13주차 예비보고서

전공: 수학/컴퓨터공학 학년: 3학년 학번: 20181294 이름: 임승섭

**1.**

DFS와 BFS는 adjacency matrix와 adjacency list 중 어떤 자료구조를 이용하는지에 따라 구현 방법이 달라진다. 이에 따라 시간복잡도도 달라지게 된다.

우선 DFS를 adjacency matrix로 구현했을 때, 먼저 N번의 loop을 돈다. 각 loop에서는 또다시 N개의 정점을 방문하여 이미 방문했는지 여부를 확인한다. 즉, 시간복잡도는 O(N^2)이다. adjacency list로 구현했을 때, 각 노드에 몇 개의 노드가 연결되어있는지 모르기 때문에 세밀한 시간은 예측하기 어렵지만, 결국 모든 vertex와 모든 edge를 방문하는 것을 알 수 있다. 따라서 시간복잡도는 O(N + E)이다.

BFS도 결론적으로 DFS와 시간복잡도가 같다. BFS를 adjacency matrix로 구현했을 때, 마찬가지로 N번의 loop을 돈다. 이 loop은 BFS에서 사용하는 큐에 아무것도 남지 않을 때까지 돌게 된다. 즉, 모든 vertex를 방문했을 때까지 loop이 돈다. 따라서 시간복잡도는 O(N^2)이다. adjacency list로 구현했을 때는 DFS와 마찬가지로 생각할 수 있다. 코드를 분석하기보다는 결과적으로 모든 vertex와 모든 edge를 방문하므로 시간복잡도는 O(N + E)라고 할 수 있다.

**2.**

2 - 1. DFS

우선 2주차에 미로의 정보를 저장한 2차원 배열 input[2\*HEIGHT + 1][2 \* WIDTH + 1]을 이용한다. 배열의 여기서 HEIGHT, WDITH는 실제 미로의 가로, 세로를 의미하고 배열 내에는 방 사이의 벽과 맨 끝 테두리 벽을 포함하기 때문에 위와 같이 계산하였다. 이 배열과 같은 크기의 2차원 배열 visited[2\*HEIGHT + 1][2 \* WIDTH + 1]을 생성한다. 빈 방을 방문하였는지 확인하기 위한 용도이다. 그런데 벽은 방문할 수가 없기 때문에 이미 방문했다는 뜻으로 1을 할당해주면 된다. 따라서 초기 설정은 input 배열 값이 ‘ ‘인 경우에만 visited 배열 값을 0을 주고 나머지는 모두 1을 준다. 그리고 우리는 (1, 1) 위치에서 시작하기 때문에 visited[1][1] 값도 1을 준다.

DFS 구현을 위한 stack을 사용해야 한다. 전체 경로를 저장할 stack과 최적 경로의 정보만 저장할 stack을 따로 만들기로 하였다. 최적 경로를 저장할 stack을 best\_stack\_x, best\_stack\_y, 모든 경로를 저장할 stack을 all\_stack\_x, all\_stack\_y라고 하겠다.

아래는 위의 내용을 제외한 pseudo code를 만들어 보았다.

best\_stack\_x.push(1);

best\_stack\_y.push(1);

all\_stack\_x.push(1);

all\_stack\_y.push(1);

while(!best\_stack\_x.empty() && !best\_stack\_y.empty() ) {

// 현재 최적 경로의 top을 저장하고, 상하좌우 4방향으로 갈 수 있는지 확인한다.

int x = best\_stack\_x.top();

int y = best\_stack\_y.top();

// 만약 한 방향이라도 갈 수 있는 곳이 있다면, stack에 추가하고 visited값을 1로 바꾼다.

// 해당 좌표로 바로 이동할 수 있게끔 continue를 넣어주어서 다른 if문에 걸리지 않도록 한다.

if (! visited[y+1][x])

best\_stack\_x.push(x);

best\_stack\_y.push(y+1);

all\_stack\_x.push(x);

all\_stack\_y.push(y+1);

visited[y+1][x] = 1;

continue;

if (! visited[y-1][x])

best\_stack\_x.push(x);

best\_stack\_y.push(y-1);

all\_stack\_x.push(x);

all\_stack\_y.push(y-1);

visited[y-1][x] = 1;

continue;

if (! visited[y][x+1])

best\_stack\_x.push(x+1);

best\_stack\_y.push(y);

all\_stack\_x.push(x+1);

all\_stack\_y.push(y);

visited[y][x+1] = 1;

continue;

if (! visited[y][x-1])

best\_stack\_x.push(x-1);

best\_stack\_y.push(y);

all\_stack\_x.push(x-1);

all\_stack\_y.push(y);

visited[y][x-1] = 1;

continue;

// 4 방향 모두 갈 수 있는 곳이 없다면 현재 위치를 best\_stack에서는 제거하고, all\_stack에는 제거한 이후의 좌표를 넣어주어 도착했던 모든 경로를 저장시킨다

else

best\_stack\_x.pop();

best\_stack\_y.pop();

all\_stack\_x.push(best\_stack\_x.top());

all\_stack\_y.push(best\_stack\_y.top());

2 - 2.

BFS 구현 시 사용하는 visited 배열의 정보는 DFS와 동일하므로 이 부분은 생략하겠다.

BFS는 DFS와 다르게 queue 자료구조를 사용한다.

best\_queue\_x.push(1);

best\_queue\_y.push(1);

all\_queue\_x.push(1);

all\_queue\_y.push(1);

while( !best\_queue\_x.empty() && !best\_queu\_y.empty() ) {

// 현재 위치를 저장하고, 상하좌우 4방향 중 갈 수 있는 위치를 queue에 저장한다.

int x = best\_queue\_x.front();

int y = best\_queue\_y.front();

// 4방향 중 갈 수 있는 위치를 queue에 저장하기 전에, 현재 위치를 queue에서 빼준다.

best\_queue\_x.pop();

best\_queue\_y.pop();

// 4가지 방향에 접근 가능한지 확인한다.

if (! visited[y+1][x])

best\_queue\_x.push(x)

best\_queue\_y.push(y+1)

visited[y+1][x] = 1

if (! visited[y-1][x])

best\_queue\_x.push(x)

best\_queue\_y.push(y-1)

visited[y-1][x] = 1

if (! visited[y][x+1])

best\_queue\_x.push(x+1)

best\_queue\_y.push(y)

visited[y][x+1] = 1

if (! visited[y][x-1])

best\_queue\_x.push(x-1)

best\_queue\_y.push(y)

visited[y][x-1] = 1