인공지능 과제 #1

미로 탈출하기

2018008059 김은수

1. 코드 설명

(1) 각 미로마다 코드 실행방법

main문에 functioncall함수를 적고, functioncall함수의 인자로 input file의 이름을 적으면 된다. (입력파일의 파일 형식을 제외한 이름만 적는다.) functioncall 함수를 통해 4개의 search 알고리즘이 실행된다.

```
def functioncall(maze : str):
    astar_run(maze)
    gbfs_run(maze)
    bfs_run(maze)
    ids_run(maze)


if __name__ == '__main__':
    functioncall('Maze_1')
    functioncall('Maze_2')
    functioncall('Maze_3')
    functioncall('Maze_4')
```

search 알고리즘을 실행하는 함수는 알고리즘 이름 옆에 run이 붙은 형태이고 이는 functioncall 함수 안에 들어 있다.

Ex. Maze_1.txt 파일을 돌리고 싶다면 main에 functioncall('Maze_1')을 적는다.

(2) 각 미로마다 코드 동작

메인에서 functioncall를 부르고 functioncall함수에서 각 search 알고리즘의 run함수를 실행시킨다. run함수에서 입력 Maze에 맞게 key, start, goal등 초기화를 해준 후 해당 search 알고리즘을 부른다. 해당 search 알고리즘을 실행시킨 후 run 함수에서는 output text를 만든다.

```
def ids_run(maze: str):
    mz = Maze(maze+".txt")
    time, length = mz.ids()
    with open(maze + '_IDS_output.txt', 'w') as f:
        for line in mz.map:
            for w in line:
                f.write(str(w))
            f.write('\n')
        f.write('---\n')
        f.write('length=' + str(length) + '\n')
        f.write('time=' + str(time) + '\n')
```

Ex. main에서 functioncall('Maze_1') 실행하면 functioncall안 astar_run, gbfs_run, bfs_run, ids_run 함수가 실행된다. 예시로 ids_run 함수를 보겠다.

ids_run함수에서 mz = Maze(maze+".txt")를 통해 해당이름의 input file을 가지고 Maze class의 init함수를 통해 초기화를 해준다. 다음으로 ids 함수를 실행시켜서 time과 length를 받는다. 다음으로 Maze_1_IDS_output.txt 파일을 만들어주고 지나간 경로와 length, time을 적어준다.

이 astar_run, gbfs_run, bfs_run, ids_run 모두 실행하는 search 알고리즘만 다를 뿐 형식과 기능은 같다.

1. fucntioncall에서 astar_run('Maze_1') 실행 -> astar_run 함수에서 a_star()알고리즘 실행 -> a_star()함수에서 bfs_helper()함수(bfs search) 실행-> 실행 후 astar_run 함수에서 해당 maze, search 알고리즘에 맞는 output text 생성
2. functioncall에서 gbfs_run('Maze_1')실행 -> gbfs_run 함수에서 gbfs()알고리즘 실행->gbfs()함수에서 bfs_helper() 함수 (bfs search)실행-> 실행 후 gbfs_run함수에서 해당 maze, search 알고리즘에 맞는 output text 생성
3. main에서 bfs_run('Maze_1')실행 -> bfs_run 함수에서 bfs() 함수 실행->bfs()함수에서 bfs_b()함수 실행->실행후 bfs_run 함수에서 해당 maze, search알고리즘에 맞는 output text 생성
4. main에서 ids_run('Maze_1') 실행 -> ids_run 함수에서 ids()함수 실행 -> ids()함수에서 dfs()함수 실행 -> 실행후 bfs_run 함수에서 해당 maze, search알고리즘에 맞는 output text 생성

2. 함수 설명

(1) 각 함수마다 사용 용도

```
class Maze:
    start = (0, 0)
    key = (0, 0)
    goal = (0, 0)
    column = 0
    row = 0
    keynumber = 0
    keylist = []
    map = []
    maze = []
    def __init__(self, filename):
        self.keylist.clear()
        self.keynumber = 0
        with open(filename, 'r') as f:
            self.map = []
            self.map.clear()
            self.maze = []
            self.maze.clear()
            n, self.column, self.row = [int(x) for x in f.readline().strip().split(' ')]
            for l in f:
                line = [int(x) for x in l.strip()]
                self.map.append(line)
                self.maze.append(line)
                for x in range(0, len(line)):
                    if line[x] == 3:
                        self.start = (len(self.map)-1, x)
                    elif line[x] == 4:
                        self.goal = (len(self.map)-1, x)
                    elif line[x] == 6:
                        self.key = (len(self.map)-1, x)
                        self.keylist.append(self.key)
                        self.keynumber += 1
```

- Class Maze와 생성자

  Maze class 안에는 start, key, goal, column, row와 key의 개수를 저장하는 keynumber, key들을 저장하는 keylist, 입력 받은 maze를 저장하는 map과 maze가 있다.

