13주차 결과보고서

전공: 철학과 학년: 3학년 학번: 20180032 이름: 남기동

1. DFS, BFS의 알고리즘과 자료구조 요약 기술, 시공간 복잡도, 달라진 점

1) DFS

```
visited = new int[num_of_terms];
stack = new int[num_of_terms];
all = new int[num_of_terms * 4];

for (i = 0; i < num_of_terms; i++) visited[i] = 0;
visited[0] = 1; //시작점은 0번째 노드(좌측 상단)
current_pos = 0;//0번째 노드가 현 위치
push(current_pos);
```

실질적인 DFS로 경로를 구하기 이전에 필요한 작업들이다. Visited는 노드의 방문 여부를 저장하고 stack은 시작점부터 도착점에 도달할 때까지의 경로를 저장하는 용도로, all은 stack에 남지 않는 모든 경로를 남겨 놓기 위한 용도로 활용된다. Visited를 0으로 초기화하여 방문하지 않았음을 의미하게 하고 0번째 노드, 즉 좌측 상단의 노드를 시작점으로 하기에 visited[0]에 1을 대입하고 해당 위치를 스택에 push한다.

```
while (top > -1 && found == false) {
                while (current_pos == stack[top] && top > 0) {
                        current_pos = pop();
                }
                current_pos = pop(); all[++top_all] = current_pos;
                ptr = graph[current_pos];
                while (dir < 4 && found == false) {
                        if (ptr->link[dir] == NULL || visited[ptr->link[dir]->index] == 1)
{ //벽이라서 or 이미 지나온 곳이라서 못가는 방향일 때
                                dir++;
                        }
                        else if (ptr->link[dir]->index == exit) { //탈출구에 도착한 경우
                                if(stack[top] != ptr->index) push(current_pos);
                                found = true;
                                ptr = ptr->link[dir];
                                current_pos = ptr->index;
                                push(current_pos);
                        }
                        else { //이동 할 수 있는 경우
```

```
                        if (stack[top] != ptr->index) push(current_pos);
                        ptr = ptr->link[dir];
                        current_pos = ptr->index;
                        push(current_pos);
                        visited[current_pos] = 1;
                        dir = 0; //새로운 지점이니깐 dir도 초기화
                    }
                }
                dir = 0;
                //found가 true가 되었으면 마지막 지점에 ptr이 위치하고 stack도 마지막 지점의
index까지 쌓아져 있는 상황에서 while loop 바로 탈출
            }
```

DFS를 구하는 실질적인 부분은 두 개의 while loop으로 이루어져 있다. Outer loop은 도착점을 찾을 때 나오게 설정하고 inner loop은 4개의 방향을 모두 탐색한 경우 나오게 설정한다. 스택에 저장되어 있는 위치를 pop해서 불러오고 해당 위치를 ptr로 가리킨다. 그리고 해당 노드의 4가지 link를 탐색하게 되는데, 이 때는 3가지 경우가 존재한다. 첫 번째는 벽이라서 혹은 이미 지나온 곳이라서 가지 못하는 경우이다. 이 때는 다음 방향을 탐색하면 되므로 dir에 1을 더하기만 한다. 두 번째 경우는 탈출구에 도착한 경우로 해당 노드로 이동한 다음 마지막 위치를 push하고 found를 true로 변경하여 outer loop을 벗어날 수 있도록 한다. 마지막 경우는 첫 번째도 두 번째도 아닌 경우로 이동할 수 있는 경우를 의미한다. 이 때는 해당 노드로 이동하고 위치를 스택에 push하고 visited도 1로 변경하고 dir은 0으로 변경시킨다. Dir을 0으로 변경시키는 이유는 새로운 노드로 넘어갔으니 4 방향을 처음부터 조사할 수 있도록 하기 위함이다.

DFS의 시간 복잡도는 노드의 개수를 v, 노드끼리 연결하는 edge를 e라고 할 때 O(v+e)이다. 왜냐하면 worst case의 경우 모든 노드와 연결 링크를 탐색하게 되기 때문이다. 공간 복잡도는 노드의 개수가 n개일 때 O(n)이다. 내가 선언한 자료구조에 따르면, 노드 안에 4가지 방향에 해당하는 값도 포함되어 있다. 따라서 n개의 노드가 있다고 한다면 O(n)의 공간복잡도를 갖는다.

