Assignment1 보고서

IT/BT 탈출하기!

2016025496 서유림

1. 코드 설명

```
class Point(object):

    def __init__(self, x=0, y=0):
        self.x = x;
        self.y = y;

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)
    #좌표 + 해줄때 쓸거.

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
    # 좌표 그냥 =로 대입할 때 쓸거.


    def __gt__(self, other):
        return self if self.x > other.x else other



dir = [Point(0, -1), Point(0, 1), Point(1, 0), Point(-1, 0)]
class Maze():
    m = 0
    n = 0
    matrix = []
    time = 0
    length = 0

    def __init__(self, _m, _n, _matrix):
        self.m = _m;
        self.n = _n;

        self.matrix = copy.deepcopy(_matrix);
        # 배열은 deepcopy해야 값이 다 복사 됨.
```

Point class는 좌표를 표현하는 class이다. init 함수로 객체 생성할 때 객체의 x좌표와 y좌표를 받아와서 생성한다. x와 y의 값은 기본 값을 0으로 한다.
그리고 좌표끼리 +, ==
연산과 같이, 각 x, y 좌표를 더한값을 넘겨주거나 비교한 결과를 넘겨주기 위해서 add, eq 함수를 정의하였다.
greater 함수는 나중에 priority queue를 쓰기 위해 heapq를 사용할 때 Point 객체끼리 대소비교를 해야 하는 때가 있어서 그냥 x값 기준으로 큰걸 리턴한다.

dir 리스트 각 노드의 상하좌우 노드를 탐색 할때 x좌표와 y좌표에 각각 더해줘야 하는 값을 Point 형태로 넣어 놓은 것이다.

Maze 클래스는 각 search 함수에 넘겨줄 행렬의 가로 세로 크기, 구해야 하는 time , length, matrix 를 한번에 넘겨주고 받기 위해서 만든 것 이다.

```
def main():
    naming = [" ", "first", "second", "third", "fourth", "fifth"]
    matrix = []
    funcdict = {
        1: first_floor,
        2: second_floor,
        3: third_floor,
        4: fourth_floor,
        5: fifth_floor
    }

    m = 0
    n = 0
    try:
        floor = int(input("Choose the floor want to escape: "))
        temp = floor
        f = open(naming[floor] + "_floor_input.txt", "r")
        floor, m, n = map(int, f.readline().split())
        #맨 첫줄 읽어서 몇 층인지랑 map크기 받아옴.

        #키의 위치, 시작점, 도착점을 저장할 튜플
        key = [-1, 0]
        st = [-1, 0]
        ed = [-1, 0]


        while True:
            line = f.readline()#한 줄 식 입력 받음.
```

맨 처음에 내가 탈출할 층을 입력하면 그 int 값에 해당하는 dictionary 값을 가져와 input 파일을 읽어온다.

그리고 한줄 씩 읽는다.

```
while True:
    line = f.readline()#한 줄 식 입력 받음.

    if(key[1] == 0): key[0] += 1

    if(st[1] == 0): st[0] += 1

    if(ed[1] == 0): ed[0] += 1

    if not line: break
    temp_list = list(map(int, line.split()))
    #입력 받은 한 줄을 int map 형태로 바꿈.
    if(6 in temp_list): key[1] = temp_list.index(6)
    #입력 받으면서 키 위치를 찾는다.

    if(3 in temp_list): st[1] = temp_list.index(3)
    #입력 받으면서 시작점을 찾는다.

    if(4 in temp_list): ed[1] = temp_list.index(4)
    #입력 받으면서 도착점을 찾는다.
    matrix.append(temp_list)
f.close()
```

한줄 씩 읽으면서 내가 읽은 줄에 6이 있으면 거기가 key가 있는 좌표니까, 그 좌표를 key 튜플에 저장한다.

같은 원리로 4가 입력되면, 도착점이니까 ed 튜플에 저장하고

같은 원리로 3이 입력되면, 시작점이니까 st 튜플에 저장한다.

```
#시작 좌표, key 좌표, goal 좌표를 Point 형태로 만든다
start = Point(st[1], st[0])
goal1 = Point(key[1], key[0])
goal2 = Point(ed[1], ed[0])


_maze = Maze(m, n, matrix)

funcdict[floor](start, goal1, goal2, _maze)

#미로 탐색 결과를 파일에 출력
f = open(naming[floor] + "_floor_output.txt","w")
for i in range(_maze.m) :
    line = map(str, _maze.matrix[i])
    f.write(" ".join(line))
    f.write("\n")
f.write("---\n")
f.write("length=%d\n"%(_maze.length))
f.write("time=%d\n"%(_maze.time))
f.close()
```

start 에 튜플 값을 넘겨서 Point 객체로 만든다.
goal1 은 key의 좌표 값을 갖고
goal2는 도착점을 좌표 값을 갖는다.