  생성자는 file을 읽어서 column, row를 할당 받는다. 또한, start, goal, key에 위치를 할당한다. Goal과 key의 위치는 informed search인 A* search와 Greedy best fit search에서만 사용된다. Key는 입력 받은 뒤 keylist라는 list에 넣어준다. 이에 따라 key의 개수인 keynumber 변수도 1증가한다.

```
def position_check(self, position):
    if position[0] < 0 or position[0] >= self.column or position[1] < 0 or position[1] >= self.row:
        return False
    if self.map[position[0]][position[1]] != 1:
        return True
    else:
        return False
```

- Position_check(self, position)

  position을 함수인자로 받아서 position의 높이가 column을 초과하거나 0이하인지, 넓이가 row를 초과하거나 0이하인지 확인해준다. 그렇다면 비정상적인 경우이므로 False를 return한다. 다음으로 이 위치가 1이 아닌지 즉 벽이 아닌지 확인해주고 벽이 아니라면 true, 벽이라면 갈 수 없으니 false를 return해준다.

```
def move(self, position, direction):
    new = (position[0] + direction[0], position[1] + direction[1])

    if self.position_check(new):
        return new

    return None
```

- move(self, position, direction)

  이후 나올 search 알고리즘에서 위치와 이동방향을 move 함수에 넣어준다. move함수에서는 인자로 받은 위치와 이동방향을 더한 후 새로운 위치로 갈 수 있는지 position_check함수를 통해 확인해준 뒤 갈 수 있다면 node를 반환해주고 갈 수 없다면 None을 반환해준다.

```
def coordinate(self):
    ret = []
    ret.clear()

    for i in range(0, self.column):
        ret.append([(i, x) for x in range(0, self.row)])

    return ret
```

- Coordinate(self)

  해당 maze의 column, row에 맞춰 좌표 배열을 만들어 준다. 이는 나중에 경로를 역

추적할 때 사용될 배열이다.

```
def tracking(self, path, goal):
    node = goal

    while path[node[0]][node[1]] != node:
        if self.map[node[0]][node[1]] == 2 or self.map[node[0]][node[1]] == 6:
            self.map[node[0]][node[1]] = 5

        node = path[node[0]][node[1]]
```

- Tracking(path, goal)

Goal 노드에 도달할 때까지 경로를 역추적하여 통로나 key일 경우, 즉 지나갈 수 있을 경우 map을 5로 바꾼다. Tracking 함수의 인자로 받는 path는 coordinate 함수로 만든 배열이다. Tracking 함수를 통해 map 배열에서 지나간 경로는 5로 바뀌며 이 배열이 나중에 output text에서 이동한 경로로 적힌다.

```
def dfs(self, start, limit, path):
    mz = copy.deepcopy(self)
    mz.map = copy.deepcopy(self.map)
    mz.maze = copy.deepcopy(self.maze)
    stack = []
    stack.clear()
    time = 0
    stack.append((start, 0, start))
    while not len(stack) <= 0:
        if limit<=0:
            return (time, False, 0)
        node = stack.pop()
        position, length, previous = node
        path[position[0]][position[1]] = previous
        mz.map[position[0]][position[1]] = 1

        if mz.maze[position[0]][position[1]] == 4:
            self.tracking(path, position)
            return (time, True, length)

        elif mz.maze[position[0]][position[1]] == 6:
            self.keynumber = self.keynumber-1
            self.tracking(path, position)
            self.start = (position[0], position[1])
            return (time, False, length)

        for direction in [(1,0),(0,1),(-1,0),(0,-1)]:
            mv = mz.move(position, direction)
            if mv is not None:
                stack.append((mv, length + 1, position))
        time = time + 1
        limit = limit - 1
    return (time, False, 0)
```

