13주차 결과보고서

전공: 신문방송학과 학년: 3학년 학번: 20191150 이름: 전현길

1. 실습 및 숙제로 작성한 프로그램의 알고리즘과 자료구조를 요약하여 기술한다. 완성한 알고리즘의 시간 및 공간 복잡도를 보이고 실험 전에 생각한 방법과 어떻게 다른지 아울러 기술한다.

멤버 변수 및 함수

```
bool readFile();
void freeMemory();
bool DFS();
void dfsdraw();
bool BFS();
void bfsdraw();
void resetMaze();

int HEIGHT;         // 미로의 높이
int WIDTH;          // 미로의 너비
char** input;       // 텍스트 파일의 모든 정보를 담는 이차원 배열이다.
int** visited;      // 방문여부를 저장할 포인터
int maze_col;       // 미로칸의 열의 인덱스를 가리킨다.
int maze_row;       // 미로칸의 행의 인덱스를 가리킨다.
int k;
int isOpen;         // 파일이 열렸는지를 판단하는 변수. 0이면 안열렸고 1이면 열렸다.
int isDFS;          // DFS함수를 실행시켰는지 판단하는 변수. 0이면 실행안했고 1이면 실행했다.
int isBFS;          // BFS함수를 실행시켰는지 판단하는 변수. 0이면 실행안했고 1이면 실행했다.
```

직접 작성한 함수와 변수는 위와 같다. BFS(), bfsdraw(), DFS(), dfsdraw() 함수는 각각 알고리즘에 따른 길찾기를 실행하여 visited 이차원 배열의 값을 수정한 후 해당 값에 따라 길찾기의 결과를 그림으로 출력한다. resetMaze() 함수는 방문 여부를 기록하는 visited 배열을 초기화한다.

Vertex 구조체

```
struct Vertex {
    int row, col;
    vector<pair<int, int>> path;
    Vertex(int r, int c) : row(r), col(c) {};
};
```

BFS 알고리즘에서 정점을 저장하기 위해 작성한 구조체이다. Vertex 구조체는 현재 정점의 행과 열, 그리고 이전까지 정점을 이동한 경로를 저장한다. 정점에 경로를 저장하면서 구현은 좀 더 편리해지지만 메모리 낭비가 심해지게 된다.

void ofApp::resetMaze()

```
void ofApp::resetMaze() {
    for (int i = 0; i < HEIGHT; i++)
        for (int j = 0; j < WIDTH; j++)
            if (visited[i][j] > 1) visited[i][j] = 0;
}
```

DFS 알고리즘과 BFS 알고리즘은 실행하기 전에 resetMaze() 함수로 방문 배열을 초기화한다. resetMaze() 함수는 미로를 처음 입력받은 상태 그대로 복구하도록 하는 함수이다.

void ofApp::dfsdraw() / void ofApp::bfsdraw()

```
ofSetLineWidth(5);

for (int i = 0; i < HEIGHT; i++) {
    for (int j = 0; j < WIDTH; j++) {
        ofSetColor(0, 0, 0);
        if (visited[i][j] == 2) ofDrawRectangle(i * 15, j * 15, 15, 15);
        if (visited[i][j] == 3) {
            ofSetColor(200, 50, 50);
            ofDrawRectangle(i * 15, j * 15, 15, 15);
        }
    }
}
```

dfsdraw() 함수는 visited 배열의 값에 따라 미로를 화면에 출력한다. dfs() 알고리즘이 방문한 정점이라면 검은색으로 표시하되, 최단 경로에 속하는 정점이라면 붉은색으로 표시한다. bfsdraw() 함수는 최초에 구상할 때는 dfsdraw() 함수와 서로 다른 동작을 하도록 구현했지만, BFS() 알고리즘의 구조를 변경한 뒤에는 dfsdraw() 함수와 동일한 동작을 하게 되었다.

bool ofApp::DFS()

DFS 알고리즘은 시작 정점을 삽입한 후 동작한다.
스택의 top에 있는 정점의 좌표에 방문 표시를 한 뒤,
(1) 목적지에 도달했는지 체크하고, 도달하지 않았다면 **(2) 미방문 정점이 있는지 체크**한다.

(1) 목적지에 도달했다면 스택에 삽입된 정점을 스택이 빌 때까지 삭제하며 최단 경로를 표시한다.
(2) 미방문 정점이 존재한다면 미방문 정점을 삽입한 뒤 새로운 정점에 대해 동일한 동작을 반복한다.

