[연도]

목차

코드 설명 – 개요 및 실행 방법

- 탐색 알고리즘은 순차적으로 BFS, IDS, GBFS, ASTAR 알고리즘을 실행합니다.
- 모든 알고리즘은 TestCase에 대해 가장 이상적인 [하, 우, 좌, 상]의 순서로 주변 노드를 탐색합니다.
- 미로는 int값의 이중리스트 형태로 저장합니다.
- 모든 알고리즘은 deepcopy로 원본 미로를 복사하여 탐색하고 경로를 저장합니다.
- 내부 라이브러리 copy, heapq를 사용하였습니다.
- 열쇠가 n개 있는 경우

  시작점 - 열쇠1 – 열쇠2 - … - 열쇠n-1 – 열쇠n – 도착점

  순으로 탐색합니다. 첫 시행에서는 열쇠1이 도착점이 되고, 이를 연쇄적으로 구현합니다. 이 때 방문한 노드의 리스트는 초기화되여, 이전 시행에서 방문한 노드를 재방문할 수도 있습니다.

- 실행 방법

1. 동일 폴더에 미로지도 텍스트파일 {name}.txt가 있다면 input으로 {name}만 입력합니다.
2. 순차적으로 BFS, IDS, GBFS, ASTAR 순으로 실행됩니다.
3. 모든 실행이 완료 후 finished 라는 메시지가 출력됩니다.

   ※ IDS의 경우 최대 깊이가 1씩 증가하여 굉장히 오랜 시간이 걸립니

다. (2배씩 증가로 수정하여 빠르게 할 수 있습니다.)

코드 설명 – 코드 구조

※ 모든 변수와 메소드는 Maze class 내에 구현되어 있습니다.

공통 변수

변수명	타입	설명
fileName	string	입력한 미로 파일의 이름
map	이차원 list int	미로를 저장할 이차원 list, 크기는 (h, w)
start	list int	시작 위치를 저장하는 list, [startx, starty]
dest	list int	도착 위치를 저장하는 list [destx, desty]
num	int	미로 번호
height	int	미로 높이
width	int	미로 폭
visited	이차원 list int	노드의 방문 여부, 0: 방문X, 1+: 방문
isfrom	이차원 list int	노드가 어디서 확장되었는지 기록 (부모 노드)
dx	list int	x 이동의 변화량 [1, 0, 0, -1]
dy	list int	y 이동의 변화량 [0, 1, -1, 0]
route	이차원 list	최적 탐색 경로
time	int	탐색한 노드의 갯수
isFound	bool	IDS에서 사용, 도착점 발견 시 True

메소드

```
def __init__(self, fileName):
    self.fileName = fileName
    f = open(fileName + ".txt", 'r')
    line = f.readline()
    self.num, self.height, self.width = map(int, line.split(' '))
    while True:
        line = f.readline().split()  # str list로 되어있음
        if not line:
            break
        line = list(map(int, list(line[0])))  # str만 추출 후 개개 chr로 분리, int 변환 mapping 후 다시 list로
        self.map.append(line)
        for i in range(0, len(line)):
            if line[i] == 3:  # starting point
                self.start = [len(self.map) - 1, i]
            if line[i] == 4:  # destination point
                self.dest = [len(self.map) - 1, i]
            if line[i] == 6:  # key point >= 0
                self.key += [[len(self.map) - 1, i], ]

    self.visited = [[0 for x in range(self.width)] for y in range(self.height)]
    self.isfrom = [[[-1, -1] for x in range(self.width)] for y in range(self.height)]
```

◆ __init__ (self, fileName)

- maze 객체 생성 시 자동으로 실행됩니다.
- fileName.txt를 한 줄씩 읽어 maze 지도, 시작점, 도착점, 키 위치를 int형태로 저장합니다.
- 저장 후 visited list와 isfrom list를 초기화 합니다.

```
def writeAnswer(self, maze, time, length, algo):
    f = open(self.fileName + "_" + algo + "_output.txt", 'w')
    for x in range(self.height):
        f.write("".join(map(str, maze[x])) + '\n')
    f.write('---\n')
    f.write('length=' + str(length) + '\n')
    f.write('time=' + str(time) + '\n')
```

◆ writeAnswer (self, maze, time, length, algo)

- maze, time, length = len(route), algo를 받아 텍스트 파일로 출력합니다.