- dfs(self, start, limit, path)

  deepcopy를 통해서 maze를 복사한다. 그 다음 방문할 node를 저장할 stack을 만들어준다. Stack list는 clear를 통해 초기화를 해준다. Stack에 처음으로 인자로 받은 start점을 넣어준다. (start, 0, start)를 넣어준다. Start는 시작점이고 0은 length이다. 그 다음 stack이 비기 전까지, 또한 limit이 0이 되기 전까지 반복문을 돌려준다. Stack에서 pop을 통해 node를 하나 꺼내 주고 이를 position, length, previous에 할당해준다. Position은 위치, length는 경로 길이, previous는 이전 위치이다. 따라서 처음에 position에는 start, length에는 0, 이전 노드에는 start가 들어가게 된다. 처음에 maze를 배열로 받았던 maze 배열을 통해 해당 위치에 대한 숫자를 확인해준다. 만약 4라면 goal이므로 tracking 함수를 통해 경로를 추적해주고 time과 length 그리고 탐색이

잘되었다는 True를 반환해준다. 다음으로 4는 아니지만 6이라면 key인 것이므로 해당 위치까지의 경로도 tracking으로 알아준다. maze에 있는 key를 다 습득해야 하고 6을 발견했다는 것은 key를 하나 습득했다는 것이므로 keynumber의 수는 하나 줄여준다. 그 다음 start지점을 해당 위치로 바꿔준다. 그래야 다음번에 이 위치에서부터 다시 경로를 찾아 주기 때문이다. 그 다음 time과 False, length를 반환해준다. False를 반환하는 이유는 아직 4, 즉 goal이 아니기 때문이다. 다음으로 4나 6이 아니라면 해당 node의 위아래 좌우 노드로 갈 수 있는지 move를 통해 확인해주고 갈 수 있다면 stack에 넣어준다. 한 노드를 탐색했으므로 time은 1증가해주고 limit은 1 감소시켜준다. Limit이 0이 될 때까지, stack이 빌 때까지 goal을 찾지 못했다면 time과 length는 0, 그리고 False를 반환해준다.

```
def bfs_helper(self, start, goal, heuristic):
    mz = copy.deepcopy(self)
    mz.map = copy.deepcopy(self.map)
    q = PriorityQueue()
    time = 0
    path = self.coordinate()
    q.put((0, (start, 0, start)))

    while not q.empty():
        node = q.get()
        position, length, previous = node[1]

        path[position[0]][position[1]] = previous
        mz.map[position[0]][position[1]] = 1

        if position == goal:
            self.tracking(path, goal)
            return (time, length)

        for dr in [(1,0),(0,1),(-1,0),(0,-1)]:
            mv = mz.move(position, dr)
            if mv is not None:
                q.put((heuristic(position, goal, length + 1), (mv, length + 1, position)))

        time = time + 1
    return (0, 0)
```

- bfs_helper(self, start, goal, heuristic)

  이 함수는 heuristic function을 사용하는 A* search와 Greedy BFS search에서 BFS search 알고리즘을 실행해주는 함수이다. 시작점, goal점, 사용하는 휴리스틱 함수를 인자로 받는다. 탐색하기 위해 Queue를 PriorityQueue()를 통해 만들어준다. Queue 안에는 (우선순위, (노드, length, 노드) 가 넣어진다. Queue 가 empty가 되기 전까지

while을 통해 밑 코드를 반복한다. 우선, queue에서 node를 하나 꺼내 준다. Position이 goal이라면 tracking을 통해 경로를 추적해주고 time과 length를 리턴 해준다. Position이 goal이 아니라면 node의 위아래 좌우 노드가 갈 수 있는지 move를 통해 확인해주고 갈 수 있다면 queue에 넣어준다. 이때 우선순위 위치에 heuristic(position, goal, length+1)이 넣어지는데 이는 인자로 받은 각 search 알고리즘의 heuristic 함수 정의에 따라서 가공된다. 이에 따라 우선순위로 들어가진다. 따라서 우선순위에 맞게 queue가 정렬된다. 그 다음으로 한 노드를 탐색한 것이므로 time을 1 증가시켜준다.

```
def ids(self):
    limit = 1
    time = 0
    length = 0
    path = self.coordinate()
    while limit < self.column*self.row:
        ret = self.dfs(self.start, limit, path)
        time += ret[0]
        if ret[2] != 0:
            length += ret[2]
        if ret[1] and self.keynumber == 0:
            return (time, length)
        limit = limit + self.row
    return (0,0)
```

