13주차 결과보고서

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

**1.**

1 - 1. DFS

#include <stack>을 ofApp.h 파일에 선언하고, ofApp class 안에 stack<int>로 best\_stack\_x, best\_stack\_y, all\_stack\_x, all\_stack\_y를 선언하였다. best\_stack은 최적 경로, all\_stack은 전체 경로를 의미하고 x, y는 각 위치의 좌표를 의미한다.

bool ofApp::DFS()

visited 배열에 대한 설정을 먼저 진행한다. HEIGHT x WIDTH 크기의 2차원 배열을 생성하였고, input 배열의 원소가 ‘ ‘인 위치만 접근 가능하다는 뜻으로 0을 값으로 주고, 나머지는 1을 주었다.

처음 위치 (1, 1)에서 시작하기 때문에 두 stack에 모두 (1, 1)을 push하였다. 동시에 visited[1][1] 값을 1로 바꾸어 이미 지났다는 것을 알게 하였다.

이제 while loop으로 들어간다. loop은 최적 경로 stack이 비어있지 않을 때 돌 수 있도록 조건을 걸어두었다.

현재 위치, 즉 stack의 맨 위에 있는 좌표를 따로 x, y 변수로 저장하였다. 만약 x, y가 미로의 목표 지점에 도달했다면 loop을 끝낸다.

이제 상, 하, 좌, 우, 총 네 방향에 접근 가능한 방이 있는지 확인한다. visited 배열을 이용하여 값이 0일 때 처리할 수 있도록 조건문을 설정하였다. 만약 접근 가능한 방이 있다면, 최적 경로 stack과 전체 경로 stack에 그 방의 위치를 push한다. 또한 이미 방문하였기 때문에 visited 값도 1로 바꿔준다.

만약 네 방향 모두 방문할 수 없다면, 현재 가고 있는 경로는 최적 경로가 아니다. 따라서 최적 경로 stack에서 맨 위의 위치를 pop한다. 전체 경로는 나중에 표시해야 하기 때문에 제거한 다음 위치를 다시 전체 경로 stack에 push한다. 만약 최적 경로 stack에서 pop 한 뒤 stack이 빌 수도 있기 때문에 stack size가 양수일 때만 작동되도록 조건문을 걸어둔다.

마지막으로 isdfs와 isDFS 값을 1로 바꿔주어 dfs 함수가 마무리되었음을 알린다.

bool ofApp::dfsdraw()

먼저 전체 경로를 그리고, 그 다음에 최적 경로를 그려서 두 경로가 모두 보이게끔 구현하였다.

우선 전체 경로 stack의 size만큼 loop을 돈다. 이후 전체 경로 stack의 맨 위 위치를 (x1, y1), 그 아래 위치를 (x2, y2)로 저장한다. 이 과정에서 pop을 하기 전 또는 하고난 후에 stack이 비어버리면 작동할 수가 없기 때문에 조건문을 걸어두어 처리하였다. ofDrawLine 함수를 이용해서 두 점 사이의 선분을 그린다. 현재 방 한 칸의 size가 40이기 때문에 각 좌표에 20을 곱해주었다.

다음 최적 경로를 그리는 과정도 전체 경로와 동일하다. 다만 색깔을 확실히 다르게 하기 위해 전체 경로는 (255, 0, 0), 최적 경로는 (0, 0, 255)로 설정하고 loop에 들어간다. 이 외의 내용은 동일하기 때문에 생략한다.

경로 그림이 완성되었으면 이제 visited 배열은 필요가 없기 때문에 메모리 해제를 시켜주었다. 또한 isdfs 값도 0으로 해주어 dfs가 진행되지 않았음을 알린다.

