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

Name : 임용성 (Lim Yong Sung)

Student ID [redacted]

[목차]

Ⅰ. Explanation and Implementation of Search Algorithm

■ Uniform_Cost_Search (at my.solver.py)

```python
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 indicied
                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 Algorithm Code]

The Uniform Cost Search algorithm is a method of Unformed Search. Uniform Cost Search gives each cell a cost and selects the next cell, that is, the next cell with the lowest cost in consideration of the cost so far.

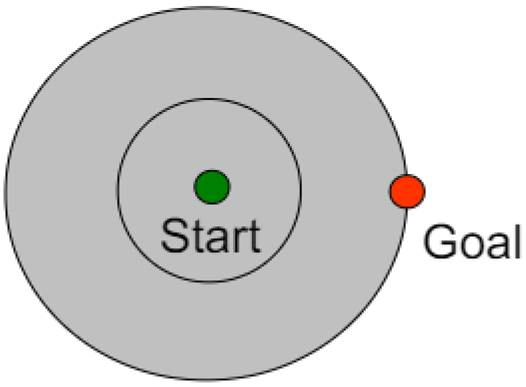
In this maze example, all the Cost required to move is fixed at 1.

It can be said that there is no difference between Uniform Cost Search and Breadth First Search.

This algorithm has the advantage of being completely and optimally.

Unformed Search has the **disadvantage of being slow** because it expands in 'all directions' as shown in the following figure.

Due to the characteristics of UCS, the cost is expanded in all directions with low cost, **so there is an advantage when the path is wide and not deep.**



[Fig. At our maze, UCS is the same as BFS, and has the disadvantage of expanding in 'all directions'.]

< Time Complexity >

The time complexity of the Uniform Cost Search algorithm with all Cost equal to 1 is O(b^d)

The time complexity is O($b^d * logn$)in the worst case, because we use priority queues (O(n) → O(log(n))), every time we move, we find the path of the least cost while checking all the elements in the queue.

< Space Complexity >

The Open List is a place to register where you need to search, and in the code above, it is as follows.

→
```
priority_queue = PriorityQueue()
```

The Closed List is a place to register where the search has been completed, and in the code above, it is as follows. →
```
path = list()
```

Thus, the pace complexity of this algorithm is O(N). (N is the number of all cells in the maze)

<Optimal and Completeness>

In this situation, Uniform Cost is complete because all moving costs are 1, Uniform Cost Search can find Optimal Path, and completeness is complete because 'N' is finite.

- **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. A*search Algorithm Implemented in the AstarSearch Class]

The A Star Search algorithm is a method of Informed Search. Unlike UCS, which is a Unformed Seach, we use a function called Heuristic Func to measure and use how close a node is to the target **node because it searches based on information about the target location**.

The way A* Search works is to move where **F(n) = G(n) + H(n)** is minimized.

To explain each of them,

F(n) = the sum of the weights for the corresponding node n

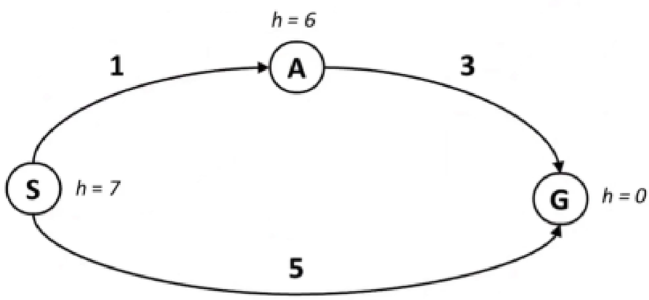
G(n) = Cost moved from the start node to that node (Backword Cost)

H(n) = a heuristic function, which is the expected weight from the corresponding node to the point of arrival. (Forword Cost)

In this task, H(n) will be defined as the Euclidean distance.

A Star Search does not stop when Goal is Enquired, but stops when it is Dequene.

Unlike UCS, which always finds an Optimal Path, **this algorithm is not Optimal.**



[Fig. In this example, the A * search function is not Optimal.]

Like the above figure, A* Search is completeness but it has the disadvantage of Optimal in our maze problem.

As can be seen in the figure below, A* Search generally has a **lower cost** than Uniform Cost Search.

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

[Fig. Uniform Cost Search와 A*Search Comparison]'

Therefore, this algorithm has the disadvantage that it may not be optimal compared to UCS, but it has the advantage of finding a target with generally less Cost than UCS through heuristic functions.

< Time Complexity >

The time complexity of the A* Search algorithm is **O(b^d).** (b: number of movable directions, d: maximum depth to target node) This code uses priority queues, so each cell visits **O(logn)** time.

Thus, the time complexity of A* Search is O(b^d * logn)

< Space Complexity >

The Open List is a place to register where you need to search, and in the code above, it is as follows.

→
```
priority_queue = PriorityQueue()
```

The Closed List is a place to register where the search has been completed, and in the code above, it is as follows.

→
```
path = list()
```

In the maze example, the closed list may be equal to the number of cells of the entire node.

Thus, the spatial complexity of this algorithm is O(N). (N is the number of all cells in the maze)

<Optimal and Completeness>

A*Search is not as Optimal as described above in this maze example.

On the other hand, Completeness is always established under the assumption that the beginning and the end of the maze are connected.

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

Ⅱ. Results and Analysis

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. Results of 10 Maze UCS and A* search runs]

When looking at the results table above, it can be seen that in all cases, the A* Search algorithm consumes less or the same Cost compared to the Uniform Cost Search algorithm. However, if you look at the total time spent while the algorithm is operating, you can see that A* Search has more time.

This is a problem caused by the calculation of the heuristic function. let's look at the code below.

