인공지능 과제 1 (IT/BT 탈출하기)

2016025987 조민지

1. 코드 설명

모든 층 공통 전처리

```
297     def fifth_floor():
298         where = "fifth_floor"
299         info = text_info(where)
300         row = info[1]
301         col = info[2]
302         out_list = text_read(where, row, col)
303         pos = []
304         pos = position_check(pos, out_list, row, col)
305         deepcopy_copy1 = copy.deepcopy(out_list)
306         deepcopy_copy2 = copy.deepcopy(out_list)
307         ans = [0, 0]
```

Line 298: where로 층을 정해줍니다.

Line 299: text_info() 함수는 ./input/*.txt의 맨 첫 줄을 읽어 사이즈(row, col)에 대한 정보를 리스트 형태로 반환합니다.

```
def text_info(where):
    f = open("./input/" + where+".txt", 'r')
    line = f.readline()
    line = line.replace("\n", "")
    result = line.split(" ")
    a=[int(result[0]),int(result[1]),int(result[2])]
    return a
```

Line 302: Text_read() 함수는 ./input/*.txt의 map 부분을 읽어와 2차 배열을 반환합니다.

```
def text_read(where,row,col):
    f = open("./input/"+where+".txt", 'r')
    line = f.readline()
    list1 = [[0 for cols in range(col + 1)] for rows in range(row + 1)]
    a = 1
    line2 = f.readline()
    while line2:
        line2 = line2.replace("₩n", "")
        result2 = line2.split(" ")
        for v in range(col):
            list1[a][v + 1] = int(result2[v])
        line2 = f.readline()
        a += 1
    f.close()
    return list1
```

Line 304: Position_check() 함수는 text_read()로 읽어온 2차 배열 값 중 입구(pos[0]), 키(pos[1]), 출구(pos[2])의 위치를 찾아 리스트형태로 반환합니다.

```
def position_check(pos, out_list, row, col):
    for r in range(1,row+1):
        for c in range(1,col+1):
            if out_list[r][c] == 3 or out_list[r][c] == 6 or out_list[r][c] == 4:
                pos.append([r,c])
    return pos
```

Line 305,306: Python의 list는 mutable 하므로 깊은 복사를 해주었습니다. (deepcopy_copy1 , deepcopy_copy2)

Line 307: ns리스트는 length 와 time의 값을 담는 리스트입니다.

2. 구현한 알고리즘 (bfs, IDS, a*, greedy)

함수의 공통 파라 미터

```
#path1=bfs(pos[0],pos[1],deepcopy_copy1,row,col,ans)
#path2=bfs(pos[1],pos[2],deepcopy_copy2,row,col,ans)

#path1 = IDS(pos[0], pos[1], deepcopy_copy1, row, col, ans)
#path2 = IDS(pos[1], pos[2], deepcopy_copy2, row, col, ans)

#path1 = astar(pos[0],pos[1],deepcopy_copy1,row,col,ans)
#path2 = astar(pos[1], pos[2], deepcopy_copy2, row, col, ans)

path1 = greedy(pos[0],pos[1],deepcopy_copy1,row,col,ans)
path2 = greedy(pos[1], pos[2], deepcopy_copy2, row, col, ans)
```

Path1 = 입구에서 키까지 (pos[0] ~ pos[1]) 깊은 복사를 한 deepcopy_copy1 맵을 보고 row, col의 정보를 바탕으로 ans의 값을 찾고 지나온 길을 '5'로 체크한 2차 배열을 반환한다.

Path2 = 키에서 출구까지 (pos[1] ~ pos[2]) 깊은 복사를 한 deepcopy_copy2 맵을 보고 row, col의 정보를 바탕으로 ans의 값을 찾고 지나온 길을 '5'로 체크한 2차 배열을 반환한다.

```
320     for i in range(1, row):
321         for j in range(1, col + 1):
322             if path1[i][j] == 5 or path2[i][j] == 5:
323                 out_list[i][j] = 5
324
325     text_write(where, out_list, ans, row, col)
```

Line 320~323: 2중 for문을 돌면서 path1에서 체크된 '5'와 path2에서 체크된 '5'를 합치는 작업을 한다.

