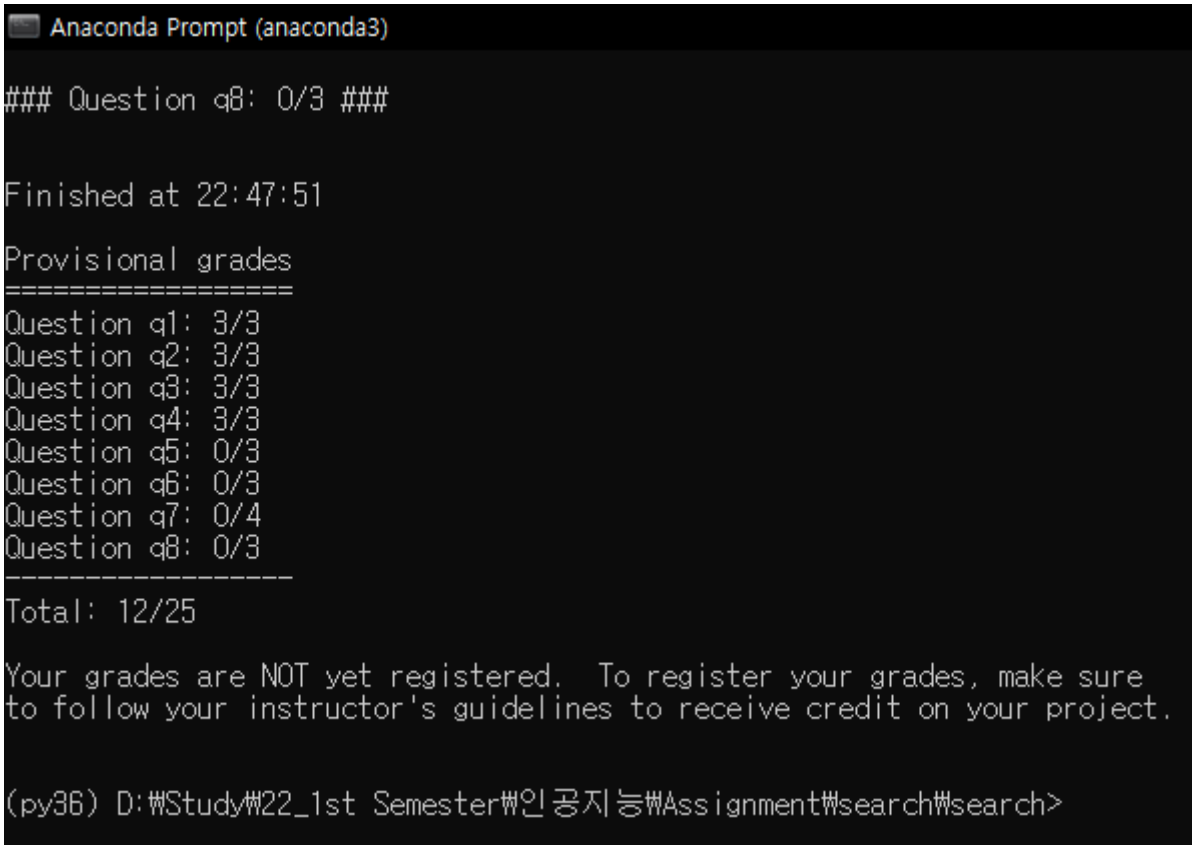
[COSE-361] 인공지능: Assignment #1

2016130927 박준영

A. autograder.py 결과

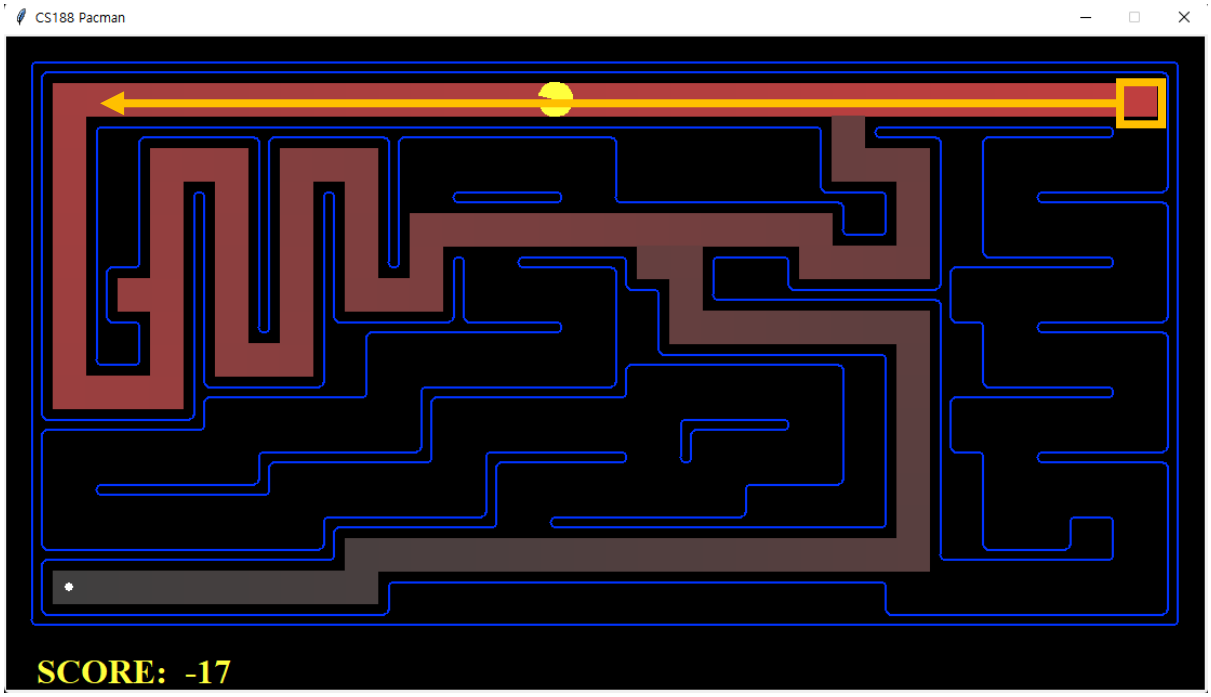


B. 팩맨 게임을 실행 시 세 개의 알고리즘(DFS, BFS, A*) 비교

Q1. 중간 크기 미로에서 DFS와 BFS 중 어떤 탐색 알고리즘이 더 나은가?

