18-2 인공지능 과제 1

ITBT 탈출하기

2013010926 강은석

1. 개요

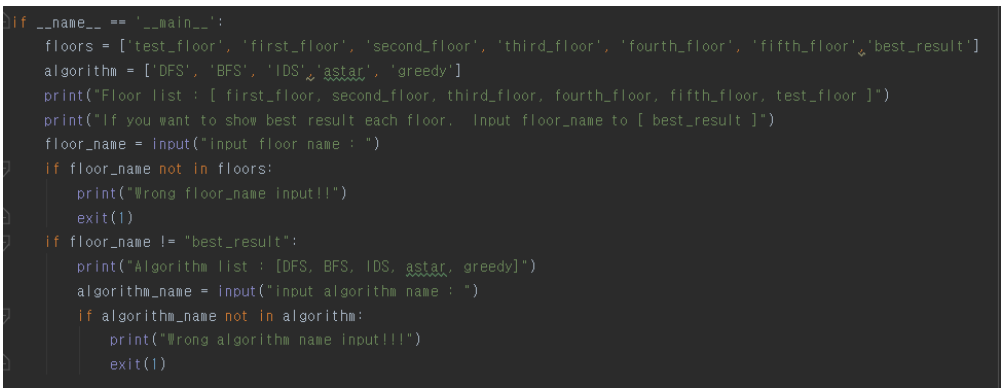
다양한 Search Algorithm을 이용해서 여러 층의 미로의 경로를 탐색해야 합니다.

각 층에는 하나의 알고리즘만 사용해야 합니다.

출구에 가기전에 키를 습득하고 출구로 가야합니다.

각 층의 최단경로(length)와 탐색한 노드의 개수(time)를 출력합니다.

2. 코드 실행 방법(주어진 층_floor_input.txt 파일 input 시)



```
if __name__ == '__main__':
    floors = ['test_floor', 'first_floor', 'second_floor', 'third_floor', 'fourth_floor', 'fifth_floor','best_result']
    algorithm = ['DFS', 'BFS', 'IDS','astar', 'greedy']
    print("Floor list : [ first_floor, second_floor, third_floor, fourth_floor, fifth_floor, test_floor ]")
    print("If you want to show best result each floor.  Input floor_name to [ best_result ]")
    floor_name = input("input floor name : ")
    if floor_name not in floors:
        print("Wrong floor_name input!!")
        exit(1)
    if floor_name != "best_result":
        print("Algorithm list : [DFS, BFS, IDS, astar, greedy]")
        algorithm_name = input("input algorithm name : ")
        if algorithm_name not in algorithm:
            print("Wrong algorithm name input!!!")
            exit(1)
```

코드의 최하단을 보면 main이 존재합니다.

처음에 floor를 인풋으로 받습니다. Ex) 'first_floor', 'second_floor' ... 'fifth_floor'

만약 test_floor를 돌려 테스트를 할 때는 floor 인풋으로 'test_floor'를 넣으면 됩니다.

모든 층에 대한 best 결과값을 얻고 싶을때는 floor인풋에 'best_result'를 넣으면 됩니다.

floor이름을 인풋으로 넣은 이후에는 알고리즘을 인풋으로 받습니다.

'algorithm' 이라는 배열을 보면 'DFS' [Depth First Search], 'BFS' [Breadth First Search], 'IDS' [Iterative First Search], 'astar' [A* algorithm], 'greedy' [Greedy Best First Search] 알고리즘을 사용할 수 있습니다.

