13주차 결과보고서

전공: 컴퓨터공학과 학년: 2학년 학번: 20191619 이름: 이동석

**1. DFS 및 BFS 자료구조와 알고리즘**

DFS와 BFS 모두 방문 여부를 체크 할 배열이 필요했다. 실제 함수에서는 다른 변수명으로 진행했지만, 여기서는 같은 변수명으로 설명한다. 변수 visit은 2차원 포인터배열로 그 크기는 앞선 보고서에서 설명했듯이 텍스트로 미로를 표현할 때, 필요한 배열의 크기와 같다. 즉, 실제 미로의 너비, 높이 보다 2배 +1 만큼의 크기를 가진다.

visited = (int \*\*)malloc(sizeof(int \*) \*HEIGHT);

for (int i = 0; i < HEIGHT; i++)

visited[i] = (int \*)malloc(sizeof(int) \*WIDTH);

다음과 같은 배열에 초기 값을 설정해주어야 한다. 우선 보통, 미로탐색을 배열로 표현할 때, 움직일 수 있는 공간은 1로 벽의 경우 0으로 설정해준다. 하지만, 이렇게 했을 경우 방문했던 길을 표현하기 어려웠다. 그래서, 할당 된 값을 보고 visit배열의 colum과 row를 유츄할 수 있는 값을 주기로 했다.

for (int cnt = 0; cnt < HEIGHT; cnt++) {

for (int j = 0; j < WIDTH; j++) {

if (input[cnt][j] == '+' || input[cnt][j] == '-' || input[cnt][j] == '|') visited[cnt][j] = 0;

else {

visited[cnt][j] = 10000 \* cnt + j;

//visited\_2[cnt][j] = 10000 \* cnt + j;

}

}

}

포문을 통해서 visit 배열을 초기화 해준 후, 미로에서 움직일 방향에 대한 배열도 설정한다. 미로에서는 상,하,좌,우로 움직일 수 있다. 이때, 미로의 시작점과 도착점이 고정되어있고, 위치 상 오른쪽 아래 위 왼쪽으로 탐색하는 것이 조금 더 빠를 것이라고 생각했다.

int x[4] = { 1, 0, 0, -1 };

int y[4] = { 0, 1 , -1, 0 };

이후, 목표 지점을 만나면 반복문을 탈출해야 한다. 목표 지점은 문제에서 주어졌다. 따라서, 다음과 같은 값을 할당한다.

int target = visited[HEIGHT-1][WIDTH-1] ;

DFS에서 사용할 stack의 경우에는 stack 라이브러리를 사용했으며, int형 변수 path로 정의했다. 이후, 첫 시작점의 visit 값을 psuh 해주고, visit 배열의 값을 1로 바꾸어준다. 여기서, 1의 의미는 진짜 경로라는 의미이며, 2를 할당해주는 경우는 탐색을 위해 방문 했던 길이라는 의미이다. 현재, path에는 visited[1][1]의 값 10001이 들어있다. 모든 준비가 끝나면, 이제 while 반복문을 통해 길을 찾는다.

while (!path.empty()) {

i = path.top();

i = i / 10000;

j = path.top();

j = j % 10000;

flag = 1;

if (i == HEIGHT - 2 && j == WIDTH - 2) break;

for (t = 0; t < 4; t++) {

if (visited[i + y[t]][j + x[t]] >10000) {

path.push(visited[i + y[t]][j + x[t]]);

visited[i + y[t]][j + x[t]] = 1;

flag = 0;

break;

}

}

if (flag) {

visited[i][j] = 2;

path.pop();

}

}

path.top()에는 10001이라는 값이 들어있다. 이를 이용해, i와 j는 현재 visit의 위치를 나타낸다. 즉, 현재 내 위치는 visit[1][1]에 있다는 것이다. if문은 도착지점을 만날 경우 반복문을 탈출한다. for문은 현재 위치에서 이동할 수 있는 정점이 있을 경우, 앞서 말한 방향 배열을 통해 순서대로 방문하고 push 해준다. 그렇게 계속 이동하면서 더 이상 방문할 수 있는 정점이 존재하지 않는다면, 계속해서 pop을 한다. 앞서 말했듯이, pop을 한다는 것은 우리가 원하는 길이 아닌, 탐색을 위해 방문한 길이고 그렇기에 visit의 값을 2로 설정해준다. 즉, 최종적으로 visit에는 아예 방문하지 않은 노드는 10000이상의 값을 가지고 있고, 벽은 0, 길은 1, 방문 했던 길은 2로 값이 저장되어있다. 이 값들을 이용해, dfsdraw 함수에서 최종적으로 길을 그려주기만 하면 된다.

