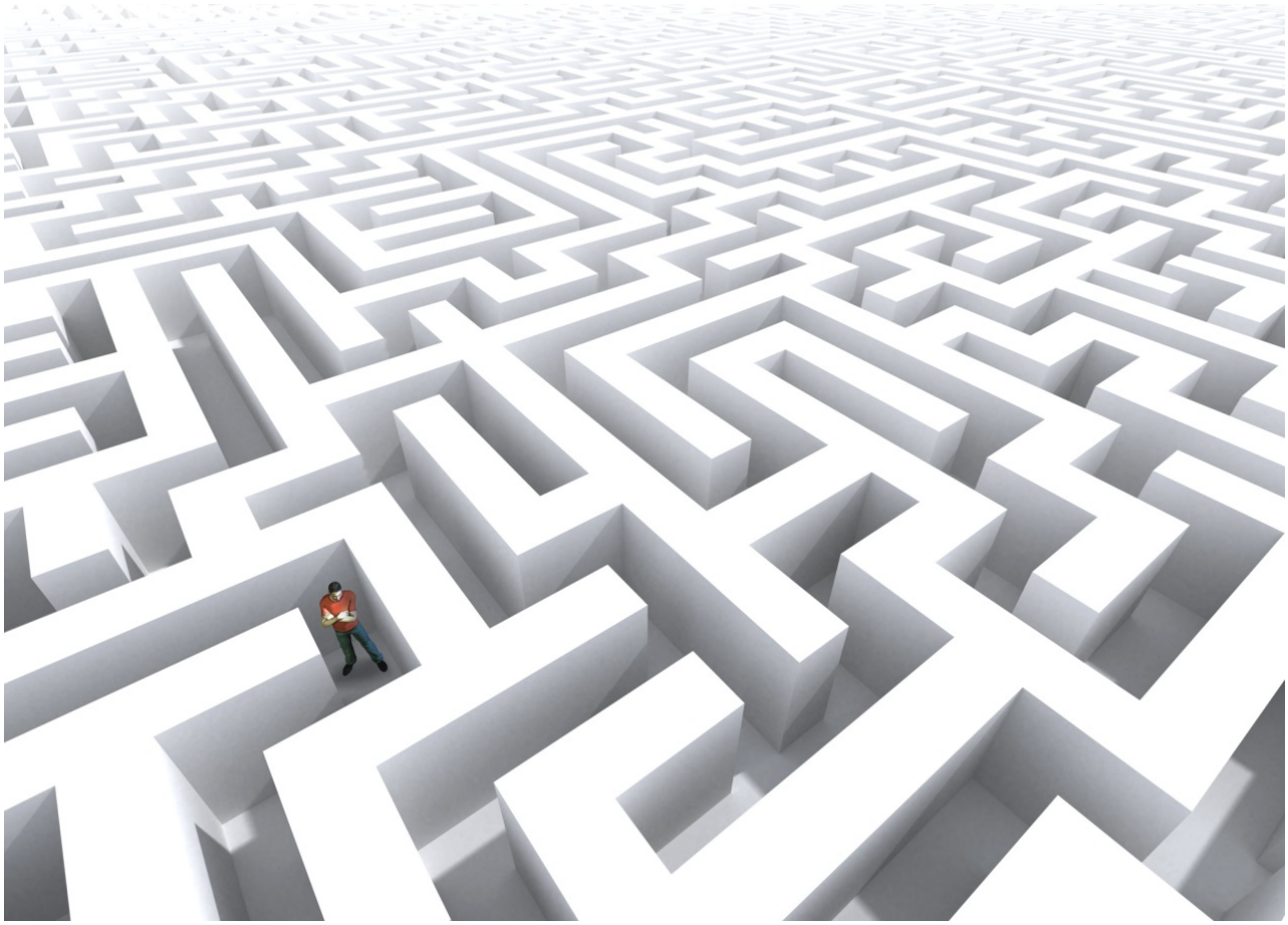
# IT/BT 탈출하기

## 18 - 2 인공지능 과제 1

2014004693 송보석



# 목 차

# 코드 설명 _ 개요

- 임의의 미로 데이터를 파일입력으로 입력 받으면 몇 층의 데이터인지 확인 후 각 층에 맞는 알고리즘을 이용해 시작점으로 부터 도착점까지의 최단거리, 총 탐색비용을 알려주는 프로그램
- 탐색 알고리즘은 DFS, BFS, A*, Greedy 총 4개의 알고리즘을 사용했으며 각 층 별 최적 알고리즘은 4개의 알고리즘을 모두 돌려본 뒤 최소의 비용으로 탐색을 하는 알고리즘으로 지정했다.
- 각 알고리즘에서 다음 맵을 탐색하는 방향은 여러 케이스를 테스트 해본 뒤 최소의 비용으로 탐색하는 좌하우상으로 지정했다.
- 각 알고리즘에서 사용할 Queue를 import 했고 maze 데이터를 복사하기 위해 deepcopy 를 import 해서 사용한다.
- 동작방식은
    1. floor에 층 수를 지정해준 뒤 파일 입력으로 *_floor_input.txt를 입력
    2. 데이터로부터 층 수 , 지도의 크기, 그리고 미로 데이터(시작점, 도착점, 키의 위치, 미로의 형태) 를 추출
    3. MapExplore 객체를 생성 후 해당 객체에서 WhichFloor 함수를 호출해 미로를 탐색
    4. 해당 탐색 결과를 파일출력으로 *_floor_ouput.txt 파일을 만들어 출력

# 코드 설명 _ MapExplore Class

```
class MapExplore:
    #상하좌우로 탐색할 때 location을 변경해주는 list
    #좌하우상 순서
    move = [(-1, 0), (0, -1), (1, 0), (0, 1)]

    #main에서 추출한 maze data로 클래스 초기화
    def __init__(self, floor_num, maze, M, S, E, key):
        self.floor_num = floor_num
        self.maze = maze
        self.visited = deepcopy(maze)
        self.S = S
        self.E = E
        self.key = key
        self.length = 0
        self.time = 0
        self.out_maze = deepcopy(maze)
        self.find_key = False
        self.path = deepcopy(maze)

    #main에서 가져온 floor number를 통해 올바른 층으로 보내줌
    def WhichFloor(self):
        if self.floor_num == 1:
            self.first_floor()
        elif self.floor_num == 2:
            self.second_floor()