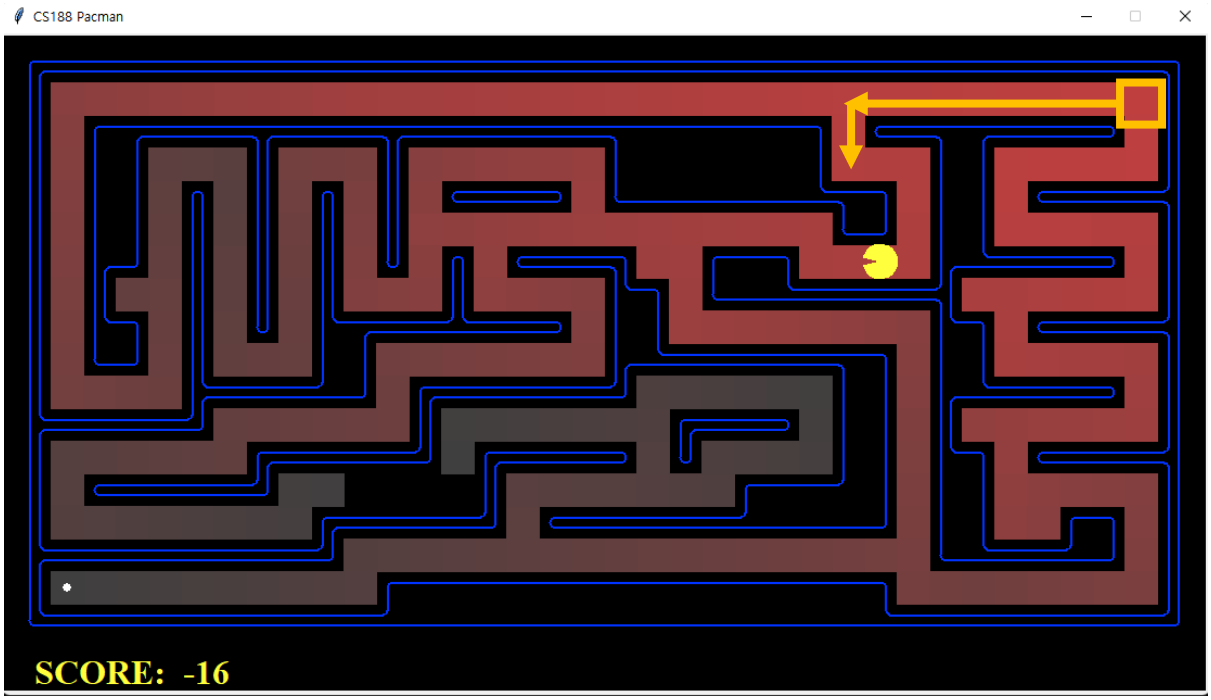
A1. 너비 우선 탐색, 즉 **BFS가 최단 경로를 찾는 데 있어 훨씬 더 효율적이다**. 중간 크기 미로를 DFS, BFS 두 종류를 통해 탐색한 결과 점수를 통해 비교해보자. 이때 특정 노드를 택하는 비용은 모두 동일하기 때문에 효율성의 측면에서 중요한 건 어떤 알고리즘이 "가장 짧은 경로"를 찾는지에 달려 있다.

