13주차 예비보고서

전공: 수학과,컴퓨터공학과 학년: 3학년 학번: 20181256 이름: 김도현

1-1.

Node와 node들을 연결하고 있는 edge가 있는 모양을 우리는 그래프라고 한다. 이런 그래프 전체적으로 모두 탐색하는 방법에는 흔히 DFS, BFS 가 있다. 우선 DFS 는 Depth Frist Search 의 약자로 깊이를 우선적으로 탐색을 한다는 의미이다. 즉 기준 node를 정하여 계속 한쪽으로 탐색을 하고 만약 더 이상 탐색할 곳이 없다면 다시 그 이전 node 로 돌아와 거기서 다시 기준을 정해 한쪽으로 계속 탐색을 진행한다. 즉 가장 나중에 들어온 node에 다시 접근하여 거기에 edge로 연결된 node들을 탐색해야 하기 때문에 recursive 나 iterative 한 방법으로 구현을 해야 한다. (stack 같은 경우 가장 나중에 들어온 node가 top을 의미하고 가장 먼저 pop을 할 수 있다.)

BFS 같은 경우 Breadth first search 로 너비를 우선적으로 탐색을 한다는 의미이다. 즉 level 별로 탐색을 진행하는데 기준 노드의 edge로 연결된 모든 node들을 queue에 저장하여 반복문이 진행할 때마다 queue에서 하나씩 꺼내어 그 꺼내진 node 의 인접한 node 들을 방문했는지 안 했는지 판단하고 만약 방문을 안 했다면 그 node를 다시 queue에 넣는 형태이다.

즉 두 방법 모두 그래프의 모든 node를 탐색하는 것이기 때문에 시간 복잡도 측면에서는 같을 것이다. 이때 시간 복잡도는 탐색을 하고자 하는 node들의 방문 여부를 결정하는데 필요한 시간, 탐색 node 를 방문하는데 필요한 시간을 더하면 된다. 만약 node의 개수가 N개이고 edge의 개수가 E 개라면 모든 node를 방문하는 데는 O(N) 이 걸린다. 그리고 이 탐색 알고리즘은 인접 행렬과 인접 리스트로 표현 할 수 있다. 우선 dfs 같은 경우 인접 행렬로 표현할 시 방문 여부를 N번 확인해야 하니 최종 시간 복잡도는 O(N^2) 이 된다. 그리고 인접 리스트 같은 경우는 각 node당 edge의 개수가 불분명 하지만 결국 전체적인 시간 복잡도로 확인하면 edge들을 이용하여 방문여부 판단을 할 수 있기에 전체적인 시간 복잡도는 O(N+E) 이다. Bfs 역시 dfs와 전체적인 시간 복잡도 측면에서는 동일하다. 즉 bfs의 시간 복잡도는 행렬로 표현 할 시 O(N^2). (인접 행렬이기 때문에 node 방문 여부에 O(N), 모든 node 탐색 O(N)), 리스트로 표현할 시 O(N+E) (모든 node 방문 O(N), node 방문 여부 결정 O(E))가 필요함을 알 수 있다. 즉 인접 리스트가 알고리즘 표현하는 측면에서는 약간 더 복잡할 수 있으나 시간 복잡도 측면에서는 인접 행렬 보다는 인접 리스트로 표현하는 것이 더 효율적이다.

우리는 미로의 전체 경로 및 최적 경로를 모두 탐색해야 하는데 이를 DFS, BFS 로 표현이 가능함을 알 수 있다.

1-2.

우리는 DFS 알고리즘을 이용하여 실습시간에 진행해야 할 가장 왼쪽 위인 시작 방에서 가장 오른쪽 아래 끝방까지의 미로의 최적 경로 탐색 및 최적 경로에서 실제 어느 경로를 탐색했는지 알 수 있는 전체 경로를 그림으로 표현해야 한다. 이 때 DFS는 recursive 한 방법이 아닌 iterative 한 방법으로 이용할 것 인데 그 이유는 재귀적으로 계속 dfs가 호출 될 때 컴퓨터 의 memory 의 stack 영역에 계속 해서 쌓이게 된다. Size가 작은 미로는 괜찮지만 만약 미로가 엄청 커진다면 이는 memory overflow 가 발생 할 수도 있기 때문에 반목문을 통해 stack을 가지고 표현하는 방법을 사용 하는게 overflow 측면에서 유리하다.