```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. Heuristic function used by A* Search algorithm]

Math.sqrt is called repeatedly, which increases the time to calculate the Euclidean distance.

Therefore, cost is that although the A* search algorithm is less than Uniform Cost Search, it takes more time.

As an improvement, Math.It can be improved by using a heuristic algorithm that uses approximate Euclidean distances using integer calculations without using sqrt.

However, in this task, it is specified to use the Euclidean distance, so we used it as above.

Ⅲ. Explanation of code added for additional task performance

solve_2019310649.py **: The file to run in the command.** **Python examples/solve_2019310649.py**

mysolver.py **: Files that implement Uniform Cost Search and A* Search algorithms**

<maze.py>

```python
    def __init__(self, num_rows, num_cols, id=0, algorithm = "dfs_backtrack"):
        """Creates a gird of Cell objects that are neighbors to each other.

            Args:
                    num_rows (int): The width of the maze, in cells
                    num_cols (int): The height of the maze in cells
                    id (id): An unique identifier

        """
        self.num_cols = num_cols
        self.num_rows = num_rows
        self.id = id
        self.grid_size = num_rows*num_cols
        self.entry_coor = self._pick_random_entry_exit(None)
        self.exit_coor = self._pick_random_entry_exit(self.entry_coor)
        self.generation_path = []
        self.solution_cost1 = -1        #ADD Cost를 Return 하려고 추가.
        self.solution_time1 = 0.0       #ADD uniform cost search 알고리즘의 소모 시간 저장을 위해 추가.
        self.solution_path = None
        self.solution_cost2 = -1        #ADD Cost를 Return 하려고 추가.
        self.solution_time2 = 0.0       #ADD A* search 알고리즘의 소모 시간 저장을 위해 추가.
        self.initial_grid = self.generate_grid()
        self.grid = self.initial_grid
        self.generate_maze(algorithm, (0, 0))
```

[Fig. Added to return cost, time]

Cost and Time (consumption time) created using each algorithm within the Maze object were added as above to additionally store.

<solve_2019310649.py>

```python
import os
import sys
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))
```

[Fig. Code added due to src path problem]

When executing the task code, an 'src PATH error' appeared, and the line above was added.

```python
        # 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. Code running Uniform Cost Search]

The above code is the code that executes Uniform Cost Search.

1. Creates an object in the MazeManager() class.
2. The add_maze function creates a maze of 20 x 20.
3. Through the solve_maze function, we determine which algorithm to solve the maze, and move on to the solution of each algorithm described at the beginning to solve the maze.

The other three functions are used for visualization.

The A Star Search algorithm works the same way, and one important difference is that it added:

```python
        # 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. Code that removes the mark of visiting each cell.]

This code removes the marks of cells that passed when doing Uniform Cost Search, allowing the A* Search algorithm to be performed again through the same Maze.

<maze_manager.py>

```
    elif method == "UniformCostSearch":                       #ADD
        solver = UniformCostSearch()
        maze.solution_cost1, maze.solution_path, maze.solution_time1 = solver.uniform_cost_search(maze)
    elif method == "AStarSearch":                             #ADD
        solver = AStarSearch()
        maze.solution_cost2, maze.solution_path, maze.solution_time2 = solver.a_star_search(maze)
```

[Fig. Code added to do Uniform_Cost_Search and A_Star_Search in the solve_maze function]

According to the form of the assignment,

1. Each function was named **uniform_cost_search** and **a_star_search**.
2. Get the maze object input.
3. Return Cost and path.