중간 크기 미로(즉, medium maze)를 통해 두 탐색 알고리즘이 어떤 경로를 찾는지 비교해보자. 각 탐색 알고리즘이 반환한 경로가 (1). 최적의 경로인지, 경로를 찾기까지 탐색 알고리즘이 (2). 탐색한 노드의 개수 등 다양한 측면에서 두 알고리즘을 비교해보자. 이때 '최적'의 경로란 출발 노드에서 도착 노드까지 걸리는 가장 짧은 경로를 일컫는다. 모든 노드에서 비용이 1로 동일하기 때문에 경로 길이가 가장 짧은 경로를 찾자(UCS, A*는 이 점에서 다소 다르다).



<DFS: 탐색 노드 146 / 경로 길이 130>

탐색 트리가 주어진다고 생각해보자. DFS는 루트 노드에서 처음 택한 분기를 따라 리프 노드에 다다라 더 이상 움직일 수 없을 때까지 트리를 따라 내려간다. 그 분기에서 도착 노드가 없다면, 마지막으로 방문하지 않은 이전 분기 갈래로 다시 돌아간다. 위 경로에서 볼 수 있듯, DFS 경로는 출발 노드에서 서쪽과 남쪽이라는 두 가지 선택지가 있었지만, 스택에 삽입되는 방향이 서쪽부터 담기고 있다. 이 분기 지점에서 경로, 즉 도착 노드가 발견되기 때문에 출발 노드에 삽입된 남쪽 방향은 여전히 스택에 남아 있다.



<BFS: 탐색 노드 269 / 경로 길이 68>

BFS는 DFS와 달리 자료구조 큐를 통해 다음 방문할 노드를 뽑기 때문에 같은 레벨의 노드를 모두 방문해야 다음 레벨로 이어진다. 따라서 BFS 경로는 현 시점에서 가장 짧은 레벨의 경로라는 점에서 도착 노드를 만나는 그 시점이 곧 최적의 경로다. 비록 BFS가 DFS보다 탐색에 사용한 노드 개수가 거의 두 배(269:146)라 하더라도 말이다.

탐색 알고리즘이 사용되는 미로의 구조에 따라 달라지겠지만, BFS는 **모든 미로에서 최적의 경로를 보장**하는 반면 DFS는 보장하지 않는다. 그렇기 때문에 최적의 경로를 찾으려면 DFS가 아니라 BFS를 택해야 한다.

Q2. 제시된 휴리스틱 함수는 맨해튼 함수다. 맨해튼 함수보다 더 효율적인 함수가 있는지, 언제 이 함수가 효율적인가?