        elif self.floor_num == 3:
            self.third_floor()
        elif self.floor_num == 4:
            self.fourth_floor()
        else:
            self.fifth_floor()

    #path를 추적해 최단경로를 5로 바꿔주는 함수
    def PrintMap(self, start):
        idx_x = start[0]
        idx_y = start[1]
        while self.out_maze[idx_x][idx_y] != 3 and self.out_maze[idx_x][idx_y] != 5:
            if self.maze[idx_x][idx_y] != 4:
                self.out_maze[idx_x][idx_y] = 5
            prev = self.path[idx_x][idx_y]
            idx_x = prev[0]
            idx_y = prev[1]

    #A*, greedy 알고리즘을 위한 heuristic 함수, 맨하탄 거리를 이용
    def heuristic(self, a, b):
        x1, y1 = a
        x2, y2 = b
        return abs(x1 - x2) + abs(y1 - y2)

    #각 floor에서 해당되는 알고리즘 함수를 호출
    def first_floor(self):
        self.DFS()

    def second_floor(self):
        self.DFS()

    def third_floor(self):
        self.Greedy()

    def fourth_floor(self):
        self.Greedy()

    def fifth_floor(self):
        self.Greedy()

    #알고리즘 함수
    def DFS(self): ...

    def Greedy(self): ...

    def Astar(self): ...

    def BFS(self): ...
```

- 탐색 순서는 클래스 변수로 좌(-1, 0), 하(0, -1), 우(1, 0), 상(0, 1) 순서인 리스트로 설정

❖ **__init__(self, floor_num, maze, M, S, E, key)**

- 초기화 메서드로 메인에서 가져온 미로 데이터를 객체 변수에 저장해준다.
- 이미 지나간 노드인지를 확인하는 visited, 탐색 후 경로를 표시해줄 out_maze, 탐색 경로를 저장해줄 path는 기존 maze 데이터를 deepcopy 해서 선언한다.
- 최단거리 length, 총 탐색 거리 time은 0으로 초기화하고, key를 찾았는지 나타내는 bool형 변수 find_key 는 False로 초기화해놓는다.