3. 각층 코드(모든 층이 같은 방식으로 돌아가기 때문에 'first_floor'함수를 예를 들어 설명하겠습니다.)

```
def first_floor(floor_name, search_algorithm):
    floor_info, floor_map = read_input_files(floor_name)
    if search_algorithm == "astar" or search_algorithm == "greedy":
        length1, time1, optimal_path1 = heuristic_search_algorithm(floor_info, floor_map, search_algorithm,
                                                                   floor_info[3], floor_info[4])
        length2, time2, optimal_path2 = heuristic_search_algorithm(floor_info, floor_map, search_algorithm,
                                                                   floor_info[4], floor_info[5])
    elif search_algorithm == "DFS":
        length1, time1, optimal_path1 = DFS(floor_info, floor_map, floor_info[3], floor_info[4])
        length2, time2, optimal_path2 = DFS(floor_info, floor_map, floor_info[4], floor_info[5])
    elif search_algorithm == "BFS":
        length1, time1, optimal_path1 = BFS(floor_info, floor_map, floor_info[3], floor_info[4])
        length2, time2, optimal_path2 = BFS(floor_info, floor_map, floor_info[4], floor_info[5])
    elif search_algorithm == "IDS":
        length1, time1, optimal_path1 = IDS(floor_info, floor_map, floor_info[3], floor_info[4])
        length2, time2, optimal_path2 = IDS(floor_info, floor_map, floor_info[4], floor_info[5])

    length = length1+length2
    time = time1+time2
    write_output_file(floor_name, floor_info, floor_map, optimal_path1, optimal_path2, length, time)
    print("----- first_floor -----")
    print("length = ", length)
    print("time = ", time)
```

메인에서 입력 받은 'floor_name'과 'algorithm_name'을 인자로 받아 first_floor함수를 실행합니다.

우선 'read_input_files(floor_name)' 함수를 실행시켜 층의 정보는 'floor_info'에 지도는 'floor_map'에 받아옵니다.

'algorithm'이름에 해당하는 알고리즘을 실행하는데, 방식은 시작점~키까지, 키~도착지점까지 총 두 번 알고리즘을 실행하게 했습니다.

그 뒤 알고리즘의 아웃풋인 'length, time, optimal_path'를 종합해 'write_output_file'함수의 인자로 넘겨서 'first_floor_output.txt'를 반환합니다.

```
def read_input_files(floor):
    cur_path = os.getcwd()
    os.chdir(cur_path)
    input_path = floor + '_input' + '.txt'
    floor_map = []
    with open(input_path, 'r') as f:
        floor_data = f.readlines()
        for floor_row in floor_data:
            floor_row = floor_row.split()
            floor_row = list(map(int, floor_row))
            floor_map.append(floor_row)

    f.close()
    floor_info = floor_map[0]
    floor_map = floor_map[1:]
    # save start_point, key_point, end_point
    for row in range(len(floor_map)):
        for col in range(int(floor_info[1])):
            if floor_map[row][col] == 3:
                floor_info.append([row, col])
            if floor_map[row][col] == 4:
                floor_info.append([row, col])
            if floor_map[row][col] == 6:
                floor_info.append([row, col])

    return floor_info, floor_map
```

```
def write_output_file(floor, floor_info, floor_map, path1, path2, length, time):
    cur_path = os.getcwd()
    os.chdir(cur_path)
    if path1[0] == floor_info[3]:   # remove start_point
        path1.pop(0)
    if path2[-1] == floor_info[5]:  # remove end_point
        path2.pop(-1)
    for i in range(len(path1)):
        floor_map[path1[i][0]][path1[i][1]] = 5

    for i in range(len(path2)):
        floor_map[path2[i][0]][path2[i][1]] = 5

    output_path = floor + '_output' + '.txt'

    with open(output_path, 'w') as f:
        for floor_row in floor_map:
            for point in floor_row:
                f.write(str(point)+' ')
            f.write('\n')
        f.write("------------------------\n")
        f.write("length = " + str(length) + "\n")
        f.write("time = " + str(time))

    f.close()
```

4. 알고리즘 코드 설명

- **DFS**

Python의 queue의 LifoQueue(stack)을 이용해서 구현했습니다.

Stack의 인자로는 좌표와 이제까지 온 length를 같이 넣어주었습니다.

```
# move down from cur_point
if cur_point not in visited_points:
    if floor_map[cur_point[0]+1][cur_point[1]] in can_go_point and \
            [cur_point[0]+1, cur_point[1]] not in visited_points:
        count += 1
    if floor_map[cur_point[0]][cur_point[1]+1] in can_go_point and \
            [cur_point[0], cur_point[1]+1] not in visited_points:
        count += 1
    if floor_map[cur_point[0]][cur_point[1]-1] in can_go_point and \
            [cur_point[0], cur_point[1]-1] not in visited_points:
        count += 1
    if floor_map[cur_point[0]-1][cur_point[1]] in can_go_point and \
            [cur_point[0]-1, cur_point[1]] not in visited_points:
        count += 1
    if count > 1:
        stack.put([[cur_point[0], cur_point[1]], length])
        path_stack.put(save_path + [cur_point])
```

