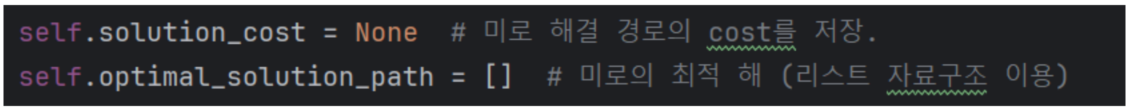
HW #1

Name: 이세형

Student Number: 2020312145

1. Implementations:

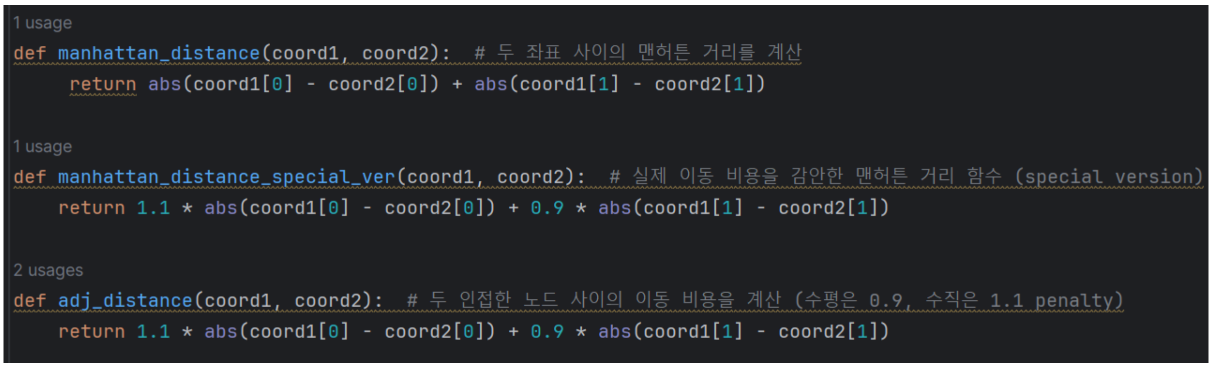
First, I added new two fields to class maze in maze.py.



```
self.solution_cost = None  # 미로 해결 경로의 cost를 저장.
self.optimal_solution_path = []  # 미로의 최적 해 (리스트 자료구조 이용)
```

'solution_cost' filed stores the cost of optimal solution. And 'optimal_solution_path' field stores optimal solution path of maze.

'manhattan_distance', 'manhattan_distance_special_ver', 'adj_distance' functions are also implemented in maze_manager.py



```
1 usage
def manhattan_distance(coord1, coord2):  # 두 좌표 사이의 맨허튼 거리를 계산
     return abs(coord1[0] - coord2[0]) + abs(coord1[1] - coord2[1])

1 usage
def manhattan_distance_special_ver(coord1, coord2):  # 실제 이동 비용을 감안한 맨허튼 거리 함수 (special version)
    return 1.1 * abs(coord1[0] - coord2[0]) + 0.9 * abs(coord1[1] - coord2[1])

2 usages
def adj_distance(coord1, coord2):  # 두 인접한 노드 사이의 이동 비용을 계산 (수평은 0.9, 수직은 1.1 penalty)
    return 1.1 * abs(coord1[0] - coord2[0]) + 0.9 * abs(coord1[1] - coord2[1])
```

The 'manhattan_distance' function returns the Manhattan distance between the two coordinates passed to it.
The 'manhattan_distance_special_ver' function returns the Manhattan distance between the two coordinates passed to it, taking into account that the cost of horizontal (0.9) and vertical (1.1) movements are different.
The 'adj_distance' function calculates and returns the movement distance cost between two adjacent nodes.

Detailed comments have been added to most lines of this code.

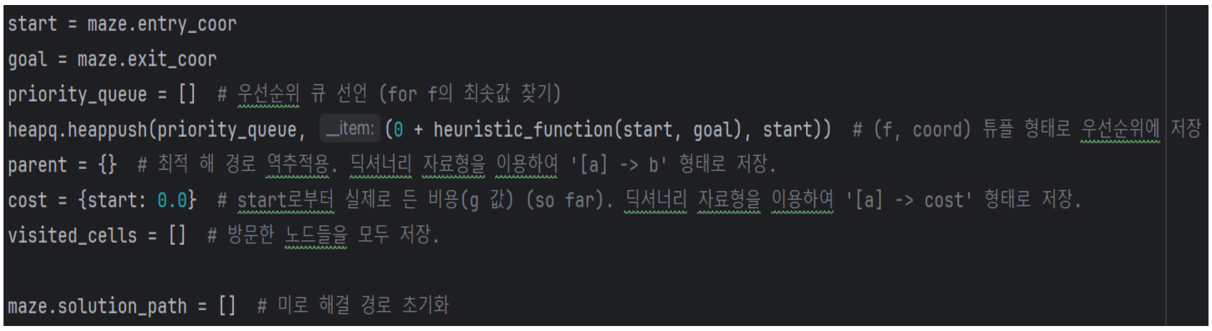
Below is detailed explanation about implementations of two functions.

a) A * search algorithm:

The 'a_star_search' function takes a maze instance and a heuristic function as inputs and returns the path and cost of the optimal solution. Below is a simplified pseudocode of the function.

