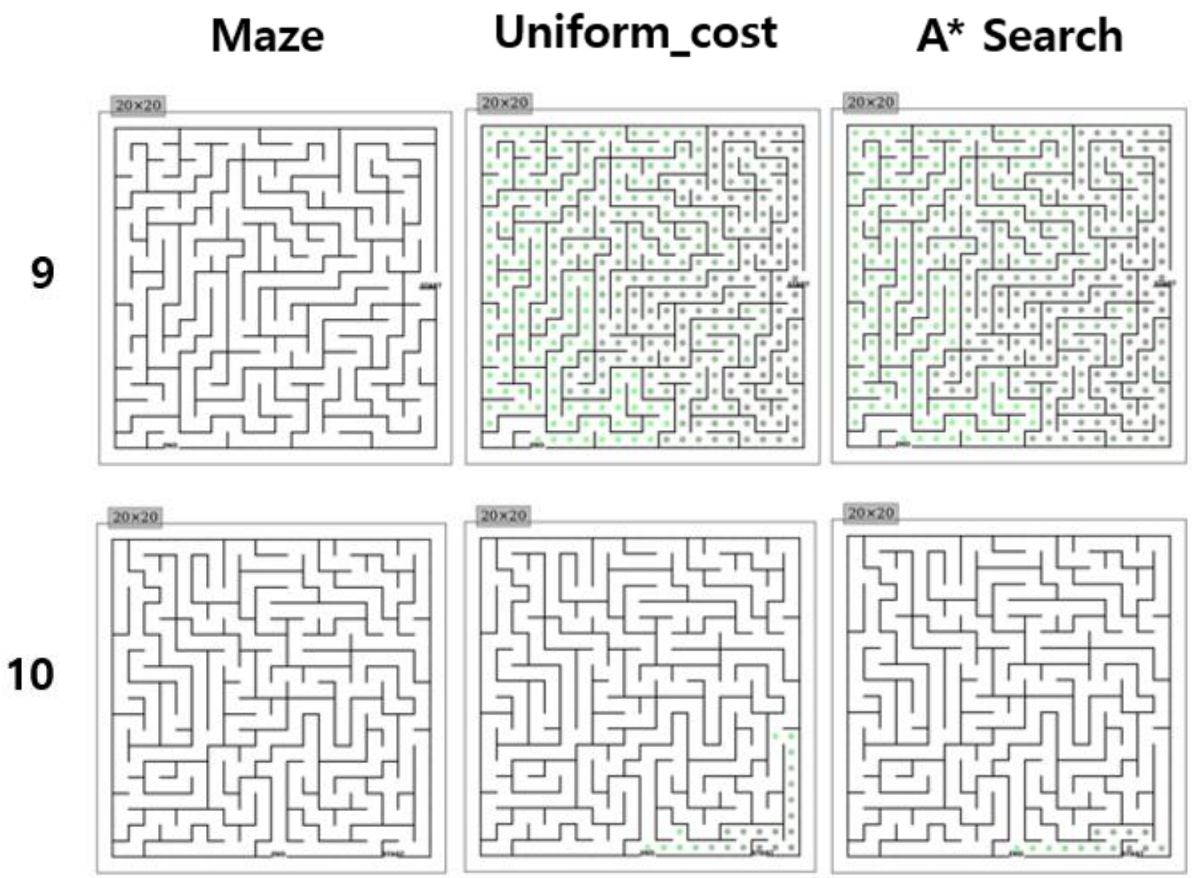
[Fig. sol_2019310649.py의 코드 중 일부로 결과를 출력한다]

총 10번의 결과를 출력하는데에 사용된다.

Ⅳ. 각 알고리즘의 시각화




Maze
Uniform_cost
A* Search
5
6
7
8
20×20


Maze
Uniform_cost
A* Search
9
10
20×20