LifoQueue에는 stack의 top을 볼 수 있는 함수가 구현되지 않았기 때문에 위의 그림과 같이 갈림길이 3개이면 stack에 cur_point를 한번 더 넣는 방법으로 구현을 하였습니다.

```
visited_points.append(cur_point)
if floor_map[cur_point[0]+1][cur_point[1]] in can_go_point and \
        [cur_point[0] + 1, cur_point[1]] not in visited_points:  # check next_point is not parent a
    stack.put([[cur_point[0] + 1, cur_point[1]], length + 1])  # save next_point, parent, length
    save_path = path_stack.get()
    path_stack.put(save_path)
    path_stack.put(save_path + [[cur_point[0] + 1, cur_point[1]]])  # save path from start_point
    cur_point = [cur_point[0]+1, cur_point[1]]
    save_path = save_path + [cur_point]
    length += 1

# move right from cur_point
elif floor_map[cur_point[0]][cur_point[1] + 1] in can_go_point and \
        [cur_point[0], cur_point[1] + 1] not in visited_points:  # check next_point is not parent a
    stack.put([[cur_point[0], cur_point[1] + 1], length + 1])  # save next_point, parent, length
    save_path = path_stack.get()
    path_stack.put(save_path)
    path_stack.put(save_path + [[cur_point[0], cur_point[1] + 1]])  # save path from start_point
    cur_point = [cur_point[0], cur_point[1]+1]
    save_path = save_path + [cur_point]
    length += 1

# move left from cur_point
elif floor_map[cur_point[0]][cur_point[1] - 1] in can_go_point and \
        [cur_point[0], cur_point[1] - 1] not in visited_points:  # check next_point is not parent a
    stack.put([[cur_point[0], cur_point[1] - 1], length + 1])  # save next_point, parent, length
    save_path = path_stack.get()
    path_stack.put(save_path)
    path_stack.put(save_path + [[cur_point[0], cur_point[1]-1]])  # save path from start_point
    cur_point = [cur_point[0], cur_point[1]-1]
    save_path = save_path + [cur_point]
    length += 1
```

```
# move up from cur_point
elif floor_map[cur_point[0] - 1][cur_point[1]] in can_go_point and \
        [cur_point[0] - 1, cur_point[1]] not in visited_points:  # check next_point is not paren
    stack.put([[cur_point[0] - 1, cur_point[1]], length + 1])  # save next_point, parent, length
    save_path = path_stack.get()
    path_stack.put(save_path)
    path_stack.put(save_path + [[cur_point[0] - 1, cur_point[1]]])  # save path from start_point
    cur_point = [cur_point[0]-1, cur_point[1]]
    save_path = save_path + [cur_point]
    length += 1
else:
    save_path = path_stack.get()
    stack_factor = stack.get()
    cur_point = stack_factor[0]
    length = stack_factor[1]

time += 1
```

cur_point가 end_point까지 도달할 때까지 while문을 돌면서 cur_point의 아래, 오른쪽, 왼쪽, 위 순서대로 DFS가 돌아가게 구현했습니다.

이렇게 구현한이유는 미로의 시작점은 좌측상단에 있었고 도착지점은 우측 하단에 있었기 때문에 아래와 오른쪽 먼저 보도록 하였습니다.

또한 path_stack을 따로 두어서 이제까지 왔던 경로를 따로 저장해 놓는 방식으로 시작점부터 도착지점까지의 경로를 저장했습니다.

- **BFS**

Python의 queue의 Queue를 사용해서 구현하였습니다.

q의 인자로는 좌표, 이제까지 온 length를 넣어주었습니다.

또한 optimal_path_q라는 queue를 만들어서 좌표까지 온 경로들을 따로 저장했습니다.