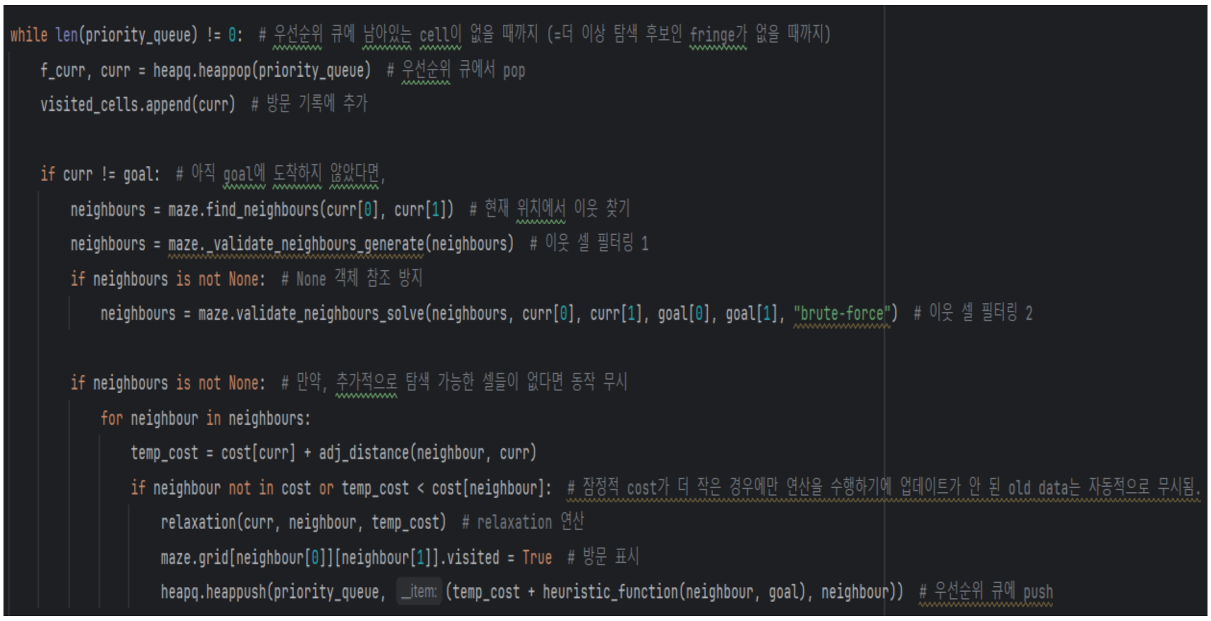
```
def a_star_search(maze, heristic_func):
    initialize varables

    while(Q is not empty) do:
        curr_node = Q.pop()
        if curr_node != goal:
            for each neighbour in curr_node:
                if tentative_cost < cost[neighbour]:
                    relaxation()
                    neighbour.visited = true
                    Q.push((g_value + h_value, neighbour))

        else:
            track optimal_solution_path with parent
            return optimal_solution_path, cost[goal]

    return None, -1
```

The local variables declared within the 'a_star_search' function are as follows.



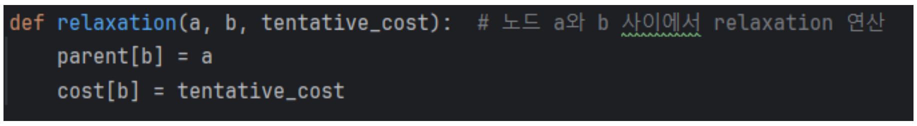
```
start = maze.entry_coor
goal = maze.exit_coor
priority_queue = []  # 우선순위 큐 선언 (for f의 최솟값 찾기)
heapq.heappush(priority_queue, _item: (0 + heuristic_function(start, goal), start))  # (f, coord) 튜플 형태로 우선순위에 저장
parent = {}  # 최적 해 경로 역추적용. 딕셔너리 자료형을 이용하여 '[a] -> b' 형태로 저장.
cost = {start: 0.0}  # start로부터 실제로 든 비용(g 값) (so far). 딕셔너리 자료형을 이용하여 '[a] -> cost' 형태로 저장.
visited_cells = []  # 방문한 노드들을 모두 저장.

maze.solution_path = []  # 미로 해결 경로 초기화
```

The 'priority_queue' variable is a priority queue that stores tuples in the form of (f value, coordinate). (cf. f value = g value + h value) The 'parent' variable is a dictionary data structure that stores information about the parent node of a given node. This variable is later used for backtracking the shortest path. 'Cost' is a data structure that stores the shortest path cost calculated so far from the start node to the given node. The 'visited_cells' variable is a dictionary data structure that records the visit history of nodes.

