

Assignment#1 Maze

IT/BT 탈출하기

2014001578 지현승

코드 설명

- 이번 과제를 풀기 위해 DFS, BFS, Iterative deepening search, Greedy best first search, A* 알고리즘을 구현했습니다.
- 각 층에서 다섯가지 알고리즘을 실행한 결과 **모든 알고리즘에서 Optimal**한 결과가 나왔습니다. 따라서 노드 방문 횟수 (Time)을 기준으로 각 층마다 알고리즘을 선택했습니다.

first_floor function

```
def first_floor(): #1

    input_map = map_import('first_floor_input.txt') #2
    size_x = len(input_map)
    size_y = len(input_map[0])

    i = 0

    for i in range(size_y):
        if input_map[0][i] == 3: #3
            state = [0,i]
            break

    if i == size_y:
        return -1

    i = 0

    stack = []
    output_map = [[0]*size_y for i in range(size_x)]
    result = DFS(state,input_map,output_map,stack,6) #4
    state = [result[0],result[1]]
    result_i = result[2]
    result_j = result[3]

    stack2 = []
    output_map = [[0]*size_y for i in range(size_x)]
    result = DFS(state,input_map,output_map,stack2,4) #5
    result_i += result[2]
    result_j += result[3]

    map_export('first_floor_output.txt',input_map,result_i,result_j) #6
```

Nth_floor function

- 위 그림의 #4, #5에서 DFS(first, second, third), Greedy(fourth, fifth) 알고리즘을 사용하는 점, txt 파일의 입, 출력을 각각 다르게 지정한다는 점을 제외 했을 때, 각 층의 floor function은 전체적인 구조가 흡사합니다.
- 따라서 floor functio의 경우 first_floor만 설명하겠습니다.

first_floor

- #1 first_floor 함수는 parameter도, return도 없는 함수입니다.
- #2 first_floor 함수는 map_import 함수를 실행합니다. map_import 함수는 first_floor_input.txt를 열어 maze를 input_map에 입력시켜줍니다.
- #3 input_map에서 시작지점(3)을 찾습니다.

```
def first_floor(): #1

    input_map = map_import('first_floor_input.txt') #2
    size_x = len(input_map)
    size_y = len(input_map[0])

    i = 0

    for i in range(size_y):
        if input_map[0][i] == 3: #3
            state = [0,i]
            break
```

first_floor

- #4 알고리즘 실행을 도와주는 output_map과 stack을 만든 뒤, Key(6)를 목적지로 하여 DFS를 실행합니다. DFS는 [x,y,i,j] 형식의 리스트를 return 합니다.
- #4 x와 y는 목적지 좌표, i는 목적지까지 탐색한 노드 수, j는 목적지까지의 length를 의미합니다.

```
stack = []
output_map = [[0]*size_y for i in range(size_x)]
result = DFS(state,input_map,output_map,stack,6) #4
state = [result[0],result[1]]
result_i = result[2]
result_j = result[3]
```

first_floor

- #5 Key(6)을 시작지점으로, Goal(4)을 도착점으로 DFS를 실행합니다.
- #5 DFS는 input_map을 정답 경로로 바꾸는 역할도 담당하고 있습니다.
- #6 map_export는 두 번의 DFS를 통해 변경된 input_map과 length, time을 파일에 입력하는 함수입니다.

```
stack2 = []  
output_map = [[0]*size_y for i in range(size_x)]  
result = DFS(state, input_map, output_map, stack2, 4) #5  
result_i += result[2]  
result_j += result[3]  
  
map_export('first_floor_output.txt', input_map, result_i, result_j) #6
```

1층

- 1층에서 선택한 알고리즘은 DFS 입니다.
- DFS는 Time이 5349가 나왔습니다.
- DFS에서 자식 노드를 탐색하는 순서는 오른쪽, 아래, 위, 왼쪽 순입니다.
- 최단 경로는 3850입니다.

Algorithm	Time
DFS	5349
Greedy	6262
A*	6651
BFS	6748
IDS	7595663

2층

- 2층에서 선택한 알고리즘은 DFS 입니다.
- DFS는 Time이 824가 나왔습니다.
- DFS에서 자식 노드를 탐색하는 순서는 오른쪽, 아래, 위, 왼쪽 순입니다.
- 최단 경로는 758입니다.

Algorithm	Time
DFS	824
Greedy	975
A*	1507
BFS	1718
IDS	308371

3층

- 3층에서 선택한 알고리즘은 DFS 입니다.
- DFS는 Time이 731이 나왔습니다.
- DFS에서 자식 노드를 탐색하는 순서는 오른쪽, 아래, 위, 왼쪽 순입니다.
- 최단 경로는 554입니다.

Algorithm	Time
DFS	731
Greedy	794
A*	831
BFS	1002
IDS	144687

4층

- 4층에서 선택한 알고리즘은 Greedy 알고리즘 입니다.
- Greedy는 Time이 434이 나왔습니다.
- 최단 경로는 334입니다.

Algorithm	Time
DFS	463
Greedy	434
A*	586
BFS	594
IDS	55450

5층

- 5층에서 선택한 알고리즘은 Greedy 알고리즘 입니다.
- Greedy는 Time이 122가 나왔습니다.
- 최단 경로는 106입니다.