❖ **WhichFloor(self)**

- floor_num을 확인해 올바른 층으로 보내주는 함수로 if문을 이용해 floor_num에 해당하는 *_floor() 메서드를 호출한다.

❖ **PrintMap(self, start)**

- 맵탐색이 끝난 뒤 탐색한 최단거리를 표시해주는 메서드이다.
- 동작방식은
  - 맵을 탐색하면서 자신이 어느 노드에서부터 왔는지 리스트에서 저장하면서 탐색을 한다. (path list)
  - 처음 인덱스는 매개변수로 넘겨받은 시작지점(start)의 x, y 좌표로 초기화 한다.
  - while문을 돌면서 자신이 온 노드가 시작점(3), 도착점(4)이거나 이미 표기한 지점(5) 이 아니라면 out_maze에서 통로(2)를 지나갔다는 의미로 5로 값을 바꿔준다.
  - 자신이 온 노드로 index를 재지정 후 반복한다. 시작점(3)이나 이미 표기한지점(5)에 도착하면 종료한다.

❖ **heuristic(self, a, b)**

- Greedy, A* 알고리즘에서 사용하는 heuristic 함수이다. a, b 지점의 manhattan distance를 리턴해준다.

❖ ***_floor(self)**

- 사전에 알아낸 층 별 최적의 알고리즘 메서드를 호출한다.

❖ **DFS(self), Greedy(self), Astar(self), BFS(self)**

- 알고리즘 메서드. 상세 내용은 후술

# 코드 개요 _ Main

```python
#input data 읽어오기. 원하는 floor를 설정하면 해당하는 층의 maze data를 받아온다
floor = "fourth"
input_source = floor + "_floor_input.txt"
with open(input_source, 'r') as f:
    data = f.readlines()

#input data에서 map number, map size 추출
line = list(map(int, data.pop(0).split(" ")))
floor_num = line[0]
M = line[1]

#input data에서 maze data 추출
maze = []
for idx, line in enumerate(data):
    lline = list(map(int, line.split(" ")))
    if(6 in lline):
        idx_x, idx_y = (idx, lline.index(6))
    maze.append(list(map(int, line.split(" "))))

#key, starting point, end point 좌표 추출
key = (idx_x, idx_y)
S = (0, maze[0].index(3))
E = (M-1, maze[M-1].index(4))

#MapExplore 객체를 만든 뒤 WhichFloor 함수 호출
result = MapExplore(floor_num, maze, M, S, E, key)
result.WhichFloor()

#output 데이터 생성
output_source = floor + "_floor_output.txt"
with open(output_source, 'w') as f:
    for line in result.out_maze:
        f.write(" ".join(map(str,line)))
        f.write("\n")
    f.write(f"---\nlength = {result.length}\ntime = {result.time}\n")
```

- 미로데이터를 파일입력으로 가져온 뒤 데이터 추출, MapExplore 객체 생성 뒤 WhichFloor 메서드를 호출해 탐색 후 나온 결과를 파입출력으로 내보내는 역활을 한다.
- floor 문자열에 호출할 층수를 입력해 초기화한 뒤 open 함수를 이용해 읽기전용으로 “{floor}_floor_input.txt” 형태의 텍스트 파일을 읽어온다. 단 input 텍스트파일은 코드와 같은 경로 안에 있어야 한다. 읽어올때는 readlines() 함수를 사용해 모든 데이터를 한번에 리스트 문자열 형태로 받아와 data 변수에 저장한다.