  이 때 length는 route list의 길이 입니다.

```
def setRoute(self, maze, startPt, endPt):  # 이동한 최소 경로
    x, y = self.isfrom[endPt[0]][endPt[1]]
    while True:
        self.route += [[x, y]]
        x, y = self.isfrom[x][y]
        if [x, y] == startPt:
            if maze[x][y] == 6:  # key를 먹고서, 다시 출발하는 경우에는 key를 새지 않음 (pdf와 다름)
                self.route += [[x, y]]
                break
            else:
                break

    self.isfrom = [[[-1, -1] for x in range(self.width)] for y in range(self.height)]  # 다음 사용을 위해 초기화
```

◆ setRoute (self, maze, startPt, endPt)

- isfrom list에서 저장된 x, y를 바탕으로 backtracking합니다.
- endPt부터 시작해서, startPt에 도달할 때까지 isfrom으로 추출한 x, y에 대해서 isfrom[x][y]를 구하는 식입니다.
- [도착 – 출발]으로 구성되지만, 출발점 자체는 route에 포함하지 않습니다. 하지만 출발점이 key인 경우에는 예외적으로 포함합니다. (key 역시 경로이기 때문에)
- 모든 수행을 마친 후에는 isfrom을 초기화 해줍니다.

```
def getHx(self, pointA, pointB):  # return the Manhattan distance (heuristic)
    return abs(pointA[0] - pointB[0]) + abs(pointA[1] - pointB[1])
```

◆ getHx (self, pointA, pointB)

- Heuristic으로 사용되는 Manhattan Distance를 계산하고 반환해주는 메소드입니다.

```
def getFx(self, pointA, pointB): # return f(x) for A star ( f = g + h )
    return self.visited[pointA[0]][pointA[1]] + self.getHx(pointA, pointB)
```

◆ getFx (self, pointA, pointB)

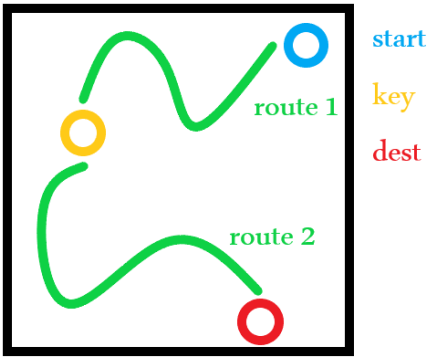
- Astar에서 사용되는 gx + hx에 해당합니다. visited를 통해서 gx를 계산하고, hx와 더하여 반환합니다.

함수설명

```
def mazeEscape(self, methodName):
    maze = copy.deepcopy(self.map)
    self.time = 0
    srcDes = [self.start] + self.key + [self.dest]

    if methodName == 'BFS':
        escapeFunc = self.bfs
    elif methodName == 'IDS':
        escapeFunc = self.ids
    elif methodName == 'GBFS':
        escapeFunc = self.gbfs
    elif methodName == 'A_star':
        escapeFunc = self.a_star

    for x in range(len(srcDes) - 1):
        escapeFunc(maze, self.time, srcDes[x], srcDes[x + 1])
        self.setRoute(maze, srcDes[x], srcDes[x + 1])
        self.visited = [[0 for x in range(self.width)] for y in
                        range(self.height)]  # s -> key1, key1 -> key2 ... 매번 초기화

    for x in range(len(self.route)):
        maze[self.route[x][0]][self.route[x][1]] = 5

    self.writeAnswer(maze, self.time, len(self.route), methodName)
    self.route = []  # route 초기화, 꼭 해주어야함
```