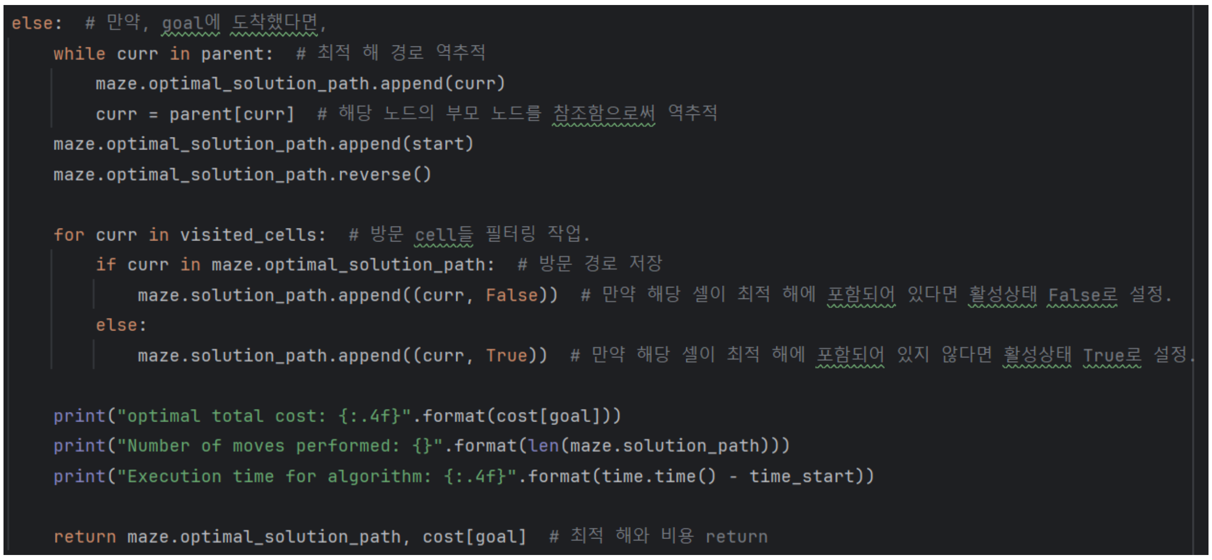


```
while len(priority_queue) != 0:  # 우선순위 큐에 남아있는 cell이 없을 때까지 (=더 이상 탐색 후보인 fringe가 없을 때까지)
    f_curr, curr = heapq.heappop(priority_queue)  # 우선순위 큐에서 pop
    visited_cells.append(curr)  # 방문 기록에 추가

    if curr != goal:  # 아직 goal에 도착하지 않았다면,
        neighbours = maze.find_neighbours(curr[0], curr[1])  # 현재 위치에서 이웃 찾기
        neighbours = maze._validate_neighbours_generate(neighbours)  # 이웃 셀 필터링 1
        if neighbours is not None:  # None 객체 참조 방지
            neighbours = maze.validate_neighbours_solve(neighbours, curr[0], curr[1], goal[0], goal[1], "brute-force")  # 이웃 셀 필터링 2

        if neighbours is not None:  # 만약, 추가적으로 탐색 가능한 셀들이 없다면 동작 무시
            for neighbour in neighbours:
                temp_cost = cost[curr] + adj_distance(neighbour, curr)
                if neighbour not in cost or temp_cost < cost[neighbour]:  # 잠정적 cost가 더 작은 경우에만 연산을 수행하기에 업데이트가 안 된 old data는 자동적으로 무시됨.
                    relaxation(curr, neighbour, temp_cost)  # relaxation 연산
                    maze.grid[neighbour[0]][neighbour[1]].visited = True  # 방문 표시
                    heapq.heappush(priority_queue, item: (temp_cost + heuristic_function(neighbour, goal), neighbour))  # 우선순위 큐에 push
```

As mentioned in the pseudocode above, the search continues through neighboring nodes and performs relaxation operations until there are no more candidates in the 'priority_queue'. For the relaxation operation, a separate function was defined within the 'a_star_search' function as follows. In this process, the 'manhattan_distance' function was used as the heuristic function. (It will be explained in detail later, but experiments were also conducted using the 'manhattan_distance_special_ver' function, which reflects the actual movement cost.)

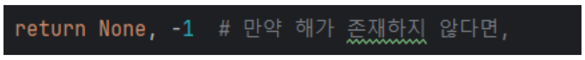


```
def relaxation(a, b, tentative_cost):  # 노드 a와 b 사이에서 relaxation 연산
    parent[b] = a
    cost[b] = tentative_cost
```



```
else:  # 만약, goal에 도착했다면,
    while curr in parent:  # 최적 해 경로 역추적
        maze.optimal_solution_path.append(curr)
        curr = parent[curr]  # 해당 노드의 부모 노드를 참조함으로써 역추적
    maze.optimal_solution_path.append(start)
    maze.optimal_solution_path.reverse()

    for curr in visited_cells:  # 방문 cell들 필터링 작업.
        if curr in maze.optimal_solution_path:  # 방문 경로 저장
            maze.solution_path.append((curr, False))  # 만약 해당 셀이 최적 해에 포함되어 있다면 활성상태 False로 설정.
        else:
            maze.solution_path.append((curr, True))  # 만약 해당 셀이 최적 해에 포함되어 있지 않다면 활성상태 True로 설정.

    print("optimal total cost: {:.4f}".format(cost[goal]))
    print("Number of moves performed: {}".format(len(maze.solution_path)))
    print("Execution time for algorithm: {:.4f}".format(time.time() - time_start))

    return maze.optimal_solution_path, cost[goal]  # 최적 해와 비용 return
```