In addition, the time consumed by executing the algorithm was returned to print the time taken by each algorithm when displaying the result. Each is stored in a maze object.

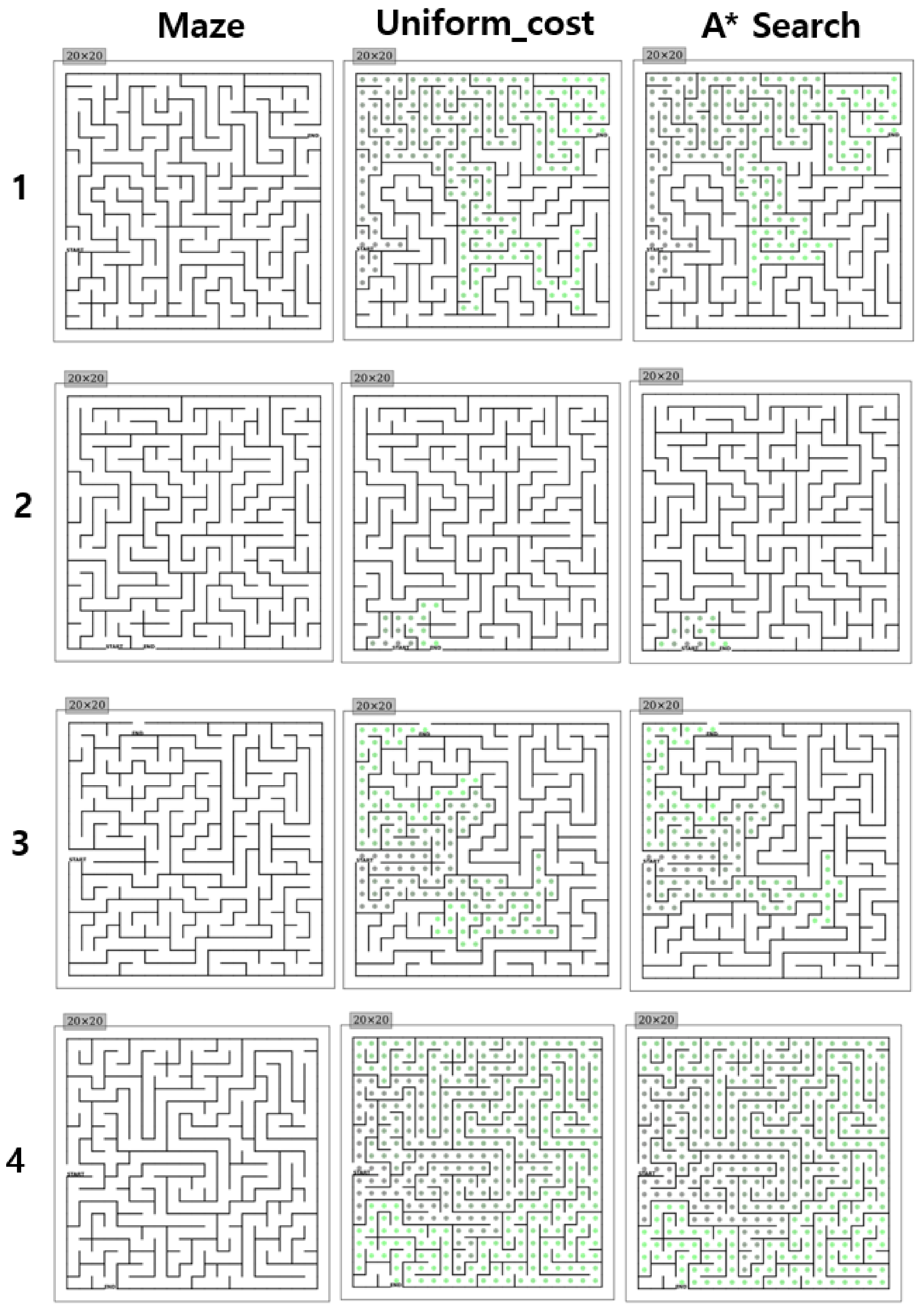
Using this, the results are as follows.