◆ mazeEscape (self, methodName)

- 각 methodName 별로 미로 탈출을 실행합니다.
- 시작점, 도착점, key를 srcDes list에 넣어 개개의 시작점과 도착점을 만들어 실행합니다.



start

key

dest

(다음과 같은 방식으로)

- 각 escapeFunc 이후에 각 시행에서 구한 route를 지정하고, 방문 기록 visited를 초기화합니다.
- dest까지 모두 찾아 경로를 찾은 경우 그 경로에 대해 maze값을 5로 설정해주고, 이를 출력합니다.
- 끝으로 route는 class 변수이므로 초기화 해줍니다.

```
def bfs(self, maze=0, time=0, startPt=0, endPt=0):
    queue = [startPt]
    self.visited[startPt[0]][startPt[1]] = 1

    while queue:
        x, y = queue.pop(0)

        for i in range(4):
            nx = x + self.dx[i]
            ny = y + self.dy[i]
            if 0 <= nx < self.height and 0 <= ny < self.width:
                if self.visited[nx][ny] == 0 and maze[nx][ny] != 1 and maze[nx][ny] != 3:
                    self.time += 1
                    if self.isfrom[nx][ny] == [-1, -1]:  # 처음만 변경 가능, 추후에 변경 불가
                        self.isfrom[nx][ny] = [x, y]
                    self.visited[nx][ny] = 1
                    if [nx, ny] == endPt:
                        return
                    queue.append([nx, ny])
```

◆ bfs (self, maze, time, startPt, endPt)

- queue를 이용합니다.
- nx, ny로 다음 탐색할 노드를 찾습니다. 단 방문하지 않았으며, 지도를 벗어나지 않으며, 벽이 아니고 start 노드가 아니어야 합니다.
- 탐색을 하면 time을 증가시켜 탐색한 노드의 수를 증가시킵니다.
- 탐색을 하며 부모 노드를 기록합니다. (isfrom, 어디서 온 노드인지 확인)
- endPt를 찾으면 함수를 탈출합니다.
- 그렇지 않다면, 주변 노드 [nx, ny]를 queue에 넣고 반복합니다.

```python
def ids(self, maze, time, startPt, endPt):
    maxDepth = 1
    depth = 0
    self.isFound = False
    while True:
        self.visited = [[0 for x in range(self.width)] for y in range(self.height)]
        self.isfrom = [[[-1, -1] for x in range(self.width)] for y in range(self.height)]
        self.idsEx(maze, time, startPt, endPt, maxDepth)
        if self.isFound:
            break
        maxDepth += 1  # +1 is too slow, for fast search please change it to " *= 2 "
```

◆ ids (self, maze, time, startPt, endPt)

- 탐색하는 깊이를 늘려가는 DFS입니다.
- 방문 여부와 어디서 왔는지 (visited, isfrom) 초기화하고 정해진 maxDepth에 대해서 DFS(idsEX)를 수행합니다.
- isFound가 False인 경우, maxDepth를 증가시켜 다시 수행합니다. (테스트에서는 maxDepth를 하나씩 증가시켰지만, 이 경우에 너무 많은 노드를 탐색하게 됩니다. 2배씩 증가시키는 것을 권장합니다.)

```python
def idsEx(self, maze, time, startPt, endPt, maxDepth):
    stack = [startPt]
    self.visited[startPt[0]][startPt[1]] = 1

    while stack:
        x, y = stack.pop()  # top의 원소 제거

        for i in range(4):
            nx = x + self.dx[i]
            ny = y + self.dy[i]
            if 0 <= nx < self.height and 0 <= ny < self.width:
                if self.visited[nx][ny] == 0 and maze[nx][ny] != 1 and maze[nx][ny] != 3:
                    self.time += 1
                    if self.isfrom[nx][ny] == [-1, -1]:  # 처음만 변경 가능, 추후에 변경 불가
                        self.isfrom[nx][ny] = [x, y]
                    self.visited[nx][ny] = self.visited[x][y] + 1
                    if [nx, ny] == endPt:
                        self.isFound = True
                        return
                    if self.visited[nx][ny] < maxDepth:
                        stack.append([nx, ny])
```