1 - 2. BFS

#include <queue>를 ofApp.h에 선언하고, class ofApp 안에 queue<int>로 best\_queue\_x, best\_queue\_y, all\_queue\_x, all\_queue\_y를 선언하였다. stack과 마찬가지로 최적 경로, 전체 경로의 x, y 좌표를 의미한다. 또한 struct parent를 선언하였다. 원소로는 int형 변수 x\_pos와 y\_pos를 갖도록 하였고 class 내에 parent 변수 2차원 배열 past를 선언하였다. past의 역할은 bfs를 하는 과정에서 현재 위치의 부모를 저장하는 것이다. 뒤에서부터 찾아야 하기 때문에 최적 경로를 그리기 위해선 이전 위치를 따로 저장해두어야 한다.

bool ofApp::BFS()

visited 배열에 대한 설정을 한다. HEIGHT x WIDTH만큼의 2차원 배열 메모리 할당을 해주고, input값이 ‘ ‘일 때만 0, 나머지는 1을 저장하였다. 다음으로 past에 대한 메모리 할당도 진행하였다. 마찬가지로 HEIGHT x WIDTH 만큼의 parent type 2차원 배열 메모리를 할당하였다.

다음으로 최적 경로 queue에 현재 시작하는 위치인 (1, 1)을 push한다. 전체 경로 queue에 하지 않는 이유는 이후에 while문 초반에 어차피 최적 경로의 front를 전체 경로 queue에 push하기 때문이다. visited[1][1]도 이제 방문이 완료되었으므로 1로 바꿔준다.

이제 while loop을 시작한다. 조건은 queue가 비어있지 않을 때 돌 수 있도록 설정하였다.

우선 최적 경로 queue의 front 위치를 변수 x, y에 따로 저장하고 queue에서 제거한다. x, y는 그대로 전체 경로 queue에 저장한다. 만약 여기서 x, y가 미로의 목표 지점이라면 loop을 종료한다.

이제 상, 하, 좌, 우 네 방향에 대한 탐색을 시작한다. visited 배열을 이용하여 값이 0일 때 작동되도록 조건문을 걸어준다. 만약 방문이 가능하다면, 우선 최적 경로 queue에 그 방의 위치를 push한다. 방문을 했다는 의미로 visited 값은 1로 바꿔준다. 또한 bfs에서 이용하려고 선언한 past 배열의 값을 x, y 위치로 저장한다. 즉, 현재 방문한 방의 부모 노드가 방문하기 이전 방의 위치를 의미하고 있다.

bfs 탐색이 완료되면 isbfs와 isBFS 값을 1로 바꿔주어 탐색이 완료되었음을 알린다.

void ofApp::bfsdraw()

전체 경로 queue의 size만큼 loop을 돈다. 우선 queue의 제일 처음에 있는 좌표 (1, 1)은 이전 방의 위치가 따로 없으므로 미리 pop해주어 오류가 생기지 않도록 한다. loop 안에 들어가면 우선 전체 경로 queue의 front 좌표를 저장하고, 그 좌표의 이전 방 정보를 past 배열을 통해 얻어서, 그 두 위치를 잇는 선을 그린다.

최적 경로는 그리는 과정에서 굳이 queue를 사용할 필요가 없다. 미로의 목표 지점을 변수에 저장하고, 그 지점의 이전 위치를 past 배열을 통해 얻는다. 이제 그 두 점을 잇는다. 이 과정을 while loop을 이용해서 반복하면 처음 위치까지 선을 그릴 수 있다.

경로 그림이 완성되었다면 초기화를 진행한다. 혹시 queue에 남은 원소가 있을 수 있으므로 최적 경로와 전체 경로 queue 모두 남은 원소들을 pop 해준다. 또한 visited 배열과 past 배열도 메모리 해제를 시켜준다.