휴리스틱 함수로 사용된 맨해튼 함수는 현재 좌표 (x1, y2)와 도착 노드의 좌표 (x2, y2) 간의 차의 절댓값 합 (|x1-x2| + |y1-y2|)를 사용해 도착 노드와 상대적으로 가까운 방향이 어디인지 알려준다. 맨해튼 함수보다 더 효율적인 함수에 관해 논하기 전, 맨해튼 함수가 특히 이 미로 찾기 게임에서 적절한 휴리스틱으로 간주되는 이유를 알아보자.

(1). 먼저 미로 찾기 게임에서 팩맨이 이동 가능하도록 설정된 방향이 **4방향(동서남북)**이고 대각선 이동은 허용되지 않기 때문이다. L1 함수 자체가 2차원 좌표 간의 직선 이동을 염두에 두었기 때문에 대각선은 허용되지 않는다. 따라서 유클리드 함수 등 대각선을 허용하는 함수보다 맨해튼 함수가 본 미로 찾기 게임에 적합하다.

(2). 탐색의 목표가 고스트를 피하거나, 더 많은 음식을 먹는 등 다른 목표가 아니라 주어진 '**가장 빠른 경로**'를 찾는다는 것 역시 이유가 될 수 있다. 현재 좌표와 도착 노드의 좌표 간의 관계만을 염두에 둔 맨해튼 함수는, 도중 고스트를 만나 팩맨이 탈락하거나 더 많은 음식을 경로 상에서 먹어야 하는 등 다른 목적보다도 '더 빠른 경로'라는 목표를 이루기 위해 설계되었다. 물론 위와 같은 다른 요소를 첨부해 맨해튼 함수를 변형할 수도 있다.

그렇다면 미로 찾기 게임에서 맨해튼 함수보다 더 뛰어난(즉 효율적인) 휴리스틱 함수는 없을까? 도착 노드의 위치가 고정되어 있고, 출발 노드가 문제마다 달라진다고 가정해보자. 맨해튼 함수보다 효율적인 휴리스틱에는 무엇이 있을까? 먼저 '**정확한**' 휴리스틱 함수라는 개념을 떠올려보자.

(1). 미로 상의 모든 노드에서 도착 노드에 대한 최단 거리를 가지고 있다면 정확한 휴리스틱 값을 얻을 수 있다. 미로의 크기가 N*M이라면 다익스트라 알고리즘을 이 노드의 개수만큼 실행하면 된다. 하지만 이 경우, 일종의 '답지'를 들고 답안을 찾는 과정이기도 하며, 시간적으로 매우 비효율적이라는 단점이 있다.

(2). 모든 노드에서 다익스트라 알고리즘을 실행하지 말고, 특정 노드만 골라 선별적으로 도착 노드에 대한 최단 경로를 기록할 수 있다. 이 특정 노드를 waypoint라고 하자. 이 지점에서는 사전에 다익스트라를 통해 연산되었기 때문에 정확한 휴리스틱 값을 얻을 수 있다. 만일 waypoint가 아닌 노드라면 현재 노드에서 가장 가까운 waypoint까지의 거리 + 해당 waypoint에서 도착 노드까지의 거리를 휴리스틱으로 사용할 수 있다(구현마다 이는 달라질 수 있다).

h(n)이 현재 좌표 n에 대한 휴리스틱을 리턴하는 함수라고 할 때, n이 waypoint에 속한다면 곧바로 h(n)=w[goal] (n 위치를 waypoint로 설정하고 다익스트라 알고리즘으로 구한 도착 노드까지의 최단 거리)라고 할 수 있다. N이 waypoint가 아니라면, h(n)=w[n] + w[goal] (이때 w는 가지고 있는 waypoint 중 현재 좌표 n과 맨해튼 거리가 가장 작은 waypoint의 다익스트라 알고리즘으로 구한 거리 배열이라고 생각하자)로, 현재 좌표-waypoint-도착 노드까지의 최단 거리를 휴리스틱으로 사용했다.

정리하자면 waypoint 휴리스틱은 모든 노드에서의 최단 거리라는 확실한 답안지는 아니지만, 답안지가 될 만한 확률을 높임으로써 맨해튼 거리보다 정확한 정보를 받을 가능성이 높다.