마지막으로 save_parent_q라는 queue를 만들어서 자신의 부모좌표가 들어가지 않도록 하였습니다.

```
while cur_point != end_point:
    cur_point = q.get()
    parent_point = save_parent_q.get()
    save_path = optimal_path_q.get()

    length = cur_point[1]
    cur_point = cur_point[0]

    if cur_point[0]+1 < floor_info[1]:  # move down from cur_point
        if floor_map[cur_point[0]+1][cur_point[1]] in can_go_point \
                and [cur_point[0]+1, cur_point[1]] != parent_point:  # check next_point is not parent and can move
            q.put([[cur_point[0]+1, cur_point[1]], length+1])        # save next_point and length
            optimal_path_q.put(save_path + [[cur_point[0]+1, cur_point[1]]])  # save path from start_point
            save_parent_q.put(cur_point)

    if cur_point[1]+1 < floor_info[2]:  # move right from cur_point
        if floor_map[cur_point[0]][cur_point[1]+1] in can_go_point \
                and [cur_point[0], cur_point[1]+1] != parent_point:  # check next_point is not parent and can move
            q.put([[cur_point[0], cur_point[1]+1], length+1])        # save next_point and length
            optimal_path_q.put(save_path + [[cur_point[0], cur_point[1]+1]])   # save path from start_point
            save_parent_q.put(cur_point)

    if cur_point[0]-1 > -1:  # move up from cur_point
        if floor_map[cur_point[0]-1][cur_point[1]] in can_go_point \
                and [cur_point[0]-1, cur_point[1]] != parent_point:  # check next_point is not parent and can move
            q.put([[cur_point[0]-1, cur_point[1]], length+1])        # save next_point and length
            optimal_path_q.put(save_path + [[cur_point[0]-1, cur_point[1]]])  # save path from start_point
            save_parent_q.put(cur_point)

    if cur_point[1]-1 > -1:  # move left from cur_point
        if floor_map[cur_point[0]][cur_point[1]-1] in can_go_point \
                and [cur_point[0], cur_point[1]-1] != parent_point:  # check next_point is not parent and can_move
            q.put([[cur_point[0], cur_point[1]-1], length+1])        # save next_point and length
            optimal_path_q.put(save_path + [[cur_point[0], cur_point[1]-1]])  # save path from start_point
            save_parent_q.put(cur_point)
```

- **IDS**

Python의 queue의 LifoQueue(stack)을 이용해서 구현하였습니다.

두번의 while문을 돌면서 첫번째 while문은 도착지점에 도달할 때까지 deep을 하나씩 올려주는 방식으로 하였습니다.

```
while cur_point != end_point:
    length = 0
    visited_points = []     # save visited points
    cur_point = start_point
    visited_points.append(cur_point)

    stack = queue.LifoQueue()             # IDS stack
    save_path_stack = queue.LifoQueue()   # save paths from start_point
    stack.put([cur_point, length])        # save cur_point and length
    save_path_stack.put([cur_point])
    deep += 1
```

두번째 while문에서는 length가 deep에 도달할 때까지 DFS알고리즘을 수행했습니다. 하나의 좌표의 length가 deep에 도달하면 pop을 해주는 방식을 이용했습니다.

```
while True:
    cur_point = stack.get()
    save_path = save_path_stack.get()
    length = cur_point[1]
    cur_point = cur_point[0]

    if cur_point == end_point:  # end_point
        optimal_path = save_path
        break
    if length == deep and stack.qsize() != 0:  # IDS length reaches to deep stack pop
        continue
```