2) BFS

```
visited = new int[num_of_terms];
queue = new int[num_of_terms];
all = new int[num_of_terms * 4];
stack = new int[num_of_terms];

for (i = 0; i < num_of_terms; i++) visited[i] = 0;
```

```
visited[0] = 1; //시작점은 0번째 노드(좌측 상단)
current_pos = 0;//0번째 노드가 현 위치
addq(current_pos);
```

BFS도 DFS와 유사한데 차이점은 queue를 활용한다는 것이다. Stack도 동적 할당한 것을 확인할 수 있는데 이것은 BFS로 경로를 탐색한 이후 최단 경로가 어떻게 되는지 이후에 다시 구하는 과정에서 사용하기 위해 사용했다. DFS와 유사하게 첫 번째 노드를 queue에 더하고 시작한다.

```
while (front != rear && found == false) {
                current_pos = deleteq();
                ptr = graph[current_pos];
                all[++rear_all] = ptr->index;

                for (dir = 0; dir < 4; dir++) {
                        if (ptr->link[dir] != NULL) { //벽이 아니고
                                if (visited[ptr->link[dir]->index] == 0) { //방문된 곳도 아닌
경우(갈 수 있는 방향)
                                        if (flag == 1) all[++rear_all] = ptr->index;
                                        addq(ptr->link[dir]->index);
                                        cout << ptr->link[dir]->index << " ";
                                        visited[ptr->link[dir]->index] = 1;
                                        flag = 1;

                                        if (ptr->link[dir]->index == exit) {//탈출구인 경우
                                                found = true;
                                        }
                                }
                        }
                }
                if (flag == 1) all[++rear_all] = ptr->index; //다른 위치랑 섞이지 않도록
                flag = 0; //다음 위치로 넘어갔을 때를 위한 초기화
        }
```

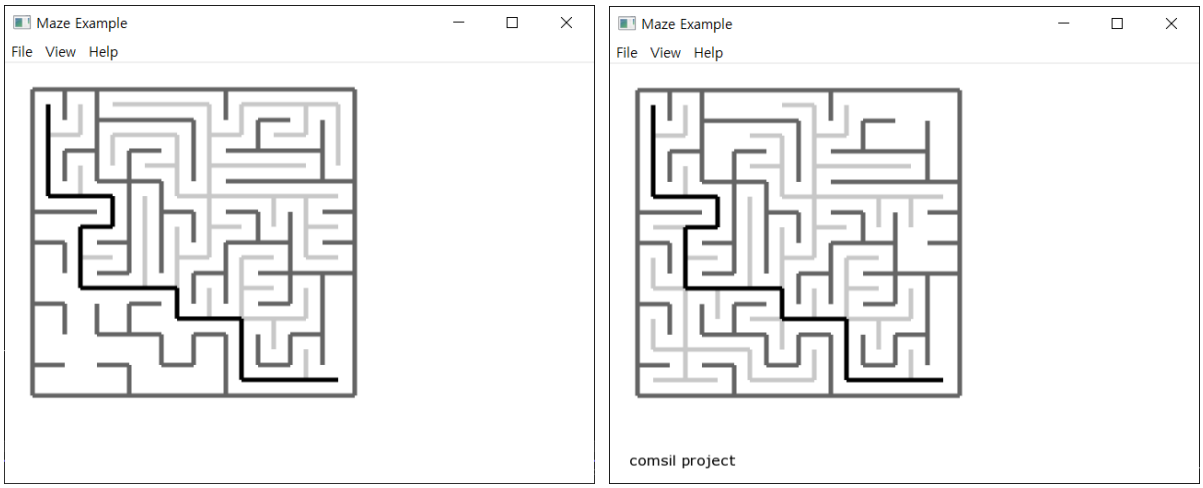
BFS는 while loop을 outer loop으로 for loop을 inner loop으로 갖는 방식으로 구현했다. Outer loop은 front가 rear와 같은 경우 나오게 되는데, 이 경우는 queue에 있는 값들을 모두 탐색한 경우이다. While loop에 들어가서 queue에 있는 값을 하나 지우고 해당 위치를 받아와 ptr로 가리킨다. 그리고 inner loop인 for loop에서 4가지 방향으로 조사하여 벽도 아니고 방문한 곳도 아닌 곳, 즉 이동할 수 있는 방향의 값들을 모두 queue에 add한다. 이렇게 queue에 노드들을 더하다가 탈출구에 도달한 경우 found를 true로 바꾼다. 이 때 중요한 것은 found가 true가 되었더라도 break를 따로 사용하지 않아 탈출구와 동일한 거리의 다른 지점들도 queue에 더할 수 있도록 한다.