If the goal node is reached, the search is terminated and the optimal solution is backtracked using the 'parent' and 'visited_cells' variables. Cells that are included in the optimal solution are stored in the maze's 'solution_path' member variable with an active state set to 'False', while those not included are stored with an active state set to 'True'. Thus, the 'a_star_search' function returns the optimal solution of the maze and its cost. If there is no solution, it returns 'None' and '-1', respectively.



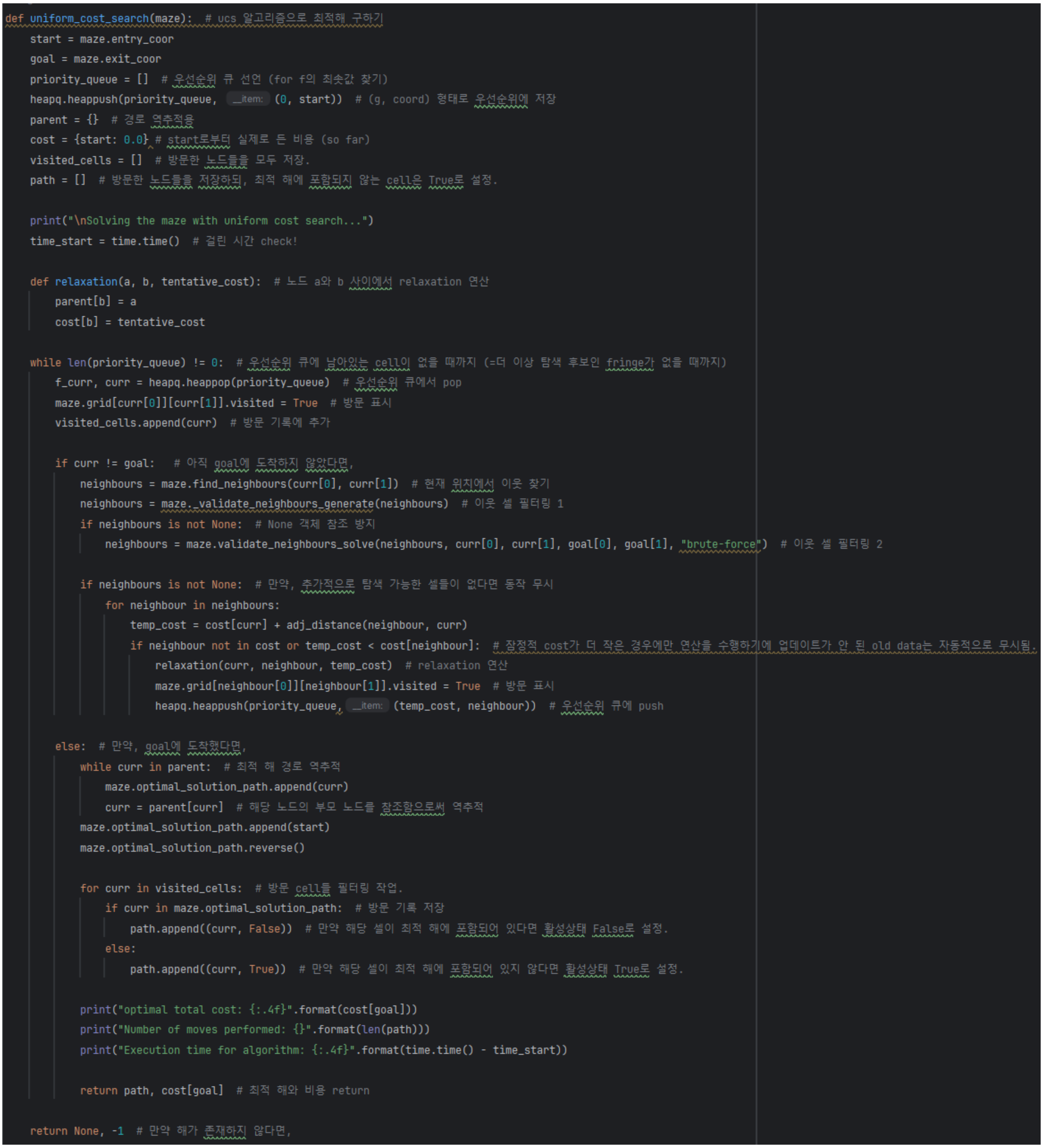
```python
return None, -1  # 만약 해가 존재하지 않다면,
```

b) Uniform Cost search algorithm:

The 'uniform_cost_search' function takes a maze instance and returns the solution path and cost. During this process, information about the optimal solution path is also stored in the maze's 'optimal_solution_cost' member variable. The pseudocode is as follows.

```
def uniform_cost_search(maze):
    initialize varables

    while(Q is not empty) do:
        curr_node = Q.pop()
        if curr_node != goal:
            for each neighbour in curr_node:
                if tentative_cost < cost[neighbour]:
                    relaxation()
                    neighbour.visited = true
                    Q.push((g_value, neighbour))

        else:
            track optimal_solution_path with parent
            return optimal_solution_path, cost[goal]

    return None, -1
```

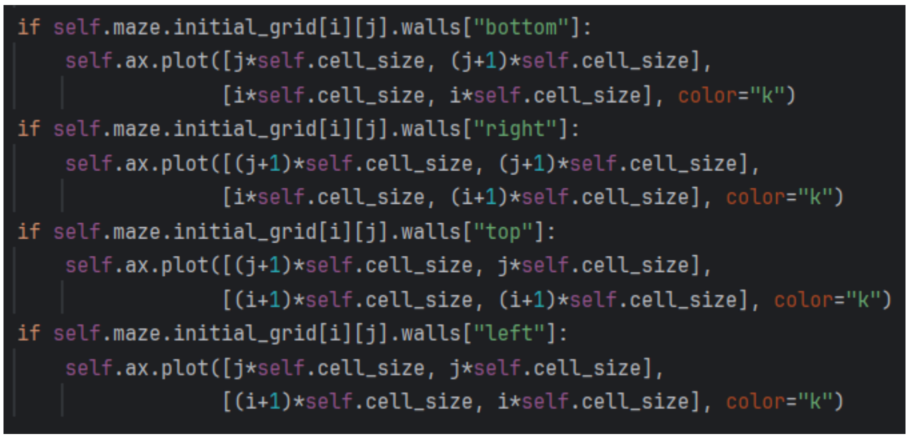
Since the 'uniform_cost_search' function, unlike the 'a_star_search' function, does not use a heuristic function, the code is almost identical, and thus a detailed explanation is omitted.



```python
def uniform_cost_search(maze):  # ucs 알고리즘으로 최적해 구하기
    start = maze.entry_coor
    goal = maze.exit_coor
    priority_queue = []  # 우선순위 큐 선언 (for f의 최솟값 찾기)
    heapq.heappush(priority_queue, __item: (0, start))  # (g, coord) 형태로 우선순위에 저장
    parent = {}  # 경로 역추적용
    cost = {start: 0.0} # start로부터 실제로 든 비용 (so far)
    visited_cells = []  # 방문한 노드들을 모두 저장.
    path = []  # 방문한 노드들을 저장하되, 최적 해에 포함되지 않는 cell은 True로 설정.

    print("\nSolving the maze with uniform cost search...")
    time_start = time.time()  # 걸린 시간 check!

    def relaxation(a, b, tentative_cost):  # 노드 a와 b 사이에서 relaxation 연산
        parent[b] = a
        cost[b] = tentative_cost

    while len(priority_queue) != 0:  # 우선순위 큐에 남아있는 cell이 없을 때까지 (=더 이상 탐색 후보인 fringe가 없을 때까지)
        f_curr, curr = heapq.heappop(priority_queue)  # 우선순위 큐에서 pop
        maze.grid[curr[0]][curr[1]].visited = True  # 방문 표시
        visited_cells.append(curr)  # 방문 기록에 추가

        if curr != goal:   # 아직 goal에 도착하지 않았다면,
            neighbours = maze.find_neighbours(curr[0], curr[1])  # 현재 위치에서 이웃 찾기
            neighbours = maze._validate_neighbours_generate(neighbours)  # 이웃 셀 필터링 1
            if neighbours is not None:  # None 객체 참조 방지
                neighbours = maze.validate_neighbours_solve(neighbours, curr[0], curr[1], goal[0], goal[1], "brute-force")  # 이웃 셀 필터링 2

            if neighbours is not None:  # 만약, 추가적으로 탐색 가능한 셀들이 없다면 동작 무시
                for neighbour in neighbours:
                    temp_cost = cost[curr] + adj_distance(neighbour, curr)
                    if neighbour not in cost or temp_cost < cost[neighbour]:  # 잠정적 cost가 더 작은 경우에만 연산을 수행하기에 업데이트가 안 된 old data는 자동적으로 무시됨.
                        relaxation(curr, neighbour, temp_cost)  # relaxation 연산
                        maze.grid[neighbour[0]][neighbour[1]].visited = True  # 방문 표시
                        heapq.heappush(priority_queue, __item: (temp_cost, neighbour))  # 우선순위 큐에 push

        else:  # 만약, goal에 도착했다면,
            while curr in parent:  # 최적 해 경로 역추적
                maze.optimal_solution_path.append(curr)
                curr = parent[curr]  # 해당 노드의 부모 노드를 참조함으로써 역추적
            maze.optimal_solution_path.append(start)
            maze.optimal_solution_path.reverse()

            for curr in visited_cells:  # 방문 cell들 필터링 작업.
                if curr in maze.optimal_solution_path:  # 방문 기록 저장
                    path.append((curr, False))  # 만약 해당 셀이 최적 해에 포함되어 있다면 활성상태 False로 설정.
                else:
                    path.append((curr, True))  # 만약 해당 셀이 최적 해에 포함되어 있지 않다면 활성상태 True로 설정.

            print("optimal total cost: {:.4f}".format(cost[goal]))
            print("Number of moves performed: {}".format(len(path)))
            print("Execution time for algorithm: {:.4f}".format(time.time() - time_start))

            return path, cost[goal]  # 최적 해와 비용 return

    return None, -1  # 만약 해가 존재하지 않다면,
```