이처럼 정확한 휴리스틱을 사용하기 위해서는 사전 연산이 필요하다는 전제가 필요하다. 미로 찾기 게임에서 출발 노드/도착 노드가 동일한 미로 구조에서 달라진다면, 미리 구해 놓은 waypoint의 각 노드 별 최단 거리 값을 유용하게 사용할 수 있다.

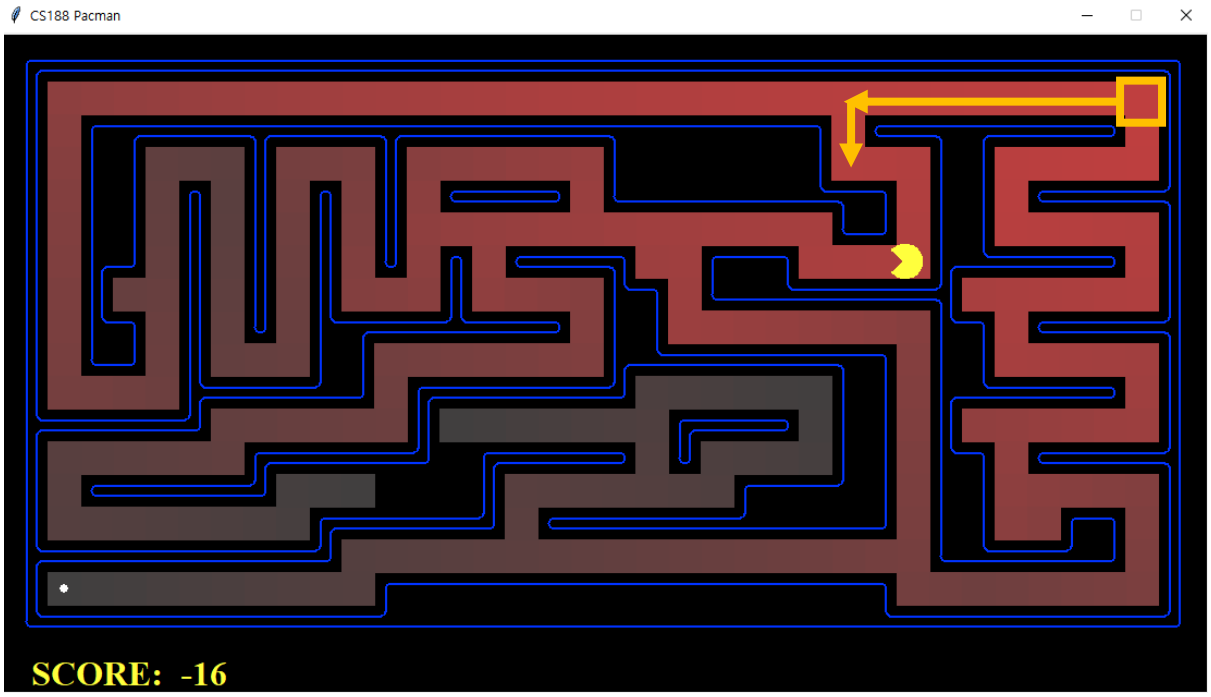
Q3. 미로 찾기에서 DFS 알고리즘을 응용한 알고리즘을 사용할 때, 어떻게 최적의 경로를 보장할 수 있을까? (직접 제안한 질문 및 답변)