- data 첫줄에는 층수, 행렬의 크기가 저장되어 있으므로 pop 함수로 받아와 int형으로 변환시킨 뒤 각각 floor_num, M에 저장한다.
- data 리스트의 각각 원소는 한줄의 문자열 데이터이므로 map과 split 함수를 이용해서 각 미로데이터를 원소로 하는 리스트로 만들어 maze 변수에 append 해준다.
  ex) [“111321111”] -> [1,1,1,3,2,1,1,1,1]
- 이와 동시에 각 줄에 키(6)이 있는지 확인 후 있다면 key 튜플에 좌표형식으로 저장해준다.
- S는 Start point, E는 End point를 나타내는 튜플형 변수로 첫줄과 마지막 줄에 있는 3, 4를 찾아 역시 좌표형식으로 저장해준다.
- result에 floor_num, maze, M, S, E, key를 가지고 MapExplore 객체를 생성한뒤, 실제적으로 탐색을 하는 WhichFloor 메서드를 호출해 탐색을 시킨다.
- result에 탐색 후 변수들이 저장되어 있으므로 이를 이용해 파일출력을 해준다. 입력과 같이 “{floor}_floor_ouput.txt” 텍스트파일을 open함수로 쓰기전용으로 연 뒤 데이터를 입력해준다. 각 리스트에 원소로 되어있는 map data를 문자열로 다시 바꿔 입력한뒤 length, time을 순서대로 입력한다.

# 코드 개요 _ Each Algorighm

- 총 4개(DFS, Greedy, Astar, BFS) 알고리즘을 만들었고 각 알고리즘은 맵을 탐색하는 형태는 비슷하나 어느 노드부터 방문하게 만드는 방법이 다르다.

## ❖ DFS

```
def DFS(self):
    #DFS
    st = queue.LifoQueue()
    st.put((self.S, self.length))

    while not   st.empty():
        now =   st.get()
        self.time += 1
        now_location = now[0]
        now_length = now[1]

        #Enging point에 도착했을때
        if(now_location == self.E and self.find_key == True):
            self.length = now_length
            self.time -= 2
            self.PrintMap(E)
            break;
        #key를 찾았을 때
        elif(now_location == self.key and self.find_key == False):
            self.find_key = True
            self.visited = deepcopy(self.maze)
            self.PrintMap(key)
            while not   st.empty():
                st.get()
            st.put((self.key, now_length))
        #그 외의 경우
        else:
            idx_x = now_location[0]
            idx_y = now_location[1]
            self.visited[idx_x][idx_y] = 5
            for m in MapExplore.move:
                idx_x = now_location[0]+ m[0]
                idx_y = now_location[1]+ m[1]
                next_length = now_length + 1

                if(idx_x < 0 or idx_x > M-1 or idx_y < 0 or idx_y > M-1 or self.maze[idx_x][idx_y] == 1 or self.visited[idx_x][idx_y] == 5):
                    pass
                else:
                    next_location = (idx_x, idx_y)
                    self.path[idx_x][idx_y] = now_location
                    st.put((next_location, next_length))
```

- Depth First Search 는 길이 없거나 목적지에 도착할 때까지 깊이 들어가며 탐색하는 방식으로 자료구조는 LifoQueue를 이용했다. LifoQueue는 후입선출의 특징을 가지는 자료구조다.
- 먼저 st에 LifoQueue를 선언한 뒤 (시작점, 최단거리) 튜플을 put 해준다.
- while문을 이용해 queue가 empty할때 까지 loop를 돌린다.
- 먼저 queue 부터 get 함수로 인자를 받아와 now에 저장을 해준다. 이 때 현재 위치인 now_location은 now[0], 현재 최단거리인 now_length는 now[1]이 된다. 큐에서 하나씩 원소나 나올때마다 그 노드를 방문했다는 의미로 볼 수 있으므로 time을 하나 증가시켜준다.