stack에는 좌표와 좌표까지 온 length를 넣어주었고, save_path_stack에는 좌표까지 온경로를 넣어주었습니다.

```
# move down from cur_point
if floor_map[cur_point[0]+1][cur_point[1]] in can_go_point \
        and [cur_point[0]+1, cur_point[1]] not in visited_points:  # check next_point is not in visi
    stack.put([[cur_point[0]+1, cur_point[1]], length+1])          # save next_point
    save_path_stack.put(save_path + [[cur_point[0]+1, cur_point[1]]])  # save path from start_point
    visited_points.append([cur_point[0]+1, cur_point[1]])

# move right from cur_point
if floor_map[cur_point[0]][cur_point[1]+1] in can_go_point \
        and [cur_point[0], cur_point[1]+1] not in visited_points:  # check next_point is not in visi
    stack.put([[cur_point[0], cur_point[1]+1], length+1])          # save next_point
    save_path_stack.put(save_path + [[cur_point[0], cur_point[1]+1]])  # save path from start_point
    visited_points.append([cur_point[0], cur_point[1]+1])

# move up from cur_point
if floor_map[cur_point[0]-1][cur_point[1]] in can_go_point \
        and [cur_point[0]-1, cur_point[1]] not in visited_points:  # check next_point is not in visi
    stack.put([[cur_point[0]-1, cur_point[1]], length+1])          # save next_point
    save_path_stack.put(save_path + [[cur_point[0]-1, cur_point[1]]])  # save path from start_point
    visited_points.append([cur_point[0]-1, cur_point[1]])

# move left from cur_point
if floor_map[cur_point[0]][cur_point[1]-1] in can_go_point \
        and [cur_point[0], cur_point[1]-1] not in visited_points:  # check next_point is not in visi
    stack.put([[cur_point[0], cur_point[1]-1], length+1])          # save next_point
    save_path_stack.put(save_path + [[cur_point[0], cur_point[1]-1]])  # save path from start_point
    visited_points.append([cur_point[0], cur_point[1]-1])

if stack.qsize() == 0:
    break
time += 1
```

- **A* & Greedy best first search**

Python의 queue의 PriorityQueue를 이용해서 구현하였습니다.

좌표가 end_point가 될 때까지 while문을 돌면서 evaluation값의 우선순위가 가장 높은 좌표를 꺼내 동작하도록 하였습니다.

```
while cur_point != end_point:
    next_points = []
    # find all next_point from cur_point
    if cur_point[0]-1 > -1:  # move up from cur_point
        if floor_map[cur_point[0]-1][cur_point[1]] in can_go_point \
                and [cur_point[0]-1, cur_point[1]] not in visited_points:  # check next_point is
            next_points.append([cur_point[0]-1, cur_point[1]])       # save next_point
            visited_points.append([cur_point[0]-1, cur_point[1]])    # add to visited_points

    if cur_point[0]+1 < floor_info[1]:  # move down from cur_point
        if floor_map[cur_point[0]+1][cur_point[1]] in can_go_point \
                and [cur_point[0]+1, cur_point[1]] not in visited_points:  # check next_point is
            next_points.append([cur_point[0]+1, cur_point[1]])       # save next_point
            visited_points.append([cur_point[0]+1, cur_point[1]])    # add to visited_points

    if cur_point[1]-1 > -1:   # move left from cur_point
        if floor_map[cur_point[0]][cur_point[1]-1] in can_go_point \
                and [cur_point[0], cur_point[1]-1] not in visited_points:  # check next_point is
            next_points.append([cur_point[0], cur_point[1]-1])       # save next_point
            visited_points.append([cur_point[0], cur_point[1]-1])    # add to visited_points

    if cur_point[1]+1 < floor_info[2]:  # move right from cur_point
        if floor_map[cur_point[0]][cur_point[1]+1] in can_go_point \
                and [cur_point[0], cur_point[1]+1] not in visited_points:  # check next_point is
            next_points.append([cur_point[0], cur_point[1]+1])       # save next_point
            visited_points.append([cur_point[0], cur_point[1]+1])    # add to visited_points
```

A* 일 때와 Greedy best first search일 때를 나눠서 evaluation function을 만들었습니다. A*는 이제까지 온 거리(length+1) 와 현재좌표와 도착지점의 좌표 값을 뺀 heuristic function 값을 더하였습니다.