- ids(self)

limit을 증가시켜주면서 dfs를 돌리는 역할을 한다. 무한루프를 막기 위해서 Limit은 column과 row의 곱 즉, 전체 칸보다 작은 경우로 제한했다. dfs함수의 인자로는 start 점과 limit 그리고 path만 준다. 만약 dfs를 돌려서 나온 return 값의 1번째 요소(true or false)가 true이고 key number가 0이라면, 즉 모든 key까지의 경로를 탐색했다면 time과 length를 return하고 아니라면 limit을 row만큼 늘려준다. Row만큼 늘려주는 이유는 다음과 같다. ids가 tree를 탐색할 때 limit을 늘린다는 것은 node 하나를 늘린다는 게 아니라 한 level을 늘려주는 것이기 때문이다. 따라서 maze에서 한 level을 늘려 주는 것은 row 만큼(다음 column으로 갈수 있게) 늘려주는 것이기 때문에 row 만큼 증가시켜준다. Dfs 함수내에서는 limit을 1씩 줄여서 dfs search를 진행한다.

```
def gbfs(self):
    heuristic = lambda a, b, c: abs(a[0] - b[0]) + abs(a[1] - b[1])
    self.keylist.sort(key=lambda x: [x[1], x[0]])
    gbfs_time = 0
    gbfs_length = 0

    r = self.bfs_helper(self.start, self.keylist[0], heuristic)
    gbfs_time += r[0]
    gbfs_length += r[1]

    for i in range(0, self.keynumber - 1):
        r = self.bfs_helper(self.keylist[i], self.keylist[i + 1], heuristic)
        gbfs_time += r[0]
        gbfs_length += r[1]

    r = self.bfs_helper(self.keylist[self.keynumber - 1], self.goal, heuristic)
    gbfs_time += r[0]
    gbfs_length += r[1]
    return (gbfs_time, gbfs_length)
```

- gbfs(self)

  gbfs search는 heuristic function으로 직선거리를 사용한다. Heuristic으로 사용되는 수식에서 a는 현재 위치, b는 goal로 heuristic 값은 현재위치와 goal의 높이의 차의 제곱과 넓이의 차의 제곱을 더한 값이다. 여기서 root값을 구해주지 않은 이유는 이를 이용해서 queue에 우선순위를 할당해줄 것이므로 정확한 값보다는 크기가 중요하기 때문에 root값을 구해주지 않았다. 그 다음 key list를 sort해준다. 넓이 우선 탐색이기 때문에 넓이, 그 다음 높이로 sort해주었다. 다음으로 start-> key1까지의 경로를 bfs_helper를 통해 구하고 return 값으로 나온 time과 length를 최종적인 time과 length인 gbfs_time과 gbfs_length에 더해준다. 그 다음으로 같은 방법으로 key들 사이의 경로를 구하고 마지막 키와 goal까지의 경로를 구한다. 그 다음 time과 length를 return 해준다.

```
def a_star(self):
    heuristic = lambda a, b, c: abs(a[0] - b[0]) + abs(a[1] - b[1]) + c
    self.keylist.sort(key=lambda x: [x[1], x[0]])
    astartime = 0
    astartlength = 0

    r = self.bfs_helper(self.start, self.keylist[0], heuristic)
    astartime += r[0]
    astartlength += r[1]

    for i in range(0, self.keynumber - 1):
        r = self.bfs_helper(self.keylist[i], self.keylist[i + 1], heuristic)
        astartime += r[0]
        astartlength += r[1]

    r = self.bfs_helper(self.keylist[self.keynumber - 1], self.goal, heuristic)
    astartime += r[0]
    astartlength += r[1]
    return (astartime, astartlength)
```

- a_star(self)

  A* search는 heuristic function으로 두 점의 직선거리와 지금까지 온 거리를 더한 값을 사용한다. Heuristic으로 사용되는 수식에서 a는 현재 위치, b는 goal위치, c는 현재까지 온 length로 이 함수를 통한 heuristic 함수 값은 두 점의 거리와 현재까지 온 거리를 더한 값이다. 다음 keylist를 sort해준다. 넓이 우선 탐색이기 때문에 넓이, 그 다음 높이로 sort해준다. 다음으로 start-> key1까지의 경로를 bfs_helper를 통해 구하고 return 값으로 나온 time과 length를 최종적인 time과 length인 astartime과 astarlength에 더해준다. 그 다음으로 같은 방법으로 key들 사이의 경로를 구하고 마지막 키와 goal까지의 경로를 구한다. 그 다음 최종적인 time과 length를 return 해준다.

```
def bfs(self):
    bfstime = 0
    bfslength = 0
    while True:
        ret = self.bfs_b(self.start)
        bfstime += ret[0]
        bfslength += ret[1]
        if ret[2] and self.keynumber == 0:
            return (bfstime, bfslength)
```