BFS의 시간 복잡도는 DFS와 동일하게 노드의 개수를 v, 노드끼리 연결하는 edge를 e라고 할 때 O(v+e)이다. 왜냐하면 worst case의 경우 모든 노드와 연결 링크를 탐색하게 되기 때문이다. 공간

복잡도는 노드의 개수가 n개일 때 O(n)이다. 내가 선언한 자료구조에 따르면, 노드 안에 4가지 방향에 해당하는 값도 포함되어 있다. 따라서 n개의 노드가 있다고 한다면 O(n)의 공간복잡도를 갖는다.

실습 이전에 생각한 방법과 DFS, BFS의 구현에서는 크게 달라진 점이 없다. 그러나 실질적으로 DFS와 BFS를 통해 경로를 탐색하는 것을 넘어 화면에 탐색했던 전체 경로와 최단 거리를 모두 표시하여야 하기 때문에 이를 위해서 경로의 정보를 저장하는 것에서 어려움이 있었다. DFS의 경우에는 stack에 쌓여 있는 값들이 이미 최단 거리로 존재했기 때문에 비교적 수월했지만 BFS의 경우에는 최단 거리의 경로가 queue에 저장되어 있지 않고 거리를 기준으로 저장되어 있기 때문에 이를 처리하는 과정이 생각보다 복잡했던 것 같다.

2. DFS, BFS의 장단점 비교, 자신의 자료구조에 어떤 알고리즘이 더 적합한가



좌측이 DFS를 사용하여 경로를 탐색한 경우이며 우측이 BFS를 사용하여 경로를 탐색한 경우이다. 두 경우 모두 경로를 잘 탐색하여 탈출구를 잘 찾았음을 확인할 수 있다. 그러나 화면에서 볼 수 있듯, DFS에 비해 BFS가 탐색한 경로가 더 많다는 것을 알 수 있다. 아마 이러한 차이가 발생한 이유는 실습에서 하는 미로 해결에서 좌측 상단을 출발점으로 우측 하단을 탈출구로 가정하기 때문인 것 같다. 좌측 상단에서 우측 하단으로, 즉 대각선 방향으로 이동해야 하는데 BFS는 동일한 거리에 있는 노드들로 퍼져 나가면서 탐색하기 때문에 DFS에 비해 탐색하는 노드의 개수가 비교적 많아진 것 같다. 현재 실습에서 하는 가정에서 심지어 노드에서 탐색하는 방향을 위쪽, 오른쪽,

아래쪽, 왼쪽으로 하는 것이 아니라 오른쪽, 아래쪽, 왼쪽, 위쪽으로 변경하면 DFS가 경로를 찾는 과정에서 탐색하게 되는 노드의 수가 더 줄어들 것으로 예상된다. 그러나 이렇게 변경한다고 해도 BFS는 각각의 노드에서 전 방위로 갈 수 있는 노드를 모두 queue에 add하기 때문에 크게 달라지는 것을 없을 것으로 예상된다.

그리고 DFS와 BFS를 고려할 때 화면에 경로를 그리는 것에 대해서도 고려해봐야 한다. DFS의 경우 경로 탐색을 완료한 이후 stack에 최단 경로의 정보만 남아 있기 때문에 DFS 경로를 그리는 과정이 BFS에 비해 훨씬 수월하다. 이에 반해 BFS는 경로 탐색 이후 queue에 남아있는 값들이 연속적이지 않기 때문에 한 차례 정리 과정을 거쳐야 최단 경로를 구할 수 있다. 왜냐하면 시작점으로부터 이동 거리를 기준으로 queue에 순차적으로 더해지기 때문이다. 따라서 단순히 경로만을 구하는 것을 넘어 화면에 그려야 하는 것까지 요구되는 실습의 경우 BFS보다 DFS가 유용한 것 같다.

이번 실습을 위해 구현한 자료구조에서 DFS와 BFS를 구현하는데 있어서 어느 쪽에 더 적합한지는 차이가 나지 않는다고 생각한다. 단, 실습이 요구하는 가정들을 고려할 때 DFS가 비교적 적은 노드를 탐색하고도 탈출구를 찾을 수 있을 것으로 예상되고 경로를 구하고 그림을 그려야 한다는 조건에서도 DFS가 보다 효율적인 것을 확인할 수 있었다.