Greedy best first search는 현재 좌표와 도착지점의 좌표 값을 뺀 heuristic function 값을 더하였습니다.

heuristic_search_queue에는 evaluation값, 이제까지 온 거리(length), 좌표, 좌표까지 온 경로를 저장해주었습니다. 그 뒤 evaluation 값의 우선순위가 높은 좌표를 꺼내주었습니다.

```
for i in range(len(next_points)):
    if algorithm == "greedy":      # calculate eval function = heuristic function
        eval = abs(end_point[0]-next_points[i][0]) + abs(end_point[1]-next_points[i][1])
    elif algorithm == "astar":     # calculate eval function = length from start_point + heuristic function
        eval = (length+1) + abs(end_point[0]-next_points[i][0]) + abs(end_point[1]-next_points[i][1])
    heuristic_search_queue.put((eval, length+1, next_points[i], save_path+[next_points[i]]))
```

```
# find min eval_score and update info
min_eval_data = heuristic_search_queue.get()
length = min_eval_data[1]
cur_point = min_eval_data[2]
save_path = min_eval_data[3]

time += 1
```

5. 각층마다 어떤 알고리즘을 선택한 이유

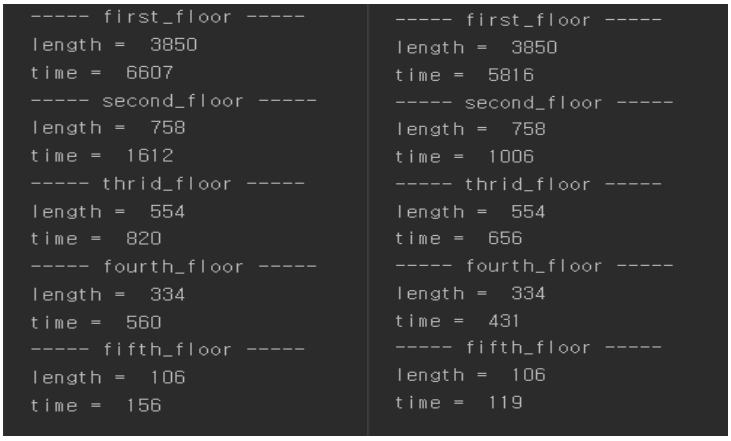
저는 우선 수업시간에 배운 모든 알고리즘을 구현하고 모든 결과값을 보고 결정을 하고 싶었기에 DFS, BFS, IDS, A*, Greedy best first search를 구현해보았고 결과를 보았습니다.

결과를 보기 전, 제 예상으로는 도착지점을 미리 알 수 있기 때문에 heuristic function을 이용한 A* 알고리즘이나 Greedy best first search 알고리즘을 좋은 결과값을 보여줄 것이라 예상하였습니다.

이론상 optimal한 경로를 찾을 수 있고 length나 time면에서 우수한 A*알고리즘이나 optimal한 경로를 찾을 수 없을 지라도 length나 time에서 좋은 Greedy best first search 알고리즘은 예상대로 좋은 결과값을 보여주었지만 Greedy알고리즘의 경우는 optimal한 경로를 찾아줄 뿐만 아니라 'time'값도 A*보다 적은 결과를 보여주었습니다.

하지만 미로의 나가는 길이 한쪽으로 치우칠 수도 있기 때문에 A* 알고리즘이나 Greedy best first search보다 DFS의 결과가 더 좋을 수 있는 경우도 있을 것이라 생각했고, 2번째 층의 경우는 DFS가 A*나 Greedy알고리즘 보다 더 좋은 결과를 보여주었습니다.

(순서대로 DFS, BFS, IDS 결과값)



(순서대로 A*, Greedy best first search 결과값)

6. 각층마다 최단경로(length), 탐색 노드 수(time) 반환

```
Floor list : [ first_floor, second_floor, third_floor, fourth_floor, fifth_floor,
If you want to show best result each floor.  Input floor_name to [ best_result ]
input floor name : best_result
----- first_floor -----
algorithm =  greedy
length =  3850
time =  5816
----- second_floor -----
algorithm =  DFS
length =  758
time =  953
----- thrid_floor -----
algorithm =  greedy
length =  554
time =  656
----- fourth_floor -----
algorithm =  greedy
length =  334
time =  431
----- fifth_floor -----
algorithm =  greedy
length =  106
time =  119
```

층에 맞는 best 알고리즘 결과를 출력하려면 input floor name에 'best_result'라고 입력하면 받아볼 수 있습니다.

1층, 3층, 4층, 5층은 Greedy Best First Search 알고리즘을 사용하였습니다.

2층은 Depth First Search 알고리즘을 사용하였습니다.