A3. Q1에서 다루었듯, DFS는 탐색이 빠르지만 최적의 경로, '가장 짧은 경로'를 보장하지 않는다는 단점이 있다. 그렇기 때문에 최적성을 보장하는 BSF가 사용되지만, 모든 레벨의 노드를 탐색해야 한다는 특징 때문에 탐색 속도가 느리다는 단점이 있다. 어떻게 DFS의 빠른 속도를 유지하면서 BFS가 보장하는 최적의 경로를 찾을 수 있을까? **IDS**(iterative deepening DFS)를 통해 확인해보자.

```
def iterativeDeepeningDepthFirstSearch(problem):
    # Q3 의 IDS 풀이를 위한 소스 코드
    max_depth = 1
    # 최대 깊이 설정. 시작 깊이는 1.

    while True:
        # 최적 경로를 찾을 때까지 깊이를 1 씩 증가시켜 DFS.
        stack = util.Stack()
        stack.push([problem.getStartState(), [], 1])
        visited = set()
        visited.add(problem.getStartState())

        while not stack.isEmpty():
            cur_node, cur_path, cur_depth = stack.pop()
            if problem.isGoalState(cur_node): break

            successors = problem.getSuccessors(cur_node)
            for successor in successors:
                next_node, next_dir, next_cost = successor
                if next_node in visited or cur_depth + 1 > max_depth:
continue
                # 다음 깊이가 탐색 가능하면 스택에 push.
                visited.add(next_node)
                next_path = cur_path + [next_dir]
                stack.push([next_node, next_path, cur_depth + 1])
```

```
        if problem.isGoalState(cur_node): return cur_path
        else: max_depth += 1
```

DFS의 깊이를 max_depth로 제한, 1부터 시작해 최적의 경로를 찾을 때까지 1씩 증가시킨다. 만일 현재 제시된 최대 깊이 안에 도착 노드가 존재한다면, BFS보다 빠른 속도로 경로를 찾을 수 있다. 하지만 DFS 역시 단점이 있다. 깊이를 재설정하는 시점에서 이전에 방문한 노드를 **중복 방문**하기 때문이다. 중간 크기 미로를 IDS로 탐색했을 때 다음과 같은 결과를 얻었다.




```
(py36) D:₩Study₩22_1st Semester₩인공지능₩Assignment₩search₩search>python pacman.py -l mediumMaze -p SearchAgent -a fn=ids
[SearchAgent] using function ids
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.3 seconds
Search nodes expanded: 8451
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

<IDS: 탐색 노드 8451 / 경로 길이 68>

IDS는 BFS와 동일한 최적의 경로를 얻는 데에는 성공했지만, 중복 방문 정도가 크기 때문에 비효율적이었다. 위 미로는 도착 노드까지 깊이를 상대적으로 깊게 확장해야 하기 때문에 중복 방문 정도가 크기 때문이었다. 깊이가 1씩 확장되는 점에서 중복 정도가 너무 컸기 때문에, 제한 깊이에서 도착 노드를 만나지 못했을 경우 10씩 확장한 경우는 다음과 같다.