Line 325: 최종 정답인 out_list와 ans 리스트를 text_write() 함수에 파라 미터로 넘겨 *_output.txt 파일을 작성한다.

```
def text_write(where, out_list,ans,row,col):
    f = open(where + "_output.txt", 'w')

    for i in range(1,row+1):
        for j in range(1,col+1):
            data ="%d " %out_list[i][j]
            f.write(data)
        f.write("₩n")
    f.write("---₩n")
    data2 = "length = %d₩n" %ans[0]
    f.write(data2)
    data3 = "time = %d" %ans[1]
    f.write(data3)
    f.close()
```

실행 방법

```
454     fifth_floor()
455     forth_floor()
456     third_floor()
457     second_floor()
458     first_floor()
```

3. 사용 알고리즘

구현한 알고리즘: Bfs, IDS, greedy best-first, A*

A. BFS

```
86  def bfs(start, end, out_list, row, col, ans):
87      des = [[0 for c in range(col+1)] for r in range(row+1)]
88      visit = [[0 for c in range(col+1)] for r in range(row+1)]
89      q = queue.Queue()
90      q.put(start)
91      visit[start[0]][start[1]]=1;
92      des[start[0]][start[1]]=0;
93      ans[1] +=1
94      while not q.empty():
95          if visit[end[0]][end[1]] ==1:
96              break
97          cur_task = q.get()
98          x=cur_task[0]
99          y=cur_task[1]
100         for k in range(4):
101             nx = x + dx[k]
102             ny = y + dy[k]
103             if nx >= 1 and nx <=row and ny >=1 and ny<=col and out_list[nx][ny] !=
104                 if visit[nx][ny] != 1:
105                     visit[nx][ny] =1
106                     des[nx][ny] = des[x][y] +1
107                     q.put([nx,ny])
108                     ans[1] += 1
109
110     ans[0] += des[end[0]][end[1]]
```

너비 우선 탐색(BFS)알고리즘은 queue을 사용합니다. 깊이가 1인 모든 정점을 방문하고 나서, 그 다음에는 깊이가 2인 모든 정점을, 깊이가 3인 모든 정점을 방문하는 식으로 계속 방문하다가 목표지점인 end에 도달하면 탐색을 마칩니다.

Line 87: des배열은 깊이를 저장하는 2차 배열입니다.

Line 88: 정점을 방문했는지 안 했는지를 체크하는 2차 배열입니다.

Line 89: queue을 사용합니다.

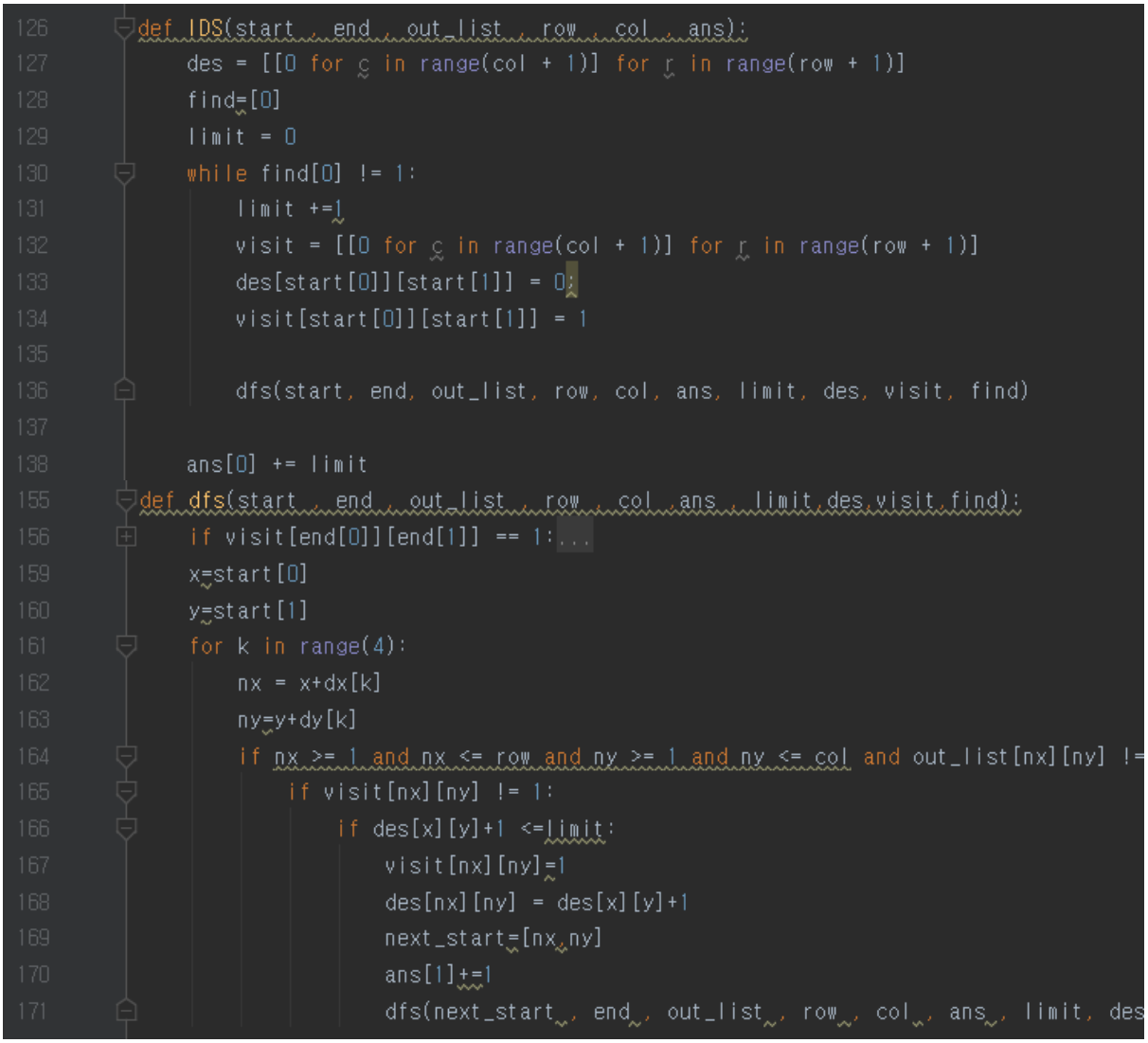
Line 90~93: start로 초기값을 정리해줍니다.