- 만약 find_key가 True(키를 찾았고)이고 now_location이 E 이면 최단거리를 찾은 상태이므로 미로의 최단거리를 나타내는 객체변수 length에는 now_length를 저장하고 PrintMap(E)를 이용해 도착점으로 부터 열쇠까지의 최단거리를 지도에 표시해 준다. 총 탐색거리인 time을 -2하는 이유는 queue에서 뺄때마다 무조건 1을 더해주다보니 출발점과 키에서 출발할 때 중복으로 time을 올리기 때문이다.
- find_key가 False(키를 찾지못했고)이고, now_location이 key의 위치이면, 처음으로 키를 발견한 상황이므로 find_key를 True로 바꿔주고 key의 위치에서 부터 다시 탐색을 재시작하기 위해서 visited 변수를 초기화하고 queue를 비워준다. 그리고 (key, now_length)를 큐에 넣어서 맨 처음 상황과 똑같이 만든다. 그리고 시작점으로부터 키의 위치까지 최단 경로를 PrintMap(key)로 지도에 표시해준다.
- 그 외의 상황은 미로를 일반적으로 탐색하는 상황이므로 다음 노드로 진행을 시켜준다. 먼저 현재 노드의 visited를 2에서 5로 변경해 이미 방문했다고 표시하고 for문을 이용해 미리 지정해 놓은 방문 순서대로 index를 변경시켜 다음 노드를 next_location에 저장한다. next_length에도 now_length에 1을 더해 값을 변경한다. 만약 next_location이 맵 밖으로 나가거나, 벽(1)이거나 이미 방문한 노드(5) 라면 무시하고, 그렇지 않다면 (next_location, next_length) 형태로 큐에 put해준다. 그리고 탐색 종료 후 backtracking을 위해서 next_location 이전 노드가 now_location임을 path에 저장해준다.

```
def Greedy(self):
    #greedy
    qu = queue.PriorityQueue()
    priority = self.heuristic(self.S, self.key)
    qu.put((priority, self.S, self.length))

    while not qu.empty():
        now = qu.get()
        self.time += 1
        now_location = now[1]
        now_length = now[2]

        #Enging point에 도착했을때
        if(now_location == self.E and self.find_key == True):
            self.length = now_length
            self.time -= 2
            self.PrintMap(E)
            break;
        #key를 찾았을 때
        elif(now_location == self.key and self.find_key == False):
            self.find_key = True
            self.visited = deepcopy(self.maze)
            self.PrintMap(key)
            while not qu.empty():
                qu.get()
            priority = self.heuristic(key, E)
            qu.put((priority, self.key, now_length))
        #그 외의 경우
        else:
            idx_x = now_location[0]
            idx_y = now_location[1]
            self.visited[idx_x][idx_y] = 5
            for m in MapExplore.move:
                idx_x = now_location[0]+ m[0]
                idx_y = now_location[1]+ m[1]
                next_length = now_length + 1

                if(idx_x < 0 or idx_x > M-1 or idx_y < 0 or idx_y > M-1 or self.maze[idx_x][idx_y] == 1 or self.visited[idx_x][idx_y] == 5):
                    pass
                else:
                    next_location = (idx_x, idx_y)
                    self.path[idx_x][idx_y] = now_location
                    if self.find_key == False:
                        priority = self.heuristic(next_location, key)
                    else:
                        priority = self.heuristic(next_location, E)
                    qu.put((priority, next_location, next_length))
```

❖ **Greedy**

- Greedy는 heuristic 함수를 이용해 목적지까지 최단거리일 것이라고 예상되는 노드부터 먼저 탐색하는 방식이다. 이를 위해 자료구조느 PriorityQueue를 이용했다. 따라서 큐에 인자를 넣어줄때 (node, length)에 우선순위를 추가해서 put해준다. priority변수는 우선순위를 지정해주는 변수로 열쇠를 찾기 전에는 현재위치와 열쇠까지의 heuristic 값, 열쇠를 찾은 후에는 현재위치와 목적지까지의 heuristic 값이다.
- priority가 높은 노드부터 queue에서 get해서 탐색하는 것 이외에는 DFS와 탐색방법이 동일하다.

❖ A*

```
def Astar(self):
    #Astar
    qu = queue.PriorityQueue()
    priority = self.length + self.heuristic(self.S, self.key)
    qu.put((priority, self.S, self.length))

    while not qu.empty():
        now = qu.get()
        self.time += 1
        now_location = now[1]
        now_length = now[2]

        #Enging point에 도착했을때
        if(now_location == self.E and self.find_key == True):
            self.length = now_length
            self.time -= 2
            self.PrintMap(E)
            break;
        #key를 찾았을 때
        elif(now_location == self.key and self.find_key == False):
            self.find_key = True
            self.visited = deepcopy(self.maze)
            self.PrintMap(key)
            while not qu.empty():
                qu.get()
            priority = now_length + self.heuristic(key, E)
            qu.put((priority, self.key, now_length))
        #그 외의 경우
        else:
            idx_x = now_location[0]
            idx_y = now_location[1]
            self.visited[idx_x][idx_y] = 5
            for m in MapExplore.move:
                idx_x = now_location[0]+ m[0]
                idx_y = now_location[1]+ m[1]
                next_length = now_length + 1

                if(idx_x < 0 or idx_x > M-1 or idx_y < 0 or idx_y > M-1 or self.maze[idx_x][idx_y] == 1 or self.visited[idx_x][idx_y] == 5):
                    pass
                else:
                    next_location = (idx_x, idx_y)
                    self.path[idx_x][idx_y] = now_location
                    if self.find_key == False:
                        priority = next_length + self.heuristic(next_location, key)
                    else:
                        priority = next_length + self.heuristic(next_location, E)
                    qu.put((priority, next_location, next_length))
```

