# Assignment 1

미로 탈출하기

2020-2 인공지능
최용석교수님

컴퓨터소프트웨어학부
2015004911 임성준

## ● 코드 및 함수 설명

### 0. 과제 이해 및 기본 동작 설명

1) 미로를 입구에서 출발하여 출구까지 도달한다.
2) 도중에 있는 key를 모두 습득해야 한다.
3) 출발 지점에서 시작해 key를 찾으러 간다.
4) Key를 찾으면 해당 지점에서 다른 key가 있는 경우 그 지점으로 간다.
5) Key를 모두 찾으면 출구를 탐색해 간다.
6) 탐색이 끝나면 지나온 길을 backtraking하며 표시한다.

* 구현하지 못한 부분 : 4번의 조건을 만족시키고 싶었지만 제출된 코드 상에서는 구현하지 못했다. 구현하여 제출한 코드는 Key를 한 번만 찾도록 동작한다.

### 1. *bfs*

```
#bfs
def bfs(x, y, goal):
    global arr, bfs_arr, dx, dy
    bfs_backtrack = [[0 for x in range(int(n))] for y in range(int(m))]
    for i in range(int(m)):
        for j in range(int(n)):
            bfs_backtrack[i][j] = -1

    queue = []
    queue.append((x,y))
    time = 0
    bfs_backtrack[x][y] = 4

    while len(queue) > 0:
        x = queue[0][0]
        y = queue[0][1]
        if arr[x][y] == goal:
            break
        del queue[0]
        time += 1

        for i in range(4):
            nx = x + dx[i]
            ny = y + dy[i]
            if nx < 0 or ny < 0 or nx >= int(m) or ny >= int(n):
                continue
            if (arr[nx][ny] == 2 or arr[nx][ny] == goal) and bfs_backtrack[nx][ny] == -1:
                queue.append((nx,ny))
                bfs_backtrack[nx][ny] = i

    rx = x
    ry = y
    length = 0
    while bfs_backtrack[rx][ry] < 4:
        length += 1
        bfs_arr[rx][ry] = 5
        for i in range(4):
            if bfs_backtrack[rx][ry] == i:
                rx = rx - dx[i]
                ry = ry - dy[i]
                break

    if goal == 4:
        bfs_arr[x][y] = goal
    return (x, y, time, length)
```

BFS(Breadth-First Search)는 넓이우선탐색 알고리즘이다.

1) Queue에 시작 좌표를 넣고, 해당 좌표에서 갈 수 있는 모든 방향의 좌표들을 queue에 넣는다.
2) FIFO형식에 따라 queue의 맨 앞에 있는 좌표를 꺼내 동일한 방법으로 좌표들을 넣는다.
3) Goal에 도착하기 전까지 2번을 반복한다.
4) Goal 좌표에 도착하면 함수를 종료한다.
5) 결과물로 출력될 파일에는 최단 경로의 길을 표시해야 하므로 역추적하는 방법을 사용했다. bfs_backtrack이라는 2중 list를 이용해 방문한 경로(방향)를 기억하며 goal을 찾고, goal에 도착하면 갔던 길을 되돌아오며 경로를 저장한다. (이 과정은 밑에 나올 모든 알고리즘의 마지막에 동일하게 동작한다.)

### 2. *ids*

```
#ids
def ids(x, y, goal):
    global arr, ids_arr, dx, dy
    ids_backtrack = [[0 for x in range(int(n))] for y in range(int(m))]
    time = 0
    limit = 0

    while True:
        is_goal = 0
        stack = []
        stack.append((x,y,0))
        for i in range(int(m)):
            for j in range(int(n)):
                ids_backtrack[i][j] = -1
        ids_backtrack[x][y] = 4

        while len(stack) > 0:
            last = len(stack) - 1
            ix = stack[last][0]
            iy = stack[last][1]
            depth = stack[last][2]

            if arr[ix][iy] == goal:
                is_goal = 1
                break
            del stack[last]
            if depth == limit:
                continue
            time += 1

            for i in range(4):
                nx = ix + dx[i]
                ny = iy + dy[i]
                if nx < 0 or ny < 0 or nx >= int(m) or ny >= int(n):
                    continue
                if (arr[nx][ny] == 2 or arr[nx][ny] == goal) and ids_backtrack[nx][ny] == -1:
                    stack.append((nx,ny,depth+1))
                    ids_backtrack[nx][ny] = i

        if is_goal == 1:
            break
        limit += 1

    rx = ix
    ry = iy
    while ids_backtrack[rx][ry] < 4:
        ids_arr[rx][ry] = 5
        for i in range(4):
            if ids_backtrack[rx][ry] == i:
                rx = rx - dx[i]
                ry = ry - dy[i]
                break
    if goal == 4:
        ids_arr[ix][iy] = goal
    return (ix, iy, time, depth)
```

IDS(Iterative Deepening Search)란 DFS를 수행하는데, limit을 정해놓고 그 limit을 만족시키며 수행하는 알고리즘이다.