현재 2주차 까지의 실습을 통해 미로를 .maz 파일로부터 읽어 들이면 input 파일에 미로 에 대한 모든 정보가 다 저장이 되어 있다. 그리고 우리는 visited 2차원 동적 배열 을 이용하여 방들의 방문 여부를 결정 할 것이다. 다음은 간단한 DFS 및 bfs 함수의 pseudo 코드이다. 경로를 찾는 방법은 주석을 통해 설명하였다.

DFS()

Visited=new int\*[HEIGHT\*2+1] //visited 2차원 배열의 [HEIGHT\*2+1][WIDTH\*2+1] 만큼 동적 할당 메모리 할당을 받는다. 여기서 HEIGHT\*2+1 인 이유는 실제 미로의 높이는 HEIGHT 이고 우리가 .maz로 부터 읽어 들인 미로의 전체 높이는 HEIGHT\*2+1 이다. (높이를 전체로 보면 under 벽들이나 가장 위 아래의 테두리도 포함되기 때문이다.) 전체 미로의 가로 길이도 마찬가지로 WIDHT\*2+1 이다.

for(int i=0;i<HEIGHT\*2+1;i++)

vistited=new int[WIDTH\*2+1]

for(int i=0;i<HEIGHT\*2+1;i++)

for( int j=0;j<WIDTH\*2+1;j++) //2중 for문을 돌면서 벽이 그려져 있는 부분에서는 애초에 미로를 탐색할 수 없기 때문에 이 곳에서는 visited를 1로 하여 방문을 하지 못하게 한다.

if(input[i][j]==’ ‘ ) visitied[i][j]=0 //

else visited[i][j]=1

dfs\_path\_x.push(1) //최적 경로를 탐색 및 저장하는 stack이다.

dfs\_path\_y.push(1) //가장 첫 노드인 (1,1)을 stack에 넣어준다.

dfs\_all\_path\_x.push(1)// 전체 경로를 탐색 및 저장하는 stack 이다.

dfs\_all\_path\_y.push(1) //가장 첫 노드인 (1,1)을 stack에 넣어준다.

Visited[1][1]=1 //(1,1) 은 방문하였음을 알린다.

While(!dfs\_path\_x.empty()&&!dfs\_path\_y.empty()) //x좌표를 저장하는 stack, y좌표르 저장하는 stack 모두 비었지 않았다면

If(dfs\_path\_x.top()==WIDTH\*2-1&&dfs\_path\_y.top()==HEIGHT\*2-1) //만약 stack의 x좌표와 y좌표가 우리가 탐색을 원하는 최종 경로의 좌표라면 Break; //반복문을 멈춰 탐색을 더이상 하지 않는다.

Int x=dfs\_path\_x.top(); // stack\_x 의 top 을 x에 넣는다.

Int y=dfs\_path\_y.top(); // stack\_y 의 top 을 y에 넣는다.

If(!visited[y+1][x]) //만약 현재 좌표 기준 아래 좌표를 방문하지 않았다면

dfs\_all\_path\_x.push(x) //전체 경로 및 최적경로의 stack에 아래 좌표를 넣어주어 방문을 한다.

dfs\_all\_path\_y.push(y+1)

dfs\_path\_x.push(x)

dfs\_path\_y.push(y+1)

visited[y+1][x]=1 // 이 좌표는 이제 방문하였다.

Continue //다음 좌표로 바로 이동할 수 있도록 한다.

If(!visited[y-1][x]) //만약 현재 좌표 기준 윗 좌표를 방문하지 않았다면

dfs\_all\_path\_x.push(x) //전체 경로 및 최적경로의 stack에 윗 좌표를 넣어주어 방문을 한다.

dfs\_all\_path\_y.push(y-1)

dfs\_path\_x.push(x)

dfs\_path\_y.push(y-1)

visited[y-1][x]=1 // 이 좌표는 이제 방문하였다.