Algorithm	Time
DFS	161
Greedy	122
A*	150
BFS	232
IDS	5171

DFS algorithm

```
while input_map[x][y] != goal:
    i = i + 1
    if output_map[x][y] != 0: # past_x, past_y means parent state in tree
        past_x = output_map[x][y][1]
        past_y = output_map[x][y][2]

    if y > 0: # left
        if input_map[x][y-1] != 1:
            if past_x != x or past_y != y-1: # For forbidding to go back parent state
                if output_map[x][y-1] == 0:
                    stack.append([x,y-1])
                    output_map[x][y-1] = [0,x,y]

    if x > 0: # up
        if input_map[x-1][y] != 1:
            if past_x != x-1 or past_y != y:
                if output_map[x-1][y] == [0,x,y] or output_map[x-1][y] == 0:
                    stack.append([x-1,y])
                    output_map[x-1][y] = [1,x,y]

    if x < size_x-1: # down
        if input_map[x+1][y] != 1:
            if past_x != x+1 or past_y != y:
                if output_map[x+1][y] == [1,x,y] or output_map[x+1][y] == [0,x,y] or output_map[x+1][y] == 0:
                    stack.append([x+1,y])
                    output_map[x+1][y] = [2,x,y]

    if y < size_y-1: # right
        if input_map[x][y+1] != 1:
            if past_x != x or past_y != y+1:
                if output_map[x][y+1] != [3,x,y]:
                    stack.append([x,y+1])
                    output_map[x][y+1] = [3,x,y]
```

DFS algorithm

- `Output_map[x][y]`는 현재 state가 `[x,y]`에 있을때, Parent state의 좌표와 Parent state에서의 이동방향을 나타냅니다.
- Child state에서는 Parent state로 이동할 수 없게 만들었습니다. (#left에서 `if past_x != x or past_y != y-1` 부분)
- Parent state에서 한 번 이동한 적 있는 Child state는 다시 갈 수 없도록 만들었습니다.

(#up에서 `if output_map[x-1][y] == [0,x,y]` or `output_map[x-1][y] == 0`: 부분)

위 그림은 위의 두 가지 규칙을 지키면서 stack에 앞으로 갈 state를 등록하는 작업을 나타낸 그림입니다.

입력 받은 목적지에 도달하기 전까지는 While문을 통해 계속 작동합니다.

DFS algorithm

```
        if len(stack) == 0:
            return -1
        state = stack.pop()
        x = state[0]
        y = state[1]

        answer = [x,y,i+1]
        if goal == 6:
            input_map[x][y] = 5

        j = 0
        while state != initial_state:
            x = state[0]
            y = state[1]
            state[0] = output_map[x][y][1]
            state[1] = output_map[x][y][2]
            if state != initial_state:
                input_map[state[0]][state[1]] = 5
            j = j + 1
        answer.append(j)

    return answer
```

- Child state의 스택 등록 작업을 모두 마쳤으면 이제 스택에서 하나를 pop 합니다. pop한 state를 현재 state로 한 뒤 앞의 Child state 등록 작업을 반복합니다.
- state가 목적지에 도달한 경우 output_map을 바탕으로 parent state를 따라 경로를 5로 바꾸어 가며 재구성합니다.

Greedy algorithm

```
def Heuristic(input_map,goal):
    size_x = len(input_map)
    size_y = len(input_map[0])
    i = 0
    k = 0
    j = 0
    l = 0

    heuristic = [[0]*size_y for i in range(size_x)]
    i = 0

    for i in range(size_x):
        for j in range(size_y):
            if input_map[i][j] == goal:
                break

    for k in range(size_x):
        for l in range(size_y):
            heuristic[k][l] = abs(k-i) + abs(l-j)

    return heuristic
```

- Greedy 알고리즘은 어떤 지점 x,y에서 goal 지점까지의 거리를 계산한 휴리스틱 함수를 바탕으로 하였습니다.

Greedy algorithm

```
if y < size_y-1: # right
    if input_map[x][y+1] != 1:
        if x != past_x or y+1 != past_y:
            if output_map[x][y][2] % 10 != 1:
                result.append([heuristic[x][y+1],x,y+1,0])

if x < size_x-1: # down
    if input_map[x+1][y] != 1:
        if x+1 != past_x or y != past_y:
            if int((output_map[x][y][2]%100)/10) != 1:
                result.append([heuristic[x+1][y],x+1,y,1])

if x > 0: # up
    if input_map[x-1][y] != 1:
        if x-1 != past_x or y != past_y:
            if int((output_map[x][y][2]%1000)/100) != 1:
                result.append([heuristic[x-1][y],x-1,y,2])

if y > 0: # left
    if input_map[x][y-1] != 1:
        if x != past_x or y-1 != past_y:
            if int((output_map[x][y][2]%10000)/1000) != 1:
                result.append([heuristic[x][y-1],x,y-1,3])
```

Greedy algorithm

```
result.sort(reverse=True)
result_state = result.pop()
state = [result_state[1],result_state[2]]
direction = result_state[3]
x = state[0]
y = state[1]
if direction == 0:
    past_x = x
    past_y = y-1
    output_map[past_x][past_y][2] += 1
elif direction == 1:
    past_x = x-1
    past_y = y
    output_map[past_x][past_y][2] += 10
elif direction == 2:
    past_x = x+1
    past_y = y
    output_map[past_x][past_y][2] += 100
elif direction == 3:
    past_x = x
    past_y = y+1
    output_map[past_x][past_y][2] += 1000
else:
    print ('error!')

if output_map[x][y] == 0:
    output_map[x][y] = [past_x,past_y,0]
```

Greedy algorithm

- Greedy 역시 DFS와 마찬가지로 1. Child에서 Parent state로 갈 수 없으며 2. 한 번 이동한 Child state로는 다시 갈 수 없도록 만들었습니다.
- result라는 스택을 통해 휴리스틱 최소값을 가지는 state를 관리하도록 하였습니다.
- Parent state에서 자신의 output_map을 입력하는 DFS와는 달리 Greedy는 result 스택에서 state가 pop된 state가 Parent state의 정보를 output_map에 입력합니다.