- bfs(self)

  bfs는 heuristic 함수 없이, start점에서 bfs search를 해준다. 이때 사용되는 bfs search 함수는 휴리스틱을 사용하는 bfs_helper 함수가 아니라 bfs_b 함수이다. Bfs_b 함수는 return 값으로 (time, length, True or False)를 준다. 따라서 return으로 받은 0번째 인자는 time일 것이므로 time에 더해주고 1번째 인자는 length일 것이므로 length에 더해준다. 마지막 인자로는 true아니면 False인데 True는 goal상태에 갔을 때만 리턴이 된다. 그리고 모든 key를 탐색해야 하기 때문에 key number가 0이여야 올바른 탐색이라 할 수 있다. 즉, goal이고 keynumber가 0일 때 bfstime과 bfs length가 return 된다

```
def bfs_b(self, start):
    mz = copy.deepcopy(self)
    mz.map = copy.deepcopy(self.map)
    mz.maze = copy.deepcopy(self.maze)
    q = PriorityQueue()
    time = 0
    path = self.coordinate()
    q.put((0,(start, 0, start)))
    while not q.empty():
        node = q.get()
        position, length, previous = node[1]
        path[position[0]][position[1]] = previous
        mz.map[position[0]][position[1]] = 1
        if mz.maze[position[0]][position[1]] == 4:
            self.tracking(path, position)
            return (time, length, True)
        elif mz.maze[position[0]][position[1]] == 6:
            self.keynumber = self.keynumber -1
            self.tracking(path, position)
            self.start = (position[0], position[1])
            return (time, length, False)
        for dr in [(1, 0), (0, 1), (-1, 0), (0, -1)]:
            mv = mz.move(position, dr)
            if mv is not None:
                q.put((0,(mv, length + 1, position)))

        time = time + 1
    return (0, 0, False)
```

- bfs_b(self, start)

  start 위치만 받아서 탐색한다. deepcopy를 통해서 maze를 복사한다. 탐색하기 위해 Queue를 PriorityQueue()를 통해 만들어준다. Queue 안에는 (0,(노드, length, 노드))가 넣어진다. 0은 우선순위로 우선순위가 없다는 것을 의미한다. 그 다음 queue가 비기 전까지, 반복문을 돌려준다. Stack에서 pop을 통해 node를 하나 꺼내 주고 이를

position, length, previous에 할당해준다. Position은 위치, length는 경로 길이, previous는 이전 위치이다. 따라서 처음에 position에는 start, length에는 0, 이전 노드에는 start가 들어가게 된다. 처음에 maze를 배열로 받았던 maze 배열을 통해 해당 위치에 대한 숫자를 확인해준다. 만약 4라면 goal일 것이므로 tracking 함수를 통해 경로를 추적해주고 time과 length 그리고 탐색이 잘되었다는 True를 반환해준다. 다음으로 4는 아니지만 6이라면 key인 것이므로 여기까지의 경로도 tracking으로 알아준다. 6을 발견했다는 것은 key를 하나 습득했다는 것이므로 keynumber의 수는 하나 줄여준다. 그 다음 start지점을 해당 위치로 바꿔준다. 그래야 다음번에 이 위치에서부터 다시 경로를 찾아 주기 때문이다. 그 다음 time과 length, False를 반환해준다. False를 반환하는 이유는 아직 4즉, goal이 아니기 때문이다. 다음으로 4나 6이 아니라면 해당 node의 위아래 좌우 노드로 갈 수 있는지 move를 통해 확인해주고 갈 수 있다면 stack에 넣어준다. 한 노드를 탐색했으므로 time은 1증가해준다. 찾지 못했다면 time과 length는 0, 그리고 False를 반환해준다.

3. 실험 결과

(1) 최단 경로 및 탐색한 노드 개수

	Length	**Time**
Maze 1 A star	4134	10862
Maze 1 BFS	4134	13926
Maze 1 GBFS	4134	6900
Maze 1 IDS	4134	260004
Maze 2 A star	640	1105
Maze 2 BFS	640	1751
Maze 2 GBFS	640	875
Maze 2 IDS	640	7969
Maze 3 A star	574	917
Maze 3 BFS	574	1408
Maze 3 GBFS	574	770
Maze 3 IDS	574	13817
Maze 4 A star	2838	6393
Maze 4 BFS	2838	10080
Maze 4 GBFS	2838	3967
Maze 4 IDS	2838	100528