그리고 내가 입력받은 층을 탈출하는 함수를 호출한다.

그리고 그 결과를 층_floor_output.txt 에 저장한다.

```
def first_floor(start, goal1, goal2, _maze):

    #bfs(start, goal1, _maze)
    #bfs(goal1, goal2, _maze)

    dfs(start, goal1, _maze)
    dfs(goal1, goal2, _maze)

    #a_star(start, goal1, _maze)
    #a_star(goal1, goal2, _maze)

    #greedy_best(start, goal1, _maze)
    #greedy_best(goal1, goal2, _maze)
```

층에 해당하는 함수를 보면, start와 goal1, goal2, _maze 객체를 넘겨받는다.
그리고 그 안에서
시작점부터 키 까지 경로를 찾고.
그 다음 키 부터 도착점까지 경로를 찾는다.

! 서치 알고리즘 설명

```
def heuristic(start, goal) :
    return abs(start.x - goal.x) + abs(start.y - goal.y)
```

greedy best first search와 a* 알고리즘에서 사용하기 위한 heuristic 함수이다. manhattan distance를 이용하였다.

모든 서치 알고리즘은 기본적으로 같은 방식으로 작동한다. 다른 점은, heuristic 함수를 사용하거나 각각 다른 자료구조를 사용한다는 것이다.

A*알고리즘의 작동 방식에 대해서 설명해 보겠다.

```python
def a_star(start, goal ,_maze):

    #힙큐를 이용함.
    pq = []

    #방문했던 노드는 1로 표시할거기 때문에, vis라는 방문 체크 배열을 만들어서 다 0으로 초기화
    vis = [[0 for j in range(_maze.n)] for i in range(_maze.m)]
    #찾았는지 확인하기 위한 bool 변수
    found = False

    # 최단경로를 알아내기 위해 각 노드의 부모 노드가 뭔지 저장하는 배열
    parent = [[Point() for j in range(_maze.n)] for i in range(_maze.m)]

    #현재까지 실제 cost 저장할 배열
    cost = [[0 for j in range(_maze.n)] for i in range(_maze.m)]


    heappush(pq, (0,start))
    vis[start.y][start.x] = 1;

    cnt = 1 # time을 구하기 위한 변수. 시작점부터 무조건 탐색하니까 1부터 시작

    while len(pq) > 0 :

        now = heappop(pq)[1]

        cost[now.y][now.x] = cnt


        # 상하좌우로 탐색
        for i in dir :
            # 다음 좌표
            next = now + i

            # 좌표가 범위를 벗어나면 continue로 넘어간다
            if next.x < 0 or next.x >= _maze.n or next.y < 0 or next.y >= _maze.m :
                continue

            # goal 찾았으니까 탈출! 그리고 goal의 부모도 표시하고
            if next == goal:
                found = True
                pq.clear()
                parent[next.y][next.x] = now
                break

            # 아직 방문 안했는데 통로면 pq에 넣는다.
            if _maze.matrix[next.y][next.x] !=1 and vis[next.y][next.x] == 0 :

                vis[next.y][next.x] = 1
                priority = cost[now.y][now.x] + heuristic(next, goal)
                heappush(pq, (priority, next))
                cnt += 1
                parent[next.y][next.x] = now # trace를 위한 부분. 부모가 뭐였는지 기록
```

pq라는 리스트를 만들고 그 리스트를 heqpq라는 모듈을 이용해 priority queue처럼 사용한다. heapq는 기본적으로 min heqp이기 때문에 heapq를 사용하면 된다.

각 노드를 방문했는지 체크하기 위한 vis배열을 m*n 만큼 만들고, 각각 노드의 부모노드가 무엇인지 저장하기 위한 parent크기도 같은 크기로 만든다.

a*알고리즘은 manhattan distance와 현재까지의 실제 탐색 길이를 더한 값을 기준으로 priority를 정한다.
그래서 각 노드별로 cost 값을 저장하기 위한 cost 배열도 m*n 크기 만큼 만든다.

맨 처음 시작하는 노드를 pq에 넣고 pq가 빌때 까지 반복문을 돌린다.

cnt는 내가 탐색하는 노드의 수 즉 time을 구하기 위한 변수이다. 새로 pq에 append할 때마다 +1 해준다. 그리고 pq에서 pop할 때마다 지금 팝한 노드 까지의 cost를 정해줘야한다.

cost[now.y][now.x] 에 현재까지 노드 탐색 수인 cnt값을 집어넣는다.