1) Stack에 시작 좌표를 넣고, 해당 좌표에서 갈 수 있는 모든 방향의 좌표들을 stack에 넣는다.
2) 탐색할 depth의 limit을 설정한다.
3) LIFO형식에 따라 stack의 맨 뒤에 있는 좌표를 꺼내 동일한 방법으로 좌표들을 넣는다.
4) Goal에 도착하기 전까지, 혹은 limit을 초과할 때까지 2번을 반복한다.
5) 만약 limit을 초과했다면 limit을 증가시키고 위의 과정을 반복한다.
6) Goal 좌표에 도착하면 함수를 종료한다.

### *3. gbfs*

```
#gbfs
def gbfs(x, y, goal):
    global arr, gbfs_arr, dx, dy
    gbfs_backtrack = [[0 for x in range(int(n))] for y in range(int(m))]
    for i in range(int(m)):
        for j in range(int(n)):
            gbfs_backtrack[i][j] = -1

    heap = Heap()
    g_xy = goal_xy(goal)
    h = distance(x,y,g_xy[0],g_xy[1])
    heap.insert(h,x,y,0)
    time = 0
    gbfs_backtrack[x][y] = 4

    while heap.size > 0:
        x = heap.h[0][1]
        y = heap.h[0][2]
        depth = heap.h[0][3]
        if arr[x][y] == goal:
            break
        time += 1
        heap.delete()

        for i in range(4):
            nx = x + dx[i]
            ny = y + dy[i]
            if nx < 0 or ny < 0 or nx >= int(m) or ny >= int(n):
                continue
            if (arr[nx][ny] == 2 or arr[nx][ny] == goal) and gbfs_backtrack[nx][ny] == -1:
                heap.insert(distance(nx,ny,g_xy[0],g_xy[1]),nx,ny,depth+1)
                gbfs_backtrack[nx][ny] = i

    rx = x
    ry = y
    while gbfs_backtrack[rx][ry] < 4:
        gbfs_arr[rx][ry] = 5
        for i in range(4):
            if gbfs_backtrack[rx][ry] == i:
                rx = rx - dx[i]
                ry = ry - dy[i]
                break

    if goal == 4:
        gbfs_arr[x][y] = goal
    return (x, y, time, depth)
```

GBFS(Greedy Best-First Search)란 우선도를 정해놓고 탐색을 수행하는 알고리즘이다. 이 알고리즘에서의 우선도는 단순 거리(Manhattan distance)이다.

1) Heap에 시작 좌표와 해당 좌표에서 목적 좌표까지의 Manhattan distance(이하 거리)를 넣는다.
2) 해당 좌표에서 갈 수 있는 모든 방향의 좌표들과 거리를 heap에 넣는다.
3) 가장 거리가 짧은(맨 위의 노드 – 아래 Heap 구현에서 설명함 - )노드를 지워주며 해당 노드에서 2번을 반복한다.
4) Goal 좌표에 도착하면 함수를 종료한다.

### 4. a_star

```
#a_star
def a_star(x, y, goal):
    global arr, a_star_arr, dx, dy
    a_star_backtrack = [[0 for x in range(int(n))] for y in range(int(m))]
    for i in range(int(m)):
        for j in range(int(n)):
            a_star_backtrack[i][j] = -1

    heap = Heap()
    g_xy = goal_xy(goal)
    h = distance(x,y,g_xy[0],g_xy[1])
    heap.insert(0+h,x,y,0)
    time = 0
    a_star_backtrack[x][y] = 4

    while heap.size > 0:
        x = heap.h[0][1]
        y = heap.h[0][2]
        depth = heap.h[0][3]
        if arr[x][y] == goal:
            break
        time += 1
        heap.delete()

        for i in range(4):
            nx = x + dx[i]
            ny = y + dy[i]
            if nx < 0 or ny < 0 or nx >= int(m) or ny >= int(n):
                continue
            if (arr[nx][ny] == 2 or arr[nx][ny] == goal) and a_star_backtrack[nx][ny] == -1:
                heap.insert(distance(nx,ny,g_xy[0],g_xy[1])+depth,nx,ny,depth+1)
                a_star_backtrack[nx][ny] = i

    rx = x
    ry = y
    while a_star_backtrack[rx][ry] < 4:
        a_star_arr[rx][ry] = 5
        for i in range(4):
            if a_star_backtrack[rx][ry] == i:
                rx = rx - dx[i]
                ry = ry - dy[i]
                break

    if goal == 4:
        a_star_arr[x][y] = goal
    return (x,y,time,depth)
```

A_star(A*)란 우선도를 정해놓고 탐색을 수행하는 알고리즘이다. 이 알고리즘에서의 우선도는 단순 거리(Manhattan distance) + 현재까지 온 거리이다.