◆ idsEx (self, maze, time, endPt, maxDepth)

- 실질적으로 DFS를 수행합니다.
- stack을 사용합니다.
- nx, ny로 인접 노드를 계속해서 탐색합니다.
- endPt를 찾은 경우 isFound를 True로 변경해주고 return합니다.
- 인접 노드의 visited는 부모 노드의 visited보다 1씩 증가시킵니다. 그렇기 때문에 시작 노드로부터의 거리 즉 깊이에 해당합니다.
- visited가 maxDepth보다 작은 경우에만, 즉 현재 노드의 깊이가 최대 깊이보다 작은 경우에만 stack에 집어넣습니다. 이렇게 DFS의 깊이를 제한하여 탐색합니다.

```python
def gbfs(self, maze, time, startPt, endPt):
    queue = [(self.getHx(startPt, endPt), startPt)]
    self.visited[startPt[0]][startPt[1]] = 1

    while queue:
        x, y = (heapq.heappop(queue))[1]  # (hx, [x, y]) 형태

        for i in range(4):
            nx = x + self.dx[i]
            ny = y + self.dy[i]
            if 0 <= nx < self.height and 0 <= ny < self.width:
                if self.visited[nx][ny] == 0 and maze[nx][ny] != 1 and maze[nx][ny] != 3:
                    self.time += 1
                    if self.isfrom[nx][ny] == [-1, -1]:  # 처음만 변경 가능, 추후에 변경 불가
                        self.isfrom[nx][ny] = [x, y]
                    self.visited[nx][ny] = 1
                    if [nx, ny] == endPt:
                        return
                    heapq.heappush(queue, (self.getHx([nx, ny], endPt), [nx, ny]))
```

◆ gbfs (self, maze, time, endPt, maxDepth)

- Greedy Best First Search입니다.
- priority queue를 사용합니다. queue에는 특정점과 도착점의 Manhattan distance를 잰 heuristic H(x)와 특정점의 위치가 저장되어 있습니다.

  [H(startPt, endPt), starPt]
- minheap을 사용하여, heuristic이 제일 작은 값을 선택합니다.
- 작동 방식은 BFS와 동일합니다. 단 queue 대신 heap이기에 heuristic이 제일 작은 노드를 선택하여 탐색합니다.

```
def a_star(self, maze, time, startPt, endPt):
    queue = [(self.getFx(startPt, endPt), startPt)]
    self.visited[startPt[0]][startPt[1]] = 1

    while queue:
        x, y = (heapq.heappop(queue))[1]  # (fx, [x, y]) 형태

        for i in range(4):
            nx = x + self.dx[i]
            ny = y + self.dy[i]
            if 0 <= nx < self.height and 0 <= ny < self.width:
                if self.visited[nx][ny] == 0 and maze[nx][ny] != 1 and maze[nx][ny] != 3:
                    self.time += 1
                    if self.isfrom[nx][ny] == [-1, -1]:  # 처음만 변경 가능, 추후에 변경 불가
                        self.isfrom[nx][ny] = [x, y]
                    self.visited[nx][ny] = self.visited[x][y] + 1
                    if [nx, ny] == endPt:
                        return
                    heapq.heappush(queue, (self.getFx([nx, ny], endPt), [nx, ny]))
```