```
(py36) D:₩Study₩22_1st Semester₩인공지능₩Assignment₩search₩search>python pacman.py -l mediumMaze -p SearchAgent -a fn=ids
[SearchAgent] using function ids
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.2 seconds
Search nodes expanded: 947
```

<max_depth += 10인 경우 IDS: 탐색 노드 947 / 경로 길이 68>

중복 방문을 없앰으로써 보다 효율적인 경로 찾기가 가능했다(물론 10씩 확장하는 도중 DFS와 같이 최적의 경로를 벗어나는 경로를 택할 수 있기 때문에 최적성을 보장하기 위해서는 깊

이를 1씩 올리는 게 가장 적합하다). 이처럼 IDS는 (구현, 미로 등에 따라 달라지겠지만) DFS를 활용하면서도 BFS와 같이 최적의 경로를 보장하도록 설계 가능하다. 일반적으로, IDS가 BFS보다 더 좋은 성능을 내려면 훨씬 더 깊은 단계에 도착 노드가 위치해 있어야 한다.

C. 구현한 휴리스틱 함수(blocked_weighted_heuristic)를 사용한 A* 알고리즘의 실행 결과

미로 찾기 게임의 맨해튼 휴리스틱은 일반적으로 매우 잘 작동하는 휴리스틱이지만, 현재 좌표와 도착 좌표 간의 관계만을 다룬다는 점에서 벽으로 가로 막혀 있는 상황, 벽이 없기 때문에 갈 수 있는 상황을 구별하지 못한다. "벽에 가로막혀 갈 수 없다면 가지 않아도 된다"는 직관을 통해 맨해튼 휴리스틱을 재설계했다.

(1). **벽에 가로막힌(blocked) 상황**: 현재 좌표가 벽으로 둘러 쌓여 있는지 확인할 수 있다. 상하좌우 각각 벽인지 확인해, 현재 좌표에서 팩맨 에이전트가 어느 방향으로 이동하지 못하는지 알 수 있다. 예를 들어 위쪽, 왼쪽 좌표가 벽이라면 팩맨은 현재 위치 기준 2사분면으로 이동할 수 없다.

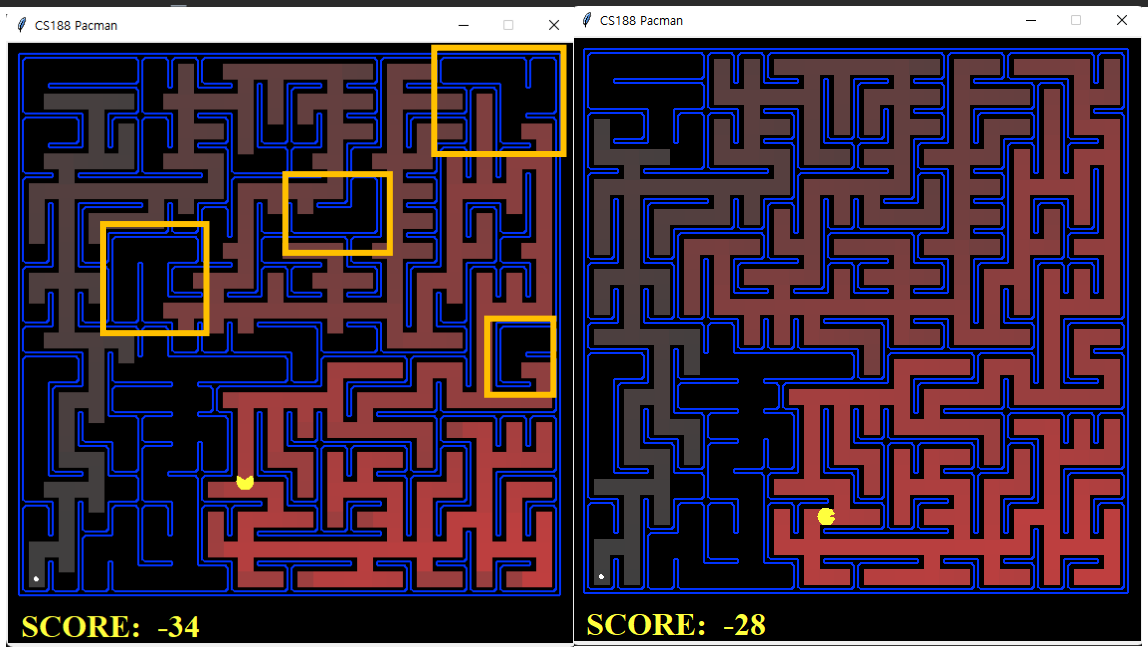
(2). **도착 노드 기준 상대적 위치**: 팩맨이 가야 하는 방향으로 가지 못한다면 우선순위를 낮게 주자는 본 휴리스틱 함수의 설계 의도에 따라, 팩맨이 갈 수 없는 상황을 위에서 구했다. 그렇다면 팩맨은 어떤 방향으로 나아가야 할까? 맨해튼 함수가 현재 좌표와 도착 노드 좌표 간의 거리를 다루고 있는 것처럼, 현재 좌표의 노드는 당연히 도착 노드가 위치한 방향으로 이동해야 한다. 도착 노드를 원점으로 생각해 어떤 사분면에 현재 팩맨 에이전트가 위치하는지 파악할 수 있다.

(3). **가중치**: 팩맨이 도착 노드를 향해 나아가야 할 방향으로 갈 수 없게 벽으로 막혀 있다면, 갈 필요가 없다. 즉 우선순위를 낮게 주기 위해 맨해튼 함수로 반환할 값을 키워주면 된다. 벽으로 가로막혀 있지 않다면, 일반적인 맨해튼 함수를 반환하자.

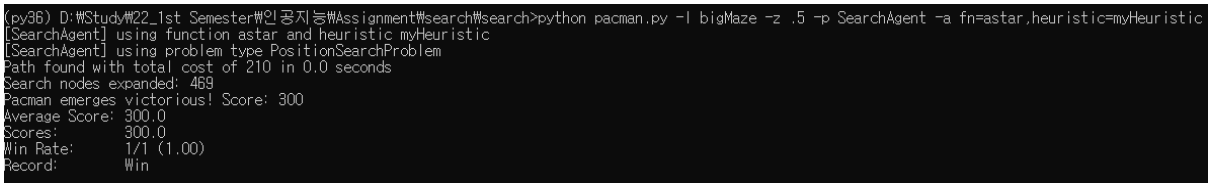
소스 코드는 다음과 같다.