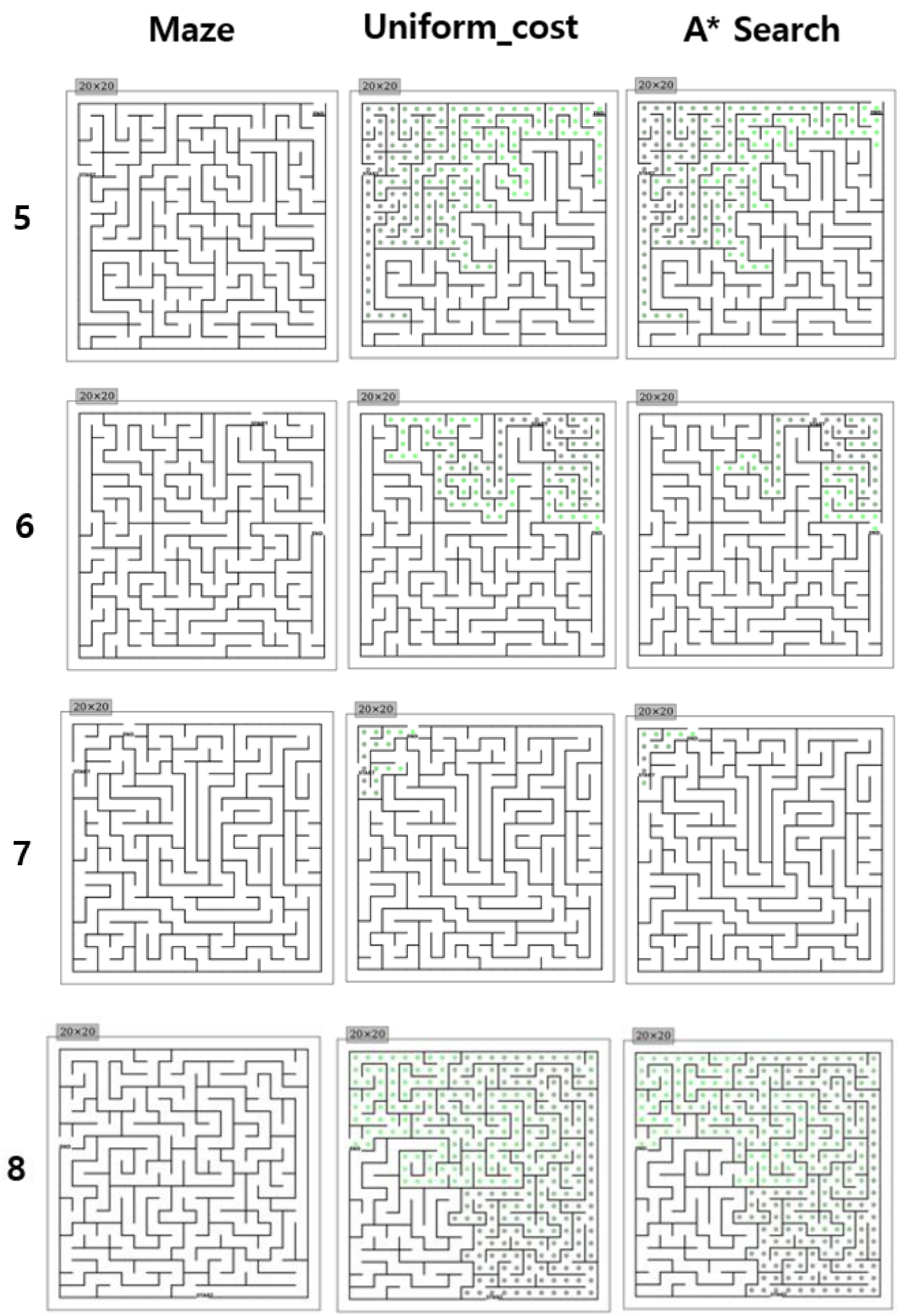
```
sum_uniform_sol_cost = 0
sum_a_star_sol_cost = 0
sum_uniform_sol_time = 0.0
sum_a_star_sol_time = 0.0


print("|-------------------------------------------------------------|")
print("|----------------------Solution Cost Tabel--------------------|")
print("|---++--------------------------++---------------------------|")
print("|IDX||-------------COST---------- ||-------------TIME---------- |")
print("|---++--------------+-------------++---------------+-------------|")
print("|IDX|| Uniform_Cost | A_STAR_COST || Uniform_TIME | A_STAR_TIME |")
print("|---++--------------+-------------++---------------+-------------|")
for i in range(len(uniform_sol_costs)) :
    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]))
    sum_uniform_sol_cost += uniform_sol_costs[i]
    sum_a_star_sol_cost  += a_star_sol_costs[i]
    sum_uniform_sol_time += uniform_sol_times[i]
    sum_a_star_sol_time  += a_star_sol_times[i]
print("|---++--------------+-------------++---------------+-------------|")
avg_uniform_sol_cost = sum_uniform_sol_cost / len(uniform_sol_costs)
avg_a_star_sol_cost = sum_a_star_sol_cost/ len(uniform_sol_costs)
avg_uniform_sol_time = sum_uniform_sol_time/ len(uniform_sol_costs)
avg_a_star_sol_time = sum_a_star_sol_time/ len(uniform_sol_costs)
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))
print("|---++--------------+-------------++---------------+-------------|")
print("|-------------------------------------------------------------|")
```