1) Heap에 시작 좌표와 해당 좌표에서 목적 좌표까지의 Manhattan distance + 해당 좌표까지 온 거리(이하 우선도)를 넣는다.
2) 해당 좌표에서 갈 수 있는 모든 방향의 좌표와 우선도를 heap에 넣는다.
3) 가장 우선도가 낮은(거리가 짧은)노드를 지워주며 해당 노드에서 2번을 반복한다.
4) Goal 좌표에 도착하면 함수를 종료한다.

### 5. Heap

```
#Heap
class Heap:
    def __init__(self):
        self.h = []
        self.size = 0

    def build(self, node):
        m = node
        if 2*node+1 < self.size and self.h[2*node+1][0] < self.h[m][0]:
            m = 2*node+1
        if 2*node+2 < self.size and self.h[2*node+2][0] < self.h[m][0]:
            m = 2*node+2
        if m != node:
            self.h[node], self.h[m] = self.h[m], self.h[node]
            self.build(m)

    def insert(self, hu, x, y, depth):
        self.h.append((hu,x,y,depth))
        cursize = self.size
        self.size += 1
        while (cursize-1)//2 >= 0 and self.h[cursize][0] < self.h[(cursize-1)//2][0]:
            temp = self.h[(cursize-1)//2]
            self.h[(cursize-1)//2] = self.h[cursize]
            self.h[cursize] = temp
            cursize = (cursize-1)//2

    def delete(self):
        cursize = self.size
        self.h[0] = self.h[cursize-1]
        self.size -= 1
        del self.h[cursize-1]
        self.build(0)
```

Heap은 위에서 서술한 GBFS와 A*알고리즘에 사용하기 위해 만든 class이다.

1) __init__ : Heap을 처음 선언할 시 호출되는 함수이다. 변수들이 선언된다.
2) build : Heap을 minheap으로 정렬해주는 함수다. 이 자료구조에서 minheap을 사용하는 이유는, 거리(우선도)가 짧을수록(작을수록) 최단 경로에 가깝기 때문에 minheap을 사용했다.
3) insert : heap에 필요한 인자들을 받아, heap에 넣는다. Insert 기능 시, 넣은 다음 정렬하는 것이 아닌, 처음부터 정렬을 하며 insert한다.
4) delete : 맨 위의 node를 제거한다. 2번의 build 함수는 여기서 사용된다. Node가 제거될 경우, heap의 구조를 유지하지 못할 경우가 있기 때문에 build를 한다.

### 6. distance

```
#distance
def distance(nx, ny, gx, gy):
    return abs(nx-gx) + abs(ny-gy)
```

(nx, ny)와 (gx, gy) 사이의 Manhattan distance를 구하는 함수다. GBFS와 A*에서 우선도를 구할 때 사용한다.

### 7. goal_xy

```
#goal_xy
def goal_xy(goal):
    for i in range(int(m)):
        for j in range(int(n)):
            if arr[i][j] == goal:
                return (i,j)
```

목적지의 좌표 값을 찾아주는 함수다. GBFS와 A*에서 distance 함수를 호출하게 되는데, 이때 거리를 구하기 위한 좌표 값을 찾을 때 사용한다.

### 8. maze_print

```
#maze_print
def maze_print(k, time, length, algorithm, arr):
    file = open("Maze_%d_" %int(k) + algorithm +"_output.txt", "w")

    for i in range(int(m)):
        for j in range(int(n)):
            file.write(str(arr[i][j]))
        file.write('\n')

    file.write('---\n' )
    file.write('length=%d\n' %length)
    file.write('time=%d\n' %time)
    file.close()
```

결과값 출력 시, file을 만들어 출력해야 한다.

1) 미로 번호(k), 탐색 횟수(time), 최소 길이(length), 알고리즘(algorithm), 출력할 배열(arr)을 인자로 받는다.
2) Open 함수로 원하는 파일명을 입력하고 생성한다.
3) 출력할 배열과 변수들을 입력한다.
4) 생성한 파일을 닫는다.