Line 94~108: queue가 empty일 때까지 BFS탐색(위, 아래, 좌, 우)을 계속합니다. 만약 도착 정점을 방문했다면 break을 합니다. Queue에 정점을 put할 때마다 ans[1]==time의 값을 +1 합니다.

Line 110: 도착 정점의 des의 값을 ans[0]==length에 더해줍니다.

선택이유: optimal을 보장해준다. 하지만 time이 너무 많이 증가한다.

B. IDS



```
126  def IDS(start, end, out_list, row, col, ans):
127      des = [[0 for c in range(col + 1)] for r in range(row + 1)]
128      find=[0]
129      limit = 0
130      while find[0] != 1:
131          limit +=1
132          visit = [[0 for c in range(col + 1)] for r in range(row + 1)]
133          des[start[0]][start[1]] = 0;
134          visit[start[0]][start[1]] = 1
135
136          dfs(start, end, out_list, row, col, ans, limit, des, visit, find)
137
138      ans[0] += limit
155  def dfs(start, end, out_list, row, col, ans, limit,des,visit,find):
156      if visit[end[0]][end[1]] == 1:...
159      x=start[0]
160      y=start[1]
161      for k in range(4):
162          nx = x+dx[k]
163          ny=y+dy[k]
164          if nx >= 1 and nx <= row and ny >= 1 and ny <= col and out_list[nx][ny] !=
165              if visit[nx][ny] != 1:
166                  if des[x][y]+1 <=limit:
167                      visit[nx][ny]=1
168                      des[nx][ny] = des[x][y]+1
169                      next_start=[nx,ny]
170                      ans[1]+=1
171                      dfs(next_start, end, out_list, row, col, ans, limit, des
```

반복적 깊이 증가는 깊이 우선 탐색 DFS처럼 메모리 필요량이 깊이 제한에 비례하면서도 최단 경로로 목표 노드를 찾는 것을 보장하는 방법이다. 반복적 깊이증가 방법에서는 목표 노드가 찾아질 때까지 깊이 제한을 1 씩 증가시키면서 연속적인 깊이 우선 탐색을 수행한다.

Line 127: des배열은 깊이를 저장하는 2차 배열입니다.

Line 128: find 리스트는 어느 limit깊이에서 end정점에 도착했는지를 체크해주는 flag 역할을 합니다.

Line 130~136: find를 할때까지 limit깊이를 +1 늘려가면서 dfs함수를 돕니다. 각 limit깊이 마다 visit을 초기화 해줘야합니다.

Line 155 ~171: 깊이 우선 탐색 DFS 코드입니다. 이 함수에 limit변수를 추가해서 des의 값과 limit을 비교해 DFS의 탐색 깊이를 제한합니다. 재귀함수형태 (Line 171)로 짜여 있어서 나아갈 길이 존재하지 않으면 이전의 위치로 돌아와 다른 길을 선택하여 움직입니다.