2. Modifications to the existing codebase:

Due to 'top' and 'bottom' being reversed in the existing code base, slight modifications were made as follows. However, even without these modifications, the experiment results are not affected.



```python
if self.maze.initial_grid[i][j].walls["bottom"]:
    self.ax.plot([j*self.cell_size, (j+1)*self.cell_size],
                 [i*self.cell_size, i*self.cell_size], color="k")
if self.maze.initial_grid[i][j].walls["right"]:
    self.ax.plot([(j+1)*self.cell_size, (j+1)*self.cell_size],
                 [i*self.cell_size, (i+1)*self.cell_size], color="k")
if self.maze.initial_grid[i][j].walls["top"]:
    self.ax.plot([(j+1)*self.cell_size, j*self.cell_size],
                 [(i+1)*self.cell_size, (i+1)*self.cell_size], color="k")
if self.maze.initial_grid[i][j].walls["left"]:
    self.ax.plot([j*self.cell_size, j*self.cell_size],
                 [(i+1)*self.cell_size, i*self.cell_size], color="k")
```

3. Analysis:

I conducted experiments using a 20 * 20 maze. For comparison purposes, I fixed the seed value for maze.py to 1 and the seed value for algorithm.py to 0. I tested the A* search algorithm by running the tests/a_star_search_test.py file and the uniform cost search algorithm by running the tests/uniform_cost_search_test.py file.

(As I will mention later, I also conducted additional experiments by applying an improved heuristic function and running the tests/a_star_search_special_version_test.py file.)

The images below are the results of the experiments.

The red cells indicate the optimal solution path, and the yellow cells represent cells that were visited but are not part of the optimal solution.

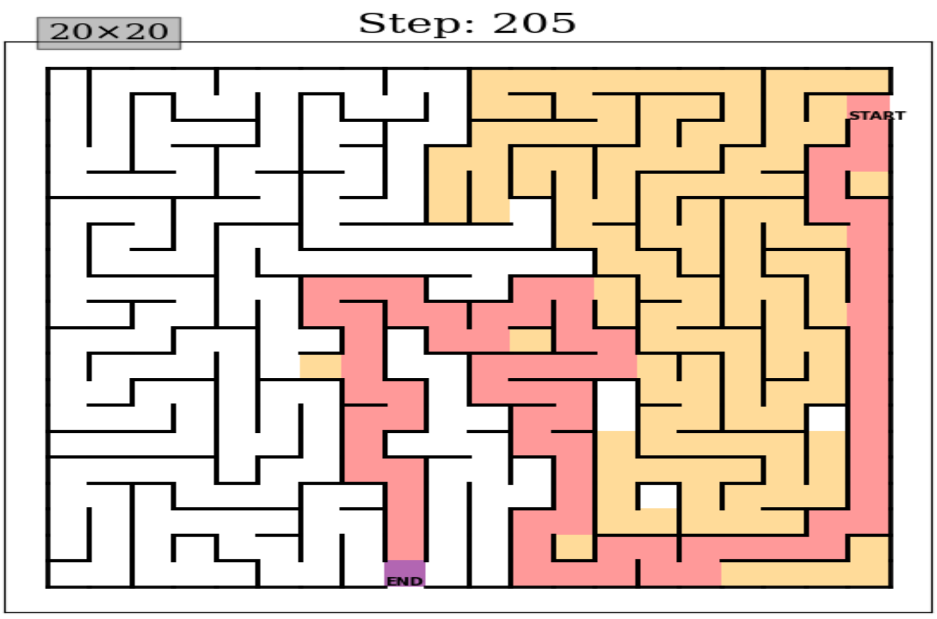


Figure 1. a * search algorithm result (animation)