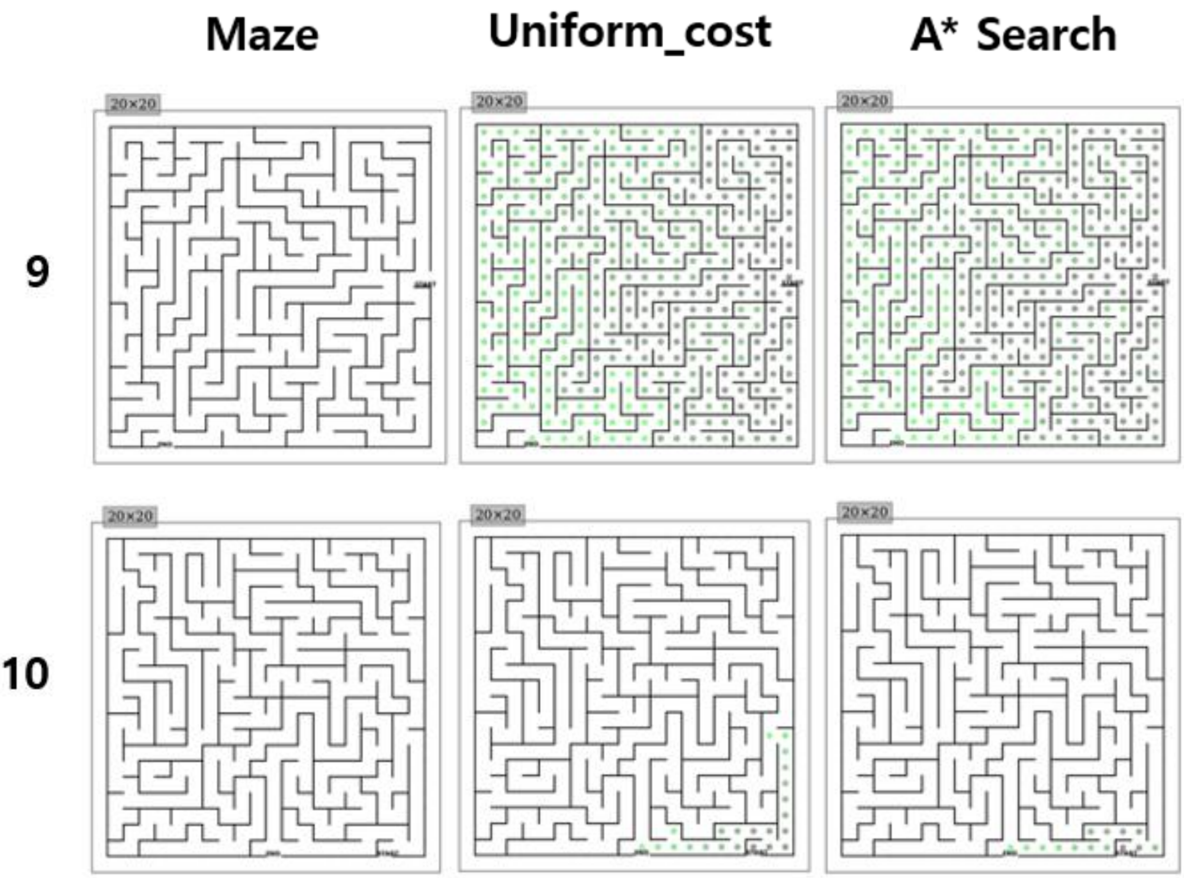
[Fig. Outputs the result as part of the code in sol_2019310649.py.]

This code is used to output a total of 10 results.

IV. Visualization of each algorithm




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


Maze
Uniform_cost
A* Search
9
10
20×20