선택 이유: 목표 정점을 몰랐다면 이 알고리즘을 사용했을 것이다. 하지만 미로는 항

상 목표 정점(출구)의 위치를 알기 때문에 이 알고리즘이 효과적이지는 않았다. Limit을 증가할 때마다 새로 visit을 해야 하므로 time의 값은 import sys를 통해 stack의 값을 늘려줘야 할 만큼 깊게 탐색되었다. 실제로 first_floor의 input 값을 이 함수로 돌렸을 때 Exception RuntimeError: 'maximum recursion depth exceeded' 에러가 났다. 스택을 늘려준 후 (sys.setrecursionlimit(100000)) time을 보니 이해가 됐다.

C. Greedy best-first search

```
215  def greedy(start, end, out_list, row, col, ans):
216      des = [[0 for c in range(col + 1)] for r in range(row + 1)]
217      visit = [[0 for c in range(col + 1)] for r in range(row + 1)]
218      visit[start[0]][start[1]] = 1;
219      des[start[0]][start[1]] = 0;
220
221      pq2 = PriorityQueue()
222      while pq2.size() !=0:
223          pq2.pop()
224      manhattan_d = abs(start[0]-end[0])+abs(start[1]-end[1])
225      pq2.insert(manhattan_d,start[0],start[1])
226      while pq2.size() != 0:
227          if visit[end[0]][end[1]] == 1:
228              break
229          priority, x_val, y_val = pq2.pop()
230          for k in range(4):
231              nx = x_val + dx[k]
232              ny = y_val + dy[k]
233              if nx >= 1 and nx <= row and ny >= 1 and ny <= col and out_list[nx][ny]
234                  if visit[nx][ny] != 1:
235                      visit[nx][ny]=1
236                      des[nx][ny]=des[x_val][y_val]+1
237                      d=abs(nx-end[0])+abs(ny-end[1])
238                      pq2.insert(d,nx,ny)
239                      ans[1] += 1
240
241      ans[0] += des[end[0]][end[1]]
```

Line 216: des배열은 깊이를 저장하는 2차 배열입니다.

Line 217: 정점을 방문했는지 안 했는지를 체크하는 2차 배열입니다.

Line 218,219: start로 초기값을 정리해줍니다.

Line 221: heapq을 응용한 PriorityQueue()를 사용합니다.

Line 222~223: priority queue를 비워줍니다.

Line 224: greedy best-first의 f(n)은 절대값(현재위치-목표위치)입니다. Manhattan distance를 이용했습니다.

Line 225: priority queue에 insert 합니다. Manhattan distance가 우선순위입니다.

Line 226~239: priority queue가 empty일 때까지 탐색(위, 아래, 좌, 우)을 계속합니다. 만약 도착 정점을 방문했다면 break를 합니다. priority queue에 정점을 insert할 때마다 ans[1]==time의 값을 +1 합니다.

Line 241: 도착 정점의 des의 값을 ans[0]==length에 더해줍니다.

Line 10~34: class로 정의한 Priority Queue입니다. Insert는 우선순위에 따라 heappush()를 합니다. Pop()은 우선순위가 가장 작은 값을 뽑습니다. Size는 priority queue의 개수를 반환해줍니다.

*class로 정의한 Priority Queue

```
10  class PriorityQueue:
11      pq=[]
12      elements={}
13      task=0
14
15      def insert(self, priority,x_val,y_val):
16          entry = [priority, self.task,x_val,y_val]
17          self.elements[self.task]=entry
18          heapq.heappush(self.pq, entry)
19          self.task += 1
20
21      def delete(self,task):
22          entry = self.elements[task]
23          entry[-1] = None
24
25      def pop(self):
26          while self.pq:
27              priority, task, x_val, y_val = heapq.heappop(self.pq)
28              if task != None:
29                  del self.elements[task]
30                  return priority, x_val, y_val
31          raise KeyError('Pop from an empty Priority Queue')
32
33      def size(self):
34          return len(self.elements)
```

선택이유: greedy best first 탐색은 optimal이 보장되지 않는 알고리즘이라고 했다. 근데 optimal한 값이 나왔다. 이것은 이 미로의 특성이라고 판단된다.

1. 이 미로의 출구로의 길이 항상 존재합니다. 때문에 optimal을 찾을 수 있었다.
2. 맨해튼 거리를 휴레스틱 함수로 사용한 것은 굉장히 효과적이었다. 실제로 input 배열은 맨해튼 거리처럼 간격이 일정하기 때문이다.

하지만 항상 optimal한 값을 보장하지는 않으므로 time이 조금 더 많지만 A*가 더 좋은 알고리즘인 것 같다.

D. A*

A*는 greedy best-first search와 동일합니다. 다만 f(n)이 다릅니다.