1 - 3. Complexity

dfs() - time complexity : visited 배열을 초기화하는 과정, 미로의 방을 탐색하는 과정에서 O(WIDTH \* HEIGHT)만큼의 시간이 걸린다. 방을 탐색하는 worst case는 모든 방을 탐색하는 경우이기 때문이다.

dfs() - space complexity : visited 배열이 갖는 크기, 4개의 stack의 크기는 O(WIDTH \* HEIGHT)라고 할 수 있다. 마찬가지로 stack에 들어가는 space의 worst case는 모든 방의 정보가 다 들어가는 경우이기 때문이다.

dfsdraw() - time complexity : 전체 경로를 그리는 과정에서 O(WIDTH \* HEIGHT)만큼의 시간이 걸린다. worst case는 모든 방의 위치를 그린 경우이기 때문이다.

dfsdraw() - space complexity : stack 사용한 부분을 고려하면 O(WIDTH \* HEIGHT)지만, stack에 따로 추가한 부분이 없기 때문에 이를 제외한다면 stack size를 변수에 저장한 것 밖에는 없다. O(1).

bfs() - time complexity : visited 배열을 초기화하는 과정, 미로의 방을 탐색하는 과정에서 O(WIDTH \* HEIGHT)만큼의 시간이 걸린다. 방을 탐색하는 worst case는 모든 방을 탐색하는 경우이기 때문이다.

bfs() - space complexity : visited 배열이 갖는 크기, past 배열이 갖는 크기, 4개의 queue의 크기는 상수를 제외하고 O(WIDHT \* HEIGHT)이다.

bfsdraw() - time complexity : 전체 경로를 그리는 과정에서 O(WIDTH \* HEIGHT)만큼의 시간이 걸린다. worst case는 모든 방의 위치를 그린 경우이기 때문이다.

bfsdraw() - space complexity : queue를 사용한 것을 고려한다면 O(WIDTH \* HEIGHT)지만, 추가한 원소는 따로 없기 때문에 이 부분을 제외한다면 queue size를 변수로 받아온 것 밖에 없다. O(1).

실험 전에 생각했던 pesudo code보다는 디테일하게 많이 추가하여 실제 코드를 구현하였다. 하지만 전체적인 구조는 동일하기 때문에 시간복잡도나 공간복잡도는 실험 전에 생각했던 것과 같다. 추가한 부분으로는 ofSetBackgroundAuto(false)를 통해 그림이 그려져 있는 상태로 유지되게 한 것과 dfs, bfs 버튼을 클릭했을 때 변수가 초기화되고 메모리가 제대로 해제되도록 하는 등의 내용이 있었다.

**2.**

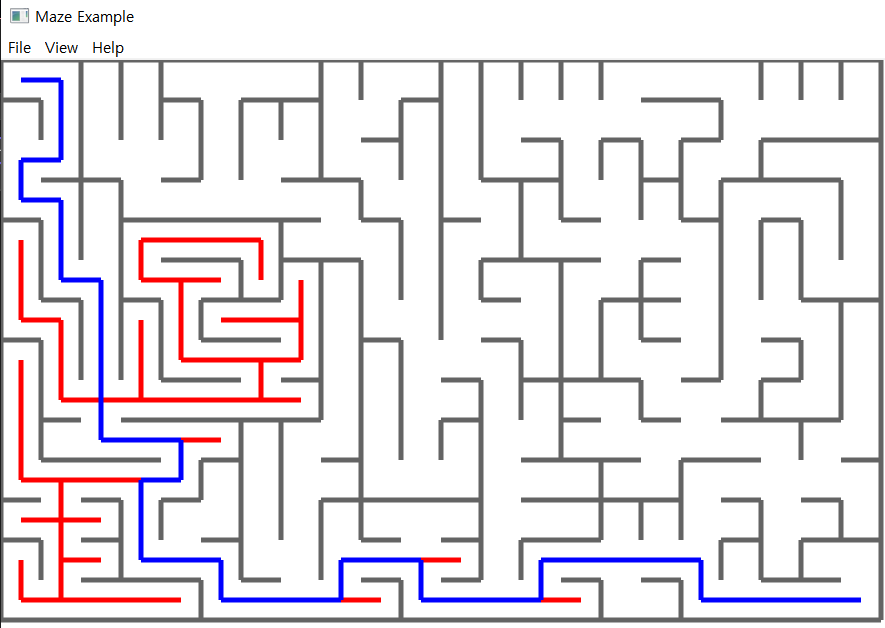
자신이 설계한 프로그램을 실행하여 보고 dfs, bfs 알고리즘을 서로 비교한다.