그리고 dir리스트 안의 Point값을 이용해 나에게 연결된 상하좌우 노드를 본다.
다음 노드가 m*n의 범위에서 벗어나면 continue로 밑에 부분을 실행하지 않고 넘어간다.

그 후에 다음에 탐색할 노드가 만약 골이라면, found 변수를 True로 바꿔서 도착점을 찾은 것을 표시한다.
그리고 pq를 비우고, 다음 노드의 부모로 현재 노드를 지정해주고 반복문을 빠져 나간다. 계속 부모노드를 저장하는 이유는 나둥에 trace해서 최단경로를 구해야 하기 때문이다.

goal을 못 찾았다면, 다음 노드가 내가 방문을 안 했고, 방문 가능한 노드인지 판단후 pq에 넣어줘야 한다.
이 다음 노드를 방문할거니까, vis배열을 체크해주고, heuristic 함수를 이용해 priority를 구한다.
a*에서의 priority는 현재 노드까지의 cost와 goal까지의 manhattan distance를 구한 값을 이용해서 구한다. 작은 값이 더 높은 우선순위를 가진다.
새롭게 구한 priority와 다음 노드를 pq에 넣어준다.
그리고 방금 넣은 노드의 부모노드도 parent 배열에 표시해준다.

a* 알고리즘은 위와 같은 방식으로 작동을 한다.
dfs는 a* 알고리즘에서 자료구조만 stack으로 바꾸고, cost와 priority개념만 빠지면 된다.
bfs도 자료구조만 queue로 바꾸고, cost, priority 개념만 빠지면 된다.
greedy best first search 알고리즘은 priority 값 정하는 기준을 manhattan distance로만 정하면 된다.

최단 경로를 표시하기 위해서 각 search 알고리즘 끝 부분에

```
path = []

# 만약 찾으면?
if found :
    # goal부터 시작해서 parent를 저장해둔 배열을 이용해서 trace를 시작한다.
    p = goal
    while p != start :
        # 시작점과 도착점은 5로 바꾸지 않는다.
        if p!=start and _maze.matrix[p.y][p.x] !=4 :
            _maze.matrix[p.y][p.x] = 5
        path.append(p)
        p = parent[p.y][p.x]
    path.append(p)
    path.reverse() # 최단 경로 생성
    _maze.length += len(path)-1
    _maze.time += cnt
else :
    print("Can't find path...")
```

이런 식으로 goal 노드 부터 부모를 추적하면서, path에 추가한다.

그리고 path를 reverse시키면 최단 경로가 완성이 된다.

2. 사용 알고리즘

일단 length가 최소가 되도록 하는게 우선이기 때문에 optimal을 보장해주는 bfs 함수를 이용해서 각 층의 optimal length를 구해보았다.

	1층	2층	3층	4층	5층
bfs	3850 | 6745	758 | 1719	554 | 1001	334 | 595	106 | 231
	length | time				

그리고 나서 dfs, greedy best first search, a* search 알고리즘으로 optimal length를 구할 수 있는지 보았다. IDS 알고리즘은 어차피 dfs를 반복하는 알고리즘 이기 때문에 time이 무조건 dfs로 찾는 것보다 클테니까, optimal을 보장해 주더라도 time면에서는 최선의 선택이 아니기 때문에 테스트 하지 않았다.

dfs, greedy best first search, a* search 알고리즘으로 테스트해본 결과 모두 optimal한 length값을 가져올 수 있었다. 그래서 각 층마다 4가지 알고리즘을 모두 사용해보고 time이 제일 적게 나온 것을 채택하기로 하였다.
그리고 미로 찾기의 경우 현재 노드 까지의 cost 값이 결국 manhattan distance와 같기 때문에 manhattan distance를 heuristic 함수 값으로 이용하는 Greedy best first search가 time이 적게 나올 것으로 예상이 되었다. 또한 각 층 마다 통로가 길 처럼 쭉 있는 경우가 있어서, 노드를 깊이 계속 탐색하는 dfs가 특정 경우에 빠를 것으로 예상이 되었다.

확인 해본 결과

1층 : DFS
2층 ~ 5층 : Greedy best first search

층마다 time이 가장 짧게 나오는 알고리즘은 위와 같다.

3. 실험 결과

4개의 알고리즘을 층별로 확인해본 결과입니다.

	1층	2층	3층	4층	5층
bfs	3850 | 6745	758 | 1719	554 | 1001	334 | 595	106 | 231
dfs	3850 | 5659	758 | 1051	554 | 1546	334 | 506	106 | 318
greedy best first	3850 | 5847	758 | 1025	554 | 675	334 | 440	106 | 126
a*	3850 | 6156	758 | 1116	554 | 772	334 | 486	106 | 136
	length | time				