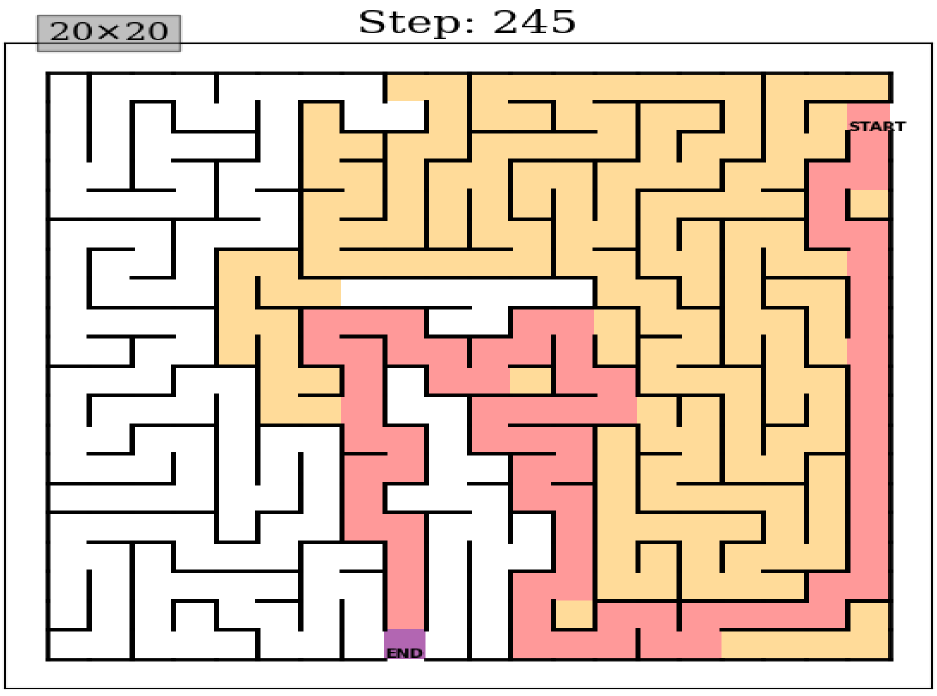
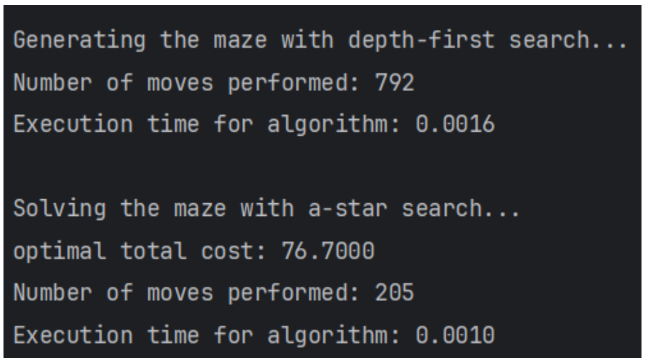
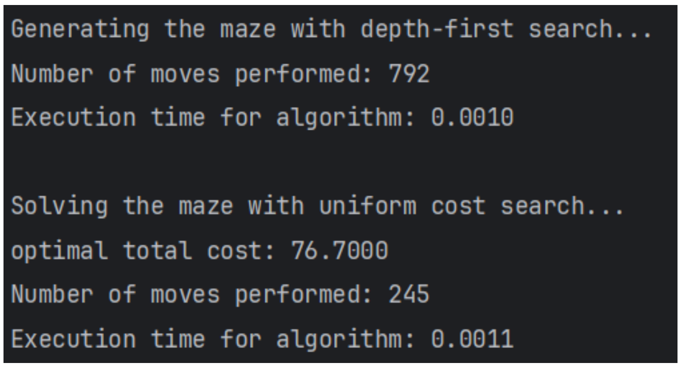


Figure 2. uniform cost search algorithm result (animation)



```
Generating the maze with depth-first search...
Number of moves performed: 792
Execution time for algorithm: 0.0016

Solving the maze with a-star search...
optimal total cost: 76.7000
Number of moves performed: 205
Execution time for algorithm: 0.0010
```

Figure 3. a * search algorithm result (terminal)



```
Generating the maze with depth-first search...
Number of moves performed: 792
Execution time for algorithm: 0.0010

Solving the maze with uniform cost search...
optimal total cost: 76.7000
Number of moves performed: 245
Execution time for algorithm: 0.0011
```

Figure 4. uniform cost search algorithm result (terminal)

a) Strengths:

- **A * search algorithm:**

  By using a heuristic function that calculates the forward cost, the search can be conducted at a lower cost than with the uniform cost search algorithm or the BFS algorithm. In the results above, the A* search algorithm visited 40 fewer cells than the uniform cost search algorithm.

- **Uniform cost search (UCS)algorithm:**

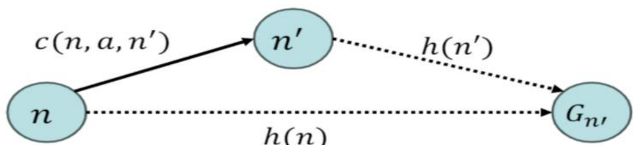
  Uniform cost search guarantees an optimal solution because it conducts the search through the Dijkstra algorithm, which finds the shortest path. If the moving cost between all cells (or nodes) is the same, using the BFS algorithm with a simple queue would yield better performance.

b) Weaknesses:

- **A * search algorithm:**

  The dependency on the heuristic function is high. That is, depending on how the heuristic function is designed, it may not guarantee an optimal solution. The existing 'a_star_search' function uses the 'manhattan_distance' function, which does not satisfy the consistency condition, and may not find the optimal solution for a given maze. "Below is the definition of the consistency condition and a proof that the existing 'manhattan_distance' function does not satisfy this consistency condition.

$$h(n) \leq c(n, a, n') + h(n') .$$



$pf)$ For example, consitency is not satisfied for two cells (n_{i-1}, n_i)that are horizontally adjacent:

$$h(n_i) - h(n_{i-1}) = 1 > c(n_i, a, n_{i-1}) = 0.9$$

Nevertheless, in the experiment, since the maze size was small (20 * 20) and the actual distance costs (1.1, 0.9) did not significantly differ, it succeeded in finding the optimal solution.