각각의 알고리즘은 어떤 장단점을 가지고 있는지

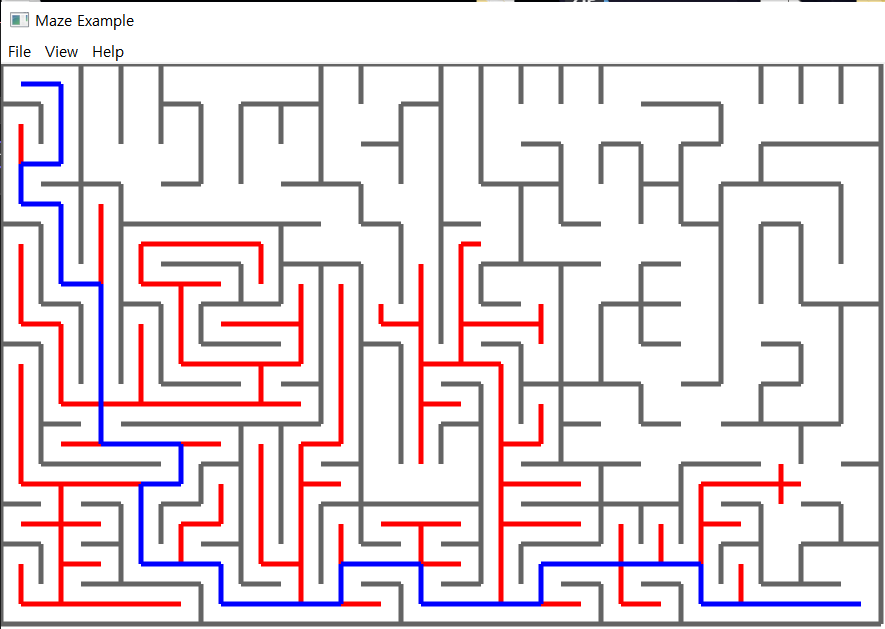
자신의 자료구조에는 어떤 알고리즘이 더 적합한지 등에 대해 관찰하고 설명한다.

2 - 1. DFS와 BFS 경로 비교

(DFS)



(BFS)



코드 구현 중 4가지 방향을 탐색하는 부분이 있다. 나는 현재 위치 기준 오른쪽 방향을 가장 먼저 탐색하였기 때문에 dfs에서 운이 좋게도 오른쪽 방향으로 갔을 때 최적 경로가 나오는 케이스여서 전체 경로와 최적 경로가 크게 다르지 않은 점을 알 수 있다. 다만 bfs에서는 너비 우선 탐색을 하였기 때문에 dfs보다는 아무래도 전체 경로의 범위가 넓다는 점을 확인할 수 있다.

그렇다고 무조건 dfs가 더 좋다고 할 순 없다. bfs는 바로바로 최단 경로를 확인할 수 있다는 장점이 있다. 목표 지점을 준다면 바로 이전 방의 위치를 확인할 수 있기 때문이다. 하지만 이 점이 단점이 되기도 한다. 이를 위해 parent 구조체의 2차원 배열을 새로 생성해야 하기 때문이다. 반면에 dfs는 경로를 탐색할 때 사용했던 stack에 최적 경로가 그대로 저장되어 있기 때문에 추가 메모리를 사용하지 않고도 최적 경로를 확인할 수 있다.

결과적으로 실험 결과와 추가 메모리를 고려하면 본인의 실험에는 dfs 알고리즘이 더 유용한 방법이라고 판단되었다.