Line 195: f(n) = 출발위치에서 현재위치까지의 des값 + 절대값(현재위치 –목표위치) -> Manhattan distance 입니다.

```
195                    d=abs(nx-end[0])+abs(ny-end[1])+des[nx][ny]
```

선택이유: 원래 예상대로라면 A*가 압도적으로 가장 좋을 것 같았다. 하지만 결과를 보니 적당한 예시의 greedy를 이길 수가 없었다. 맨해튼 거리처럼 일정간격이 아닌 미로가 주어진다면 A*가 더 뛰어날 것이다.

4. 최단 경로(length) / 탐색한 노드의 개수(time)

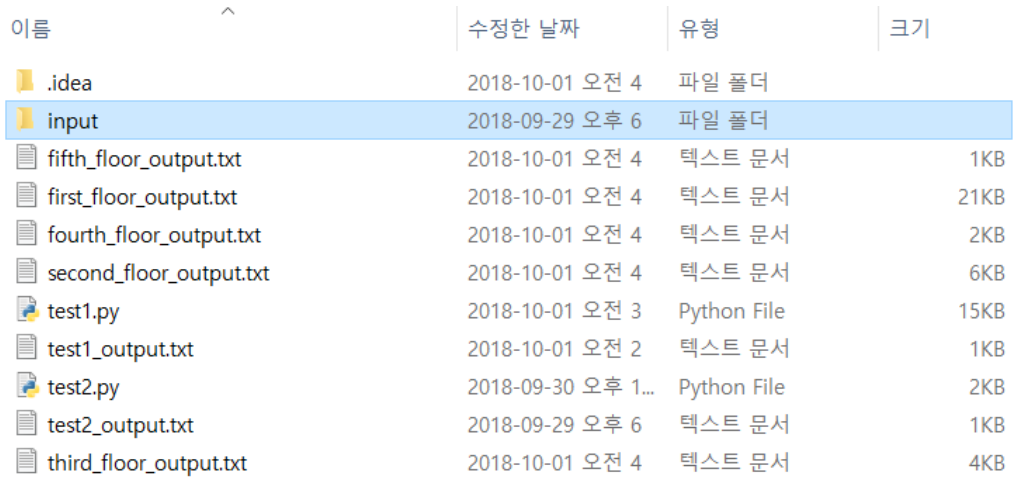
	bfs	IDS	greedy	A*
5층	106/233	106/5084	106/126	106/157
4층	334/597	334/55095	334/459	334/566
3층	554/1003	554/144035	554/676	554/833
2층	758/1722	758/308293	758/1025	758/1619
1층	3850/6748	3850/7591671	3850/5858	3850/6612

결과: greedy의 전승

심지어 greedy는 optimal을 보장하지 않는데 optimal한 값이 나왔다. 이 미로의 특성때문입니다.

5. test코드 돌리는 법

Ex) test101.txt



```
258  def test_floor():
259      where = "test1"
260      info = text_info(where)
261      row = info[1]
262      col = info[2]
263      out_list = text_read(where, row, col)
264      pos = []
265      pos = position_check(pos,out_list, row, col)
266      deepcopy_copy1 = copy.deepcopy(out_list)
267      deepcopy_copy2 = copy.deepcopy(out_list)
268      ans=[0,0]
269
```

1. Input 에 test101.txt파일을 넣습니다.
2. Line 259의 where에 해당 txt파일의 이름을 넣어줍니다.
   where = "test101"

*Input 파일은 이런 식으로 들어갑니다.

```
def text_info(where):
    f = open("./input/" + where+".txt", 'r')
```

```
270      #path1=bfs(pos[0],pos[1],deepcopy_copy1,row,col,ans)
271      #path2=bfs(pos[1],pos[2],deepcopy_copy2,row,col,ans)
272
273      #path1 = IDS(pos[0], pos[1], deepcopy_copy1, row, col, ans)
274      #path2 = IDS(pos[1], pos[2], deepcopy_copy2, row, col, ans)
275
276      #path1 = astar(pos[0],pos[1],deepcopy_copy1,row,col,ans)
277      #path2 = astar(pos[1], pos[2], deepcopy_copy2, row, col, ans)
278
279      path1 = greedy(pos[0],pos[1],deepcopy_copy1,row,col,ans)
280      path2 = greedy(pos[1], pos[2], deepcopy_copy2, row, col, ans)
```

3. bfs, IDS, astar, greedy를 준비했습니다. #주석 on/off로 선택합니다.

```
446      #test_floor()
```

4. Line 446의 주석을 풀면 실행이 됩니다.