### 9. Main

```
#main
maze_number = int(input("Input Maze's Number : "))
file = open("Maze_%d.txt" %maze_number, "r")
k,m,n = file.readline().split()
maze_input = file.readlines()
file.close()
print("program start")
print("loading...")

global arr, bfs_arr, ids_arr, gbfs_arr, a_star_arr, dx, dy
arr = [[0 for x in range(int(n))] for y in range(int(m))]
bfs_arr = [[0 for x in range(int(n))] for y in range(int(m))]
ids_arr = [[0 for x in range(int(n))] for y in range(int(m))]
gbfs_arr = [[0 for x in range(int(n))] for y in range(int(m))]
a_star_arr = [[0 for x in range(int(n))] for y in range(int(m))]
dx = [1,-1,0,0]
dy = [0,0,1,-1]

i = 0
for x in maze_input:
    j = 0
    for y in x:
        if y != '\n':
            arr[i][j] = int(y)
        j += 1
    i += 1
```

```
for i in range(int(m)):
    for j in range(int(n)):
        bfs_arr[i][j] = arr[i][j]
        ids_arr[i][j] = arr[i][j]
        gbfs_arr[i][j] = arr[i][j]
        a_star_arr[i][j] = arr[i][j]

for i in range(int(m)):
    for j in range(int(n)):
        if arr[i][j] == 3:
            #bfs
            print("bfs start")
            bfs_key = bfs(i,j,6)
            bfs_time = bfs_key[2]
            bfs_length = bfs_key[3]

            bfs_goal = bfs(bfs_key[0], bfs_key[1], 4)
            bfs_time += bfs_goal[2]
            bfs_length += bfs_goal[3]
            print("bfs end")

            #ids
            print("ids start")
            ids_key = ids(i, j, 6)
            ids_time = ids_key[2]
            ids_length = ids_key[3]

            ids_goal = ids(ids_key[0], ids_key[1], 4)
            ids_time += ids_goal[2]
            ids_length += ids_goal[3]
            print("ids end")

            #gbfs
            print("gbfs start")
            gbfs_key = gbfs(i, j, 6)
            gbfs_time = gbfs_key[2]
            gbfs_length = gbfs_key[3]

            gbfs_goal = gbfs(gbfs_key[0], gbfs_key[1], 4)
            gbfs_time += gbfs_goal[2]
            gbfs_length += gbfs_goal[3]
            print("gbfs end")

            #a_star
            print("a_star start")
            a_star_key = a_star(i, j, 6)
            a_star_time = a_star_key[2]
            a_star_length = a_star_key[3]

            a_star_goal = a_star(a_star_key[0], a_star_key[1], 4)
            a_star_time += a_star_goal[2]
            a_star_length += a_star_goal[3]
            print("a_star end")
```

```
            print("program end")
            break

    maze_print(k, bfs_time, bfs_length, "BFS", bfs_arr)
    maze_print(k ,ids_time, ids_length, "IDS", ids_arr)
    maze_print(k, gbfs_time, gbfs_length, "GBFS", gbfs_arr)
    maze_print(k, a_star_time, a_star_length, "A_star", a_star_arr)
```

1) 연산을 수행할 미로의 번호를 먼저 입력한다.
2) 해당 미로의 파일을 열고, 각 변수에 저장한다.
3) 미로 배열과 각 알고리즘 별로 저장할 배열들을 선언하고, 초기화한다.
4) 출발 좌표를 찾고, 연산을 수행한다.

* 연산을 수행하는 부분에서 key의 위치를 한번만 찾고 수행하도록 구현하는 실수를 했다. 이 부분을 늦게 발견해서 후에 수정하려 했으나 마감 기한때문에 미처 구현하지 못했다.

5) 연산이 완료되면 정답 출력을 위한 함수를 호출함으로 프로그램을 종료한다.

## ● 실험 결과

| | **BFS** | **IDS** | **GBFS** | **A*** |
|---|---|---|---|---|
| **Maze_1** | length : 4134<br>time : 11237 | length : 4134<br>time : 10722495 | length : 4134<br>time : 6623 | length : 4134<br>time : 10848 |
| **Maze_2** | length : 520<br>time : 945 | length : 520<br>time : 184989 | length : 520<br>time : 691 | length : 520<br>time : 894 |
| **Maze_3** | length : 570<br>time : 987 | length : 570<br>time : 127839 | length : 570<br>time : 638 | length : 570<br>time : 895 |
| **Maze_4** | length : 2382<br>time : 5470 | length : 2382<br>time : 3622093 | length : 2382<br>time : 4144 | length : 2382<br>time : 5447 |

## ● 결론

우선 결과적으로 최단 경로는 같아야 하기 때문에 length는 동일하게 나온 것을 볼 수 있다. 하지만 time의 경우 각 알고리즘마다 모두 다르며, 특히 IDS의 경우에는 거의 제곱수 가까이 나오는 것을 볼 수 있다.