```
def myHeuristic(position, problem, info={}):
    "You build your own heuristic function here: block_weighted_distance "

    cur_x, cur_y = position
    goal_x, goal_y = problem.goal
    w = 1

    L2_distance = manhattanHeuristic(position, problem)
    dx = [0, 0, 1, -1]
    dy = [1, -1, 0, 0]
    def is_blocked(cur_x, cur_y):
        blocked_pos = []
        for x, y in zip(dx, dy):
            next_x, next_y = cur_x + x, cur_y + y
            blocked_pos.append(problem.walls[next_x][next_y])
        return blocked_pos
    # 현재 좌표가 벽으로 둘러 쌓여 있는지 확인한다. [up, down, right, left] (상, 하, 우, 좌 위치가 벽이라면 True 리턴)
```

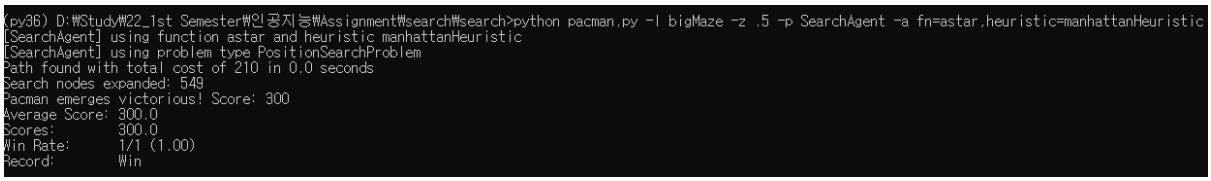
```
    def get_loc(cur_x, cur_y):
        # 현재 노드가 도착 노드를 원점 기준으로, 상대적으로 어디에 위치해 있는지
알려준다.
        # 1: 1 사분면 2: 2 사분면 3: 3 사분면 4: 4 사분면

        if goal_x <= cur_x and goal_y <= cur_y: return 1
        elif goal_x > cur_x and goal_y <= cur_y: return 2
        elif goal_x > cur_x and goal_y > cur_y: return 3
        else: return 4

    def get_weight(blocked_pos, relative_loc):
        # 현재 노드가 벽으로 가로 막혀 도착 노드로 갈 수 없다면 우선순위를 낮게 주어야
한다.
        up, down, right, left = blocked_pos
        weight = 0.5
        # bigMaze: weight 0.5 일 때 효과적, tinyMaze 또는 mediumMaze 일 때에는
0.2 일 때 효과적
        if up and left and relative_loc == 4: return weight
        elif up and right and relative_loc == 3: return weight
        elif down and left and relative_loc == 1: return weight
        elif down and right and relative_loc == 2: return weight
        else: return 0

    blocked_pos = is_blocked(cur_x, cur_y)
    relative_loc = get_loc(cur_x, cur_y)

    w += get_weight(blocked_pos, relative_loc)
    # 일반적인 경우 맨해튼 거리를 그대로, 벽으로 둘러 쌓여 있다면 우선순위를 낮게 (즉
weighted 값을 크게) 주어 방문하지 않도록 유도한다.

    return L2_distance**w
```



<블록 기반 가중치 맨해튼 휴리스틱 함수(myHeuristic)> ↔ <맨해튼 휴리스틱 함수>



< 블록 기반 가중치 맨해튼 휴리스틱 함수(myHeuristic): 탐색 노드 469 / 경로 길이 210>



<맨해튼 휴리스틱 함수: 탐색 노드 549 / 경로 길이 210>