Continue //다음 좌표로 바로 이동할 수 있도록 한다.

If(!visited[y][x+1]) //만약 현재 좌표 기준 오른쪽 좌표를 방문하지 않았다면

dfs\_all\_path\_x.push(x+1) //전체 경로 및 최적경로의 stack에 오른쪽 좌표를 넣어주어 방문을 한다.

dfs\_all\_path\_y.push(y)

dfs\_path\_x.push(x+1)

dfs\_path\_y.push(y)

visited[y][x+1]=1 // 이 좌표는 이제 방문하였다.

Continue //다음 좌표로 바로 이동할 수 있도록 한다.

If(!visited[y][x-1]) //만약 현재 좌표 기준 왼쪽 좌표를 방문하지 않았다면

dfs\_all\_path\_x.push(x-1) //전체 경로 및 최적경로의 stack에 왼쪽 좌표를 넣어주어 방문을 한다.

dfs\_all\_path\_y.push(y)

dfs\_path\_x.push(x-1)

dfs\_path\_y.push(y)

visited[y][x-1]=1 // 이 좌표는 이제 방문하였다.

Continue //다음 좌표로 바로 이동할 수 있도록 한다.

Else //현재 좌표 기준 위, 아래, 오른쪽, 왼쪽 모두 이미 방문을 더이상 방문할 곳이 없다면 dfs\_path\_x.pop() //최적 경로의 stack을 pop 하여 이 좌표는 최적 경로의 좌표가 아님을 알린다.

Dfs\_path\_y.pop()

Dfs\_all\_path\_x.push(dfs\_path\_x.top()) //모든 경로의 stack 에는 pop 한 이후의 좌표를 다시 넣어주어 돌아가는 전체 경로를 그릴 수 있도록 한다.

Dfs\_all\_path\_y.push(dfs\_path\_y.top())

이렇게 하면 최적 경로를 저장하는 dfs\_path\_x, dfs\_path\_y 에 각 각 (x,y) 좌표를 모두 이으면 최적 경로가 완성된다. for문을 돌면서 가장 최종 좌표부터 역으로 최적 경로를 그려주는 방식으로 하면 될 것이다. 전체 경로 역시 dfs\_all\_path\_x, dfs\_all\_path\_y 에 각 (x,y) 좌표를 모두 이으면 전체 경로의 좌표가 역으로 나와 전체 경로를 그릴 수 있다. 이는 stack의 가장 최근에 들어온 좌표들을 pop 을 하면서 이전 좌표들로 이동 할 수 있다.

BFS()

Visited=new int\*[HEIGHT\*2+1] //visited 2차원 배열의 [HEIGHT\*2+1][WIDTH\*2+1] 만큼 동적 할당 메모리 할당을 받는다. 여기서 HEIGHT\*2+1 인 이유는 실제 미로의 높이는 HEIGHT 이고 우리가 .maz로 부터 읽어 들인 미로의 전체 높이는 HEIGHT\*2+1 이다. (높이를 전체로 보면 under 벽들이나 가장 위 아래의 테두리도 포함되기 때문이다.) 전체 미로의 가로 길이도 마찬가지로 WIDHT\*2+1 이다.

for(int i=0;i<HEIGHT\*2+1;i++)

vistited=new int[WIDTH\*2+1]

for(int i=0;i<HEIGHT\*2+1;i++)

for( int j=0;j<WIDTH\*2+1;j++) //2중 for문을 돌면서 벽이 그려져 있는 부분에서는 애초에 미로를 탐색할 수 없기 때문에 이 곳에서는 visited를 1로 하여 방문을 하지 못하게 한다.

if(input[i][j]==’ ‘ ) visitied[i][j]=0 //

else visited[i][j]=1

bfs\_path\_x.push(1) //bfs 경로탐색에 사용하는 queue이다.

bfs\_path\_y.push(1) //가장 첫 노드인 (1,1)을 queue에 넣어준다.