- A*는 PriorityQueue를 이용해 priority가 높은 노드부터 방문해 탐색한다는 기본적인 내용은 Greedy와 동일하지만 priority가 좀 다르다. Greedy는 단순히 거리가 더 가까울 것으로 예상되는 노드를 먼저 탐색하지만 A*는 현재까지의 거리(now_length)와 목적지까지의 heuristic 값의 합이 작은 노드부터 탐색한다.

- 그 외의 과정은 Greedy와 동일하다.

❖ **BFS**

```
def BFS(self):
    #BFS
    qu = queue.Queue()
    qu.put((self.S, self.length))

    while not qu.empty():
        now = qu.get()
        self.time += 1
        now_location = now[0]
        now_length = now[1]

        #Enging point에 도착했을때
        if(now_location == self.E and self.find_key == True):
            self.length = now_length
            self.time -= 2
            self.PrintMap(E)
            break;
        #key를 찾았을 때
        elif(now_location == self.key and self.find_key == False):
            self.find_key = True
            self.visited = deepcopy(self.maze)
            self.PrintMap(key)
            while not qu.empty():
                qu.get()
            qu.put((self.key, now_length))
        #그 외의 경우
        else:
            idx_x = now_location[0]
            idx_y = now_location[1]
            self.visited[idx_x][idx_y] = 5
            for m in MapExplore.move:
                idx_x = now_location[0]+ m[0]
                idx_y = now_location[1]+ m[1]
                next_length = now_length + 1

                if(idx_x < 0 or idx_x > M-1 or idx_y < 0 or idx_y > M-1 or self.maze[idx_x][idx_y] == 1 or self.visited[idx_x][idx_y] == 5):
                    pass
                else:
                    next_location = (idx_x, idx_y)
                    self.path[idx_x][idx_y] = now_location
                    qu.put((next_location, next_length))
```

- Breadth First Search는 현재 노드에서 가장가까운 노드부터 순차적으로 탐색한다는 점에서 DFS와는 반대되는 특성을 가지고 있다. 따라서 자료구조는 선입선출인 Queue를 이용한다.
- Queue를 이용해서 가까운 노드부터 탐색한다는 점을 제외하고는 일반적인 탐색과정은 DFS와 동일하다.

# 층 별 사용 알고리즘

- 각 층 별 사용 알고리즘은 위의 4가지 알고리즘을 각 미로의 모두 적용해 본 뒤 탐색노드의 수가 제일 작은 알고리즘을 채택했다.
- 이는 각 알고리즘을 적용했을 때 length와 time를 구한 표이다.

| | first floor | | second floor | | third floor | | fourth floor | | fifth floor | |
|---|---|---|---|---|---|---|---|---|---|---|
| DFS | length | 3850 | length | 758 | length | 554 | length | 334 | length | 106 |
| | time | 5091 | time | 850 | time | 733 | time | 461 | time | 159 |
| Greedy | length | 3850 | length | 758 | length | 554 | length | 334 | length | 106 |
| | time | 5813 | time | 992 | time | 653 | time | 430 | time | 120 |
| A* | length | 3850 | length | 758 | length | 554 | length | 334 | length | 106 |
| | time | 6603 | time | 1612 | time | 816 | time | 561 | time | 156 |
| BFS | length | 3850 | length | 758 | length | 554 | length | 334 | length | 106 |
| | time | 6746 | time | 1722 | time | 1006 | time | 595 | time | 233 |

- 확인 결과 1층과 2층은 DFS가 나머지층은 Greedy가 가장 빠르게 최단거리를 찾아냄을 알 수 있다.