(I conducted additional experiments using a heuristic function that addresses these weaknesses. I will mention this later in the report.)

Additionally, since the heuristic value is calculated based on the distance to the goal, if the solution to the maze involves a detour rather than a straight path to the goal node, it would require a significant search cost.

- **Uniform cost search (UCS) algorithm:**

  The uniform cost search algorithm guarantees an optimal solution in any situation, but since it only considers the backward cost $(= g\ value)$ without taking the forward cost $(= h\ value)$ into account, it searches through more cells, thereby incurring higher search costs.

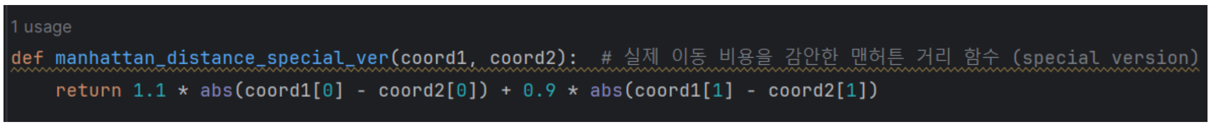
c) Improvements:

As I mentioned before, I also implemented and experimented with a Manhattan distance function ('manhattan_distance_special_ver') that reflects the actual distance costs (1.1, 0.9). This function satisfies the consistency condition, thereby guaranteeing an optimal solution.

$pf)$ Consitency is satisfied for arbitrary two cells (n_i, n_j):

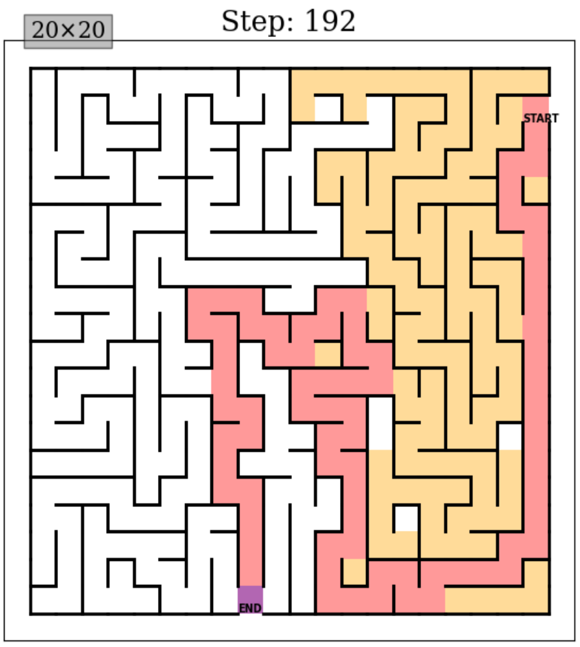
$$h(n_i) - h(n_j) = 1.1 * |n_i.second - n_j.second| + 0.9 * |n_i.first - n_j.first| \\ \leq cost(n_i, a, n_j)$$

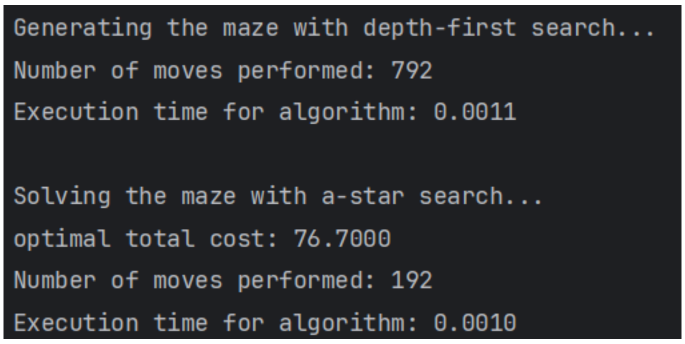
The following are the experimental results of the A* search algorithm with the improved heuristic applied. The experiment was conducted by running the 'a_star_search_special_version_test.py' file.



```
1 usage
def manhattan_distance_special_ver(coord1, coord2):  # 실제 이동 비용을 감안한 맨허튼 거리 함수 (special version)
    return 1.1 * abs(coord1[0] - coord2[0]) + 0.9 * abs(coord1[1] - coord2[1])
```

Figure 5. improved version of heuristic function





```
Generating the maze with depth-first search...
Number of moves performed: 792
Execution time for algorithm: 0.0011

Solving the maze with a-star search...
optimal total cost: 76.7000
Number of moves performed: 192
Execution time for algorithm: 0.0010
```

Figure 6, 7. a * search algorithm result (with improve heuristic function)

As seen in the experimental results, the A* search algorithm using the improved heuristic function derived the optimal solution at a lower search cost. Additionally, this function will guarantee an optimal solution for any given maze.