최종적인 함수의 시간복잡도는 결국, 생각했던 것 보다는 오래 걸리게 되었다. 왜냐하면, 보통 인접 리스트로 구현한 DFS의 경우 O(V+E)로 나타낸다. 하지만, 구현시 사용했던 미로에서 통로 역시 방처럼 생각했기 때문에 정점의 개수가 빈방의 개수가 아닌 그 정점의 개수 그 자체로 V+E가 되었다. 더불어, 늘어난 정점마다 다시 간선들이 존재하는 것 처럼 이동할 수 있는지 여부를 확인해주었다.

하지만, 시간복잡도는 매 시행마다 다르게 되는데 이는 목표지점을 만나는 경우 일반 DFS와는 다르게 바로 탈출을 하기 때문이다. 물론 최악의 경우에는 늘어난 정점과 늘어난 간선의 개수의 합이 되고 O(V+E)의 형태가 된다.

공간복잡도의 경우에는 기본적으로 visit배열의 크기인 O(HEIGHT\*WIDTH)와 stack에서 길의 경로를 저장한 만큼이 되게된다. Stack의 크기는 매 시행마다 다르고 예측할 수 없지만, 가장 작을 경우 O(HEIGHT + WIDTH) 만큼이 된다.

BFS의 경우에는 DFS와 아주 조금 다르다. BFS의 경우 DFS와는 다르게 경로를 따로 저장해주는 함수가 필요하고 queue를 이용한다. 방식은 앞과 같은 실제 경로의 경우에는 1을 아닐 경우에는 2로 저장해준다.

temp = (int \*\*)malloc(sizeof(int \*) \* HEIGHT);

for (i = 0; i < HEIGHT; i++)

temp[i] = (int \*)malloc(sizeof(int) \*WIDTH);

while (!path\_bfs.empty()) {

i = path\_bfs.front();

i = i / 10000;

j = path\_bfs.front();

j = j % 10000;

path\_bfs.pop();

if (i == HEIGHT - 2 && j == WIDTH - 2) break;

for (t = 0; t < 4; t++) {

if (visited\_2[i + y[t]][j + x[t]] > 10000) {

path\_bfs.push(visited\_2[i + y[t]][j + x[t]]);

temp[i + y[t]][j + x[t]] = num;

visited\_2[i + y[t]][j + x[t]] = 2;

}

}

num = path\_bfs.front();

}

visited\_2[i][j] = 1;

while (i != 1 || j != 1) {

num = temp[i][j];

i = num / 10000;

j = num % 10000;

visited\_2[i][j] = 1;

}

DFS와 가장 큰 차이점은 for문에 break문이 빠진 점이다. break문이 빠짐으로서 현재 정점에서 방문 가능한 정점들을 모두 방문하고 queue에 push하게 된다. 이후 모두 방문했으면, pop을 해주면서 그 위치에서 다시 방문 가능한 노드들을 모두 방문한다. 특이하게도 경로를 저장하는 temp에는 자신이 어디로부터 왔는지가 저장된다. 이를 이용해, 도착지점에서 자신이 어디서 왔는지를 계속 거슬러 올라가다보면, 출발점이 나오게 된다. 2번째 while문에서 거꾸로 거슬러 올라가며 그 위치들의 visit배열의 값을 1로 바꾸어준다. 경로를 그리는 방법은 DFS에서 그리는 방법과 완벽하게 동일하게 된다.

공간복잡도의 경우 DFS보다 2차원 포인터배열을 하나 더 사용했기 때문에, O(2\*HEIGHT\*WIDTH) 가 된다. queue의 경우도 매 시행마다 달라지므로 예측하기가 힘들다.

시간복잡도의 경우 DFS에서의 설명과 동일함을 알 수 있다.

**2. DFS와 BFS의 장단점**

DFS와 BFS의 작동방식을 보면 완전미로에서는 DFS가 조금 더 효율적이라고 볼 수 있다. 완전미로의 경우 길은 항상 한가지만 존재하게 되는데, BFS는 이동하는 모든 정점에서 방문가능한 노드들을 모두 확인하게 되므로 BRUTE FORCE느낌이 강하다. 또한, BFS의 장점은 길이 여러가지인 경우 최단거리를 항상 찾아낸다는 점이지만, 길이 한가지인 완전미로에서는 그 장점이 무색하게 된다. DFS의 경우 BFS에 비해 사용하는 공간복잡도가 작고, 운이 좋다면 BFS보다 훨씬 빠르게 경로를 탐색할 수 있다. 결국 현재 실험에 대해서는 DFS가 모든 면에서 더 나은 자료구조로 볼 수 있다.