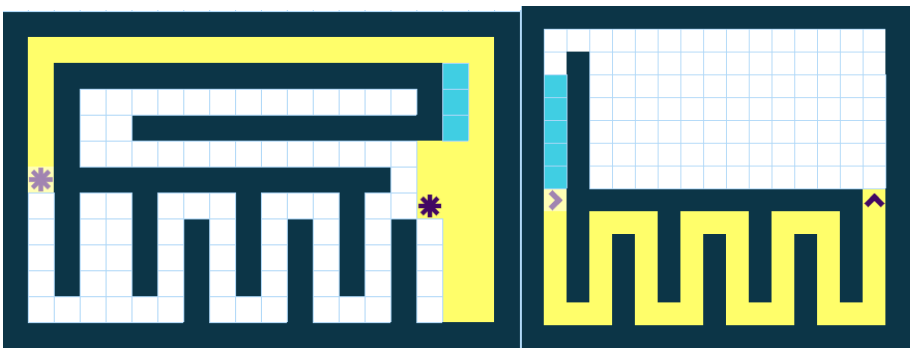
◆ a_star (self, maze, time, endPt, maxDepth)

- Astar algorithm을 사용합니다.
- gbfs와 동일하게 priority queue를 사용하며, minheap으로 구성되어 있습니다.
- F(x)를 heap의 값으로 사용합니다. F(x) = H(x) + G(x)로 H(x)는 앞서 gbfs에서 사용한 Mahattan Distance heuristic입니다. G(x)는 출발점에서 특정점까지의 거리를 의미합니다. 이는 visited list를 통해서 구현하였습니다.
- 앞서 서술한대로 BFS기반입니다. 단 F(x)를 비교로 minheap으로 작동합니다.

실험결과 – 결과 비교

	Maze_1		Maze_2	
algorithm	**length**	**time**	**length**	**time**
BFS	4133	11237	1371	2509
IDS	4133	10729641	1371	554615
GBFS	4133	6685	1371	2039
A_STAR	4133	10863	1371	2425
	Maze_3		**Maze_4**	
BFS	573	1026	5497	10406
IDS	573	96629	5497	10471569
GBFS	573	771	5497	8481
A_STAR	573	918	5497	10309

- 최단 거리는 모든 알고리즘이 동일하게 나왔습니다. 하지만 모든 알고리즘이 최단 거리를 보장하지는 않습니다.



다음과 같은 미로가 있다면, 왼쪽은 IDS와 GBFS가 최단 거리를 보장하지 않고, 오른쪽 같은 경우 GBFS가 최단 거리를 보장하지 않습니다. 실제로는 BFS, ASTAR만이 최단거리를 보장합니다.

- 탐색하는 노드는 GBFS – ASTAR – BFS – IDS 순으로 많았습니다. 하지만 앞서 말한 특이 케이스를 생각한다면 통상적으로 ASTAR가 robust한 알고리즘이라고 생각할 수 있습니다.

- IDS는 최대 깊이의 설정에 따라서 탐색 노드 수가 변할 수 있습니다. 위 값은 깊이를 1씩만 늘려서, 굉장히 큰 값을 얻게 되었습니다.

실험결과 – 개선점

- 시작점과 도착점의 위치를 확인하고, 이에 따라서 dx, dy값의 순서, 즉 어디를 먼저 탐색할지를 변경하면 속도의 개선을 얻을 수 있을 것으로 추정합니다.
- 상하좌우 중 먼저 탐색할 노드를 시작점과 도착점의 위치를 바탕으로 heuristic하게 처리한다면, 속도가 더 빨라질 수 있을 것입니다.
- IDS에서는 이미 탐색한 노드라고 해도 깊이를 늘려 재탐색하게 되는데, 이전 깊이에서 탐색한 마지막 노드들을 따로 저장해 거기부터 다시 탐색한다면 속도를 높일 수 있을 것으로 추정합니다. 하지만 재탐색의 비용이 별로 크지 않다는 경험적 결과(수업 내용)에 따르면 큰 향상이 없을 수도 있습니다.
- 앞서 IDS에서 설명했듯이, 최대 깊이를 1씩 늘리는 것은 굉장히 느리고, 무의미합니다. 그렇기에 최대 깊이의 초기값, 그리고 그 증가량을 유의미하게 설정한다면, IDS에서 시간을 많이 줄일 수 있을 것으로 추정합니다.