Visited[1][1]=1 //(1,1) 은 방문하였음을 알린다.

While(!bfs\_path\_x.empty()&&!bfs\_path\_y.empty()) //만약 경로 탐색에 사용하는 queue 가 비었지 않다면 계속해서 반목문을 실행한다.

//(x,y) 좌표를 queue의 front로 꺼내어 해당 (x,y)좌표의 인접한 좌표들을 모두 queue에 저장할 것이다.

Int x=bfs\_path.x.front() //x에 queue에 가장 먼저 들어온 x 좌표를 꺼내 저장한다.

Int y=bfs\_path.y.front() //y에 queue에 가장 먼저 들어온 y 좌표를 꺼내 저장한다.

Bfs\_all\_path\_x.push(x) //bfs 전체 경로를 그릴 때 사용하는 x 좌표의 queue에 현재의 x 좌표를 넣는다.

Bfs\_all\_path\_y.push(y) //bfs 전체 경로를 그릴 때 사용하는 y 좌표의 queue에 현재의 y 좌표를 넣는다.

If(x==WIDTH\*2-1&&y==HEIGHT\*2-1) //만약 x좌표와 y좌표가 우리가 탐색을 원하는 최종 경로의 좌표라면 Break; //반복문을 멈춰 탐색을 더이상 하지 않는다.

Bfs\_path\_x.pop() //아니라면 해당 좌표 기준 인접한 좌표들을 탐색해야 한다. 이 때 queue를 미리 pop 해줄 것이다.

Bfs\_path\_y.pop()

If(!visited[y+1][x]) //만약 현재 좌표 기준 아래 좌표를 방문하지 않았다면

bfs\_path\_x.push(x) //아래 좌표를 경로 탐색에 사용하는 queue에 넣는다.

bfs\_path\_y.push(y+1)

visited[y+1][x]=1 // 이 좌표는 이제 방문하였다.

If(!visited[y-1][x]) //만약 현재 좌표 기준 윗 좌표를 방문하지 않았다면

bfs\_path\_x.push(x) //윗 좌표를 경로 탐색에 사용하는 queue에 넣는다.

bfs\_path\_y.push(y-1)

visited[y-1][x]=1 // 이 좌표는 이제 방문하였다.

If(!visited[y][x+1]) //만약 현재 좌표 기준 오른쪽 좌표를 방문하지 않았다면

bfs\_path\_x.push(x+1) //오른쪽 좌표를 경로 탐색에 사용하는 queue에 넣는다.

bfs\_path\_y.push(y)

visited[y][x+1]=1 // 이 좌표는 이제 방문하였다.

If(!visited[y][x-1]) //만약 현재 좌표 기준 왼쪽 좌표를 방문하지 않았다면

bfs\_path\_x.push(x-1) //왼쪽 좌표를 경로 탐색에 사용하는 queue에 넣는다.

bfs\_path\_y.push(y)

visited[y][x-1]=1 // 이 좌표는 이제 방문하였다.

이렇게 하면 bfs\_all\_path\_x, bfs\_all\_path\_y queue 에는 (1,1) 부터 마지막 (WIDTH\*2-1,HEIGHT\*2-1)인 최종 목적지 좌표까지 순차적으로 들어가 있다. 이를 통해 for문으로 순서대로 pop 을 하여 전체 경로를 그리면 될 것이다. Dfs와 차이는 dfs 는 목적지 좌표부터 역으로 그림을 그리는데 비해 여기서는 시작 좌표부터 순서대로 경로를 그리는 것이다. 그리고 bfs의 최적 경로의 경우 마지막 최종 목적지의 좌표의 부모 좌표를 계속해서 역으로 따라가면 자연스럽게 (1,1) 시작 좌표가 나오는데 이가 bfs 탐색 시의 최적 경로이다. 즉 위의 pseudo code에서 부모 좌표를 저장하는 queue 자료구조나 2차원 배열 구조를 사용하여 저장하면 최적 경로 역시 그릴 수 있다.