스택이 모두 비었다면 목적지에 도착한 것이므로 알고리즘을 종료한다. **시간 복잡도는 O(V + E)**지만, 각 정점이 가지는 간선의 개수가 최대 4개이기 때문에 **E를 4 * V로 치환해 O(V)로 볼 수도 있다. 공간 복잡도는 미로 전체(=모든 정점)를 visited 배열에 저장하므로 O(V)**이다.

```
bool ofApp::DFS()//DFS탐색을 하는 함수
{
    //TO DO
    //DFS탐색을 하는 함수 ( 3주차)
    isDFS = true, isBFS = false;
    stack<pair<int, int>> stack;
    stack.push({1, 1}); // 1행 1열의 정점 입력
    const int dr[] = {1, 0, -1, 0}, dc[] = {0, 1, 0, -1};

    while (stack.size()) {
        auto [cr, cc] = stack.top();
        visited[cr][cc] = 2; // 방문 표시

        if (cr == HEIGHT - 2 && cc == WIDTH - 2) {
            while (stack.size()) { // 시작 지점까지 돌아감
                auto [cr, cc] = stack.top();
                visited[cr][cc] = 3;
                stack.pop();
            }
            break;
        }

        int visited_flag = 0;
        for (int i = 0; i < 4; i++) {
            int nr = cr + dr[i], nc = cc + dc[i];
            if (visited[nr][nc] == 0) { // 미방문 노드 존재 시 push() 후 종료
                visited_flag = 1;
                stack.push({nr, nc});
                visited[nr][nc] = 2;
                break;
            }
        }

        if (!visited_flag) stack.pop(); // 미방문 노드가 없을 시 현재 노드 pop()
    }

    return 0;
}
```

bool ofApp::BFS()

구현한 BFS() 함수는 매우 정형화된 형태의 BFS 알고리즘이지만, 구조체 Vertex를 활용해 경로를 저장하도록 조금 변형했다. 경로를 저장하기 위해서, 시작 정점을 삽입할 때 경로를 함께 삽입할 필요가 있다. 이외의 동작은 DFS() 알고리즘과 매우 유사하지만,

(1) 자료구조가 스택이 아닌 큐라는 점,

(2) **현재 노드를 큐에서 삭제한 뒤 이동 가능한 노드를 한꺼번에 삽입한다는 점,**

(3) 목적지에 도착했을 때 경로를 계산한다는 점이 주로 다른 점이다.

시간 복잡도는 DFS와 같이 최악의 경우 접근 가능한 간선을 모두 순회하게 되므로 **O(V + E)**지만, 각 정점이 갖는 간선의 개수가 최대 4개이기 때문에 **E를 4 * V로 치환해 O(V)로 볼 수도 있다.** 공간 복잡도가 문제가 되는데, vertex 구조체는 최악의 경우 미로의 크기만큼 경로의 수를 저장하므로 V만큼의 메모리가 필요하고, 큐에 최대로 삽입될 수 있는 정점은 V개이므로 V를 곱하면 **O(V^2)의 공간 복잡도가 요구**된다.

단 실제로는 이보다 적은 공간 복잡도를 소요하게 되는데, 미로는 굉장히 sparse한 그래프이므로 구조적으로 V개만큼의 정점이 큐에 한꺼번에 삽입될 수 없기 때문이다. 실제로 1000 * 1000 그래프까지 정상적으로 BFS 길찾기가 성공함을 확인할 수 있었다.

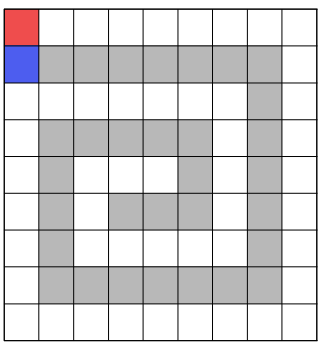
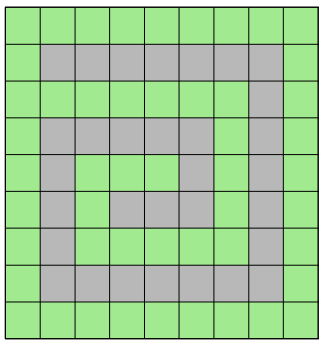
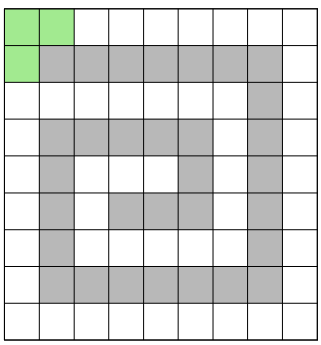
```
bool ofApp::BFS() {
    isBFS = true, isDFS = false;

    queue<Vertex> queue;
    queue.push(Vertex(1, 1)); // 1행 1열의 정점 입력
    queue.front().path.push_back({1, 1}); // 경로 입력
    visited[1][1] = 2; // 방문 표시

    const int dr[] = {0, 1, 0, -1}, dc[] = {1, 0, -1, 0};

    while (queue.size()) {
        int cr = queue.front().row;
        int cc = queue.front().col;
        vector<pair<int, int>> cp = queue.front().path;
        queue.pop();

        if (cr == HEIGHT - 2 && cc == WIDTH - 2) {
            for (pair<int, int> i : cp) {
                cr = i.first, cc = i.second;
                visited[cr][cc] = 3;
            }
            break;
        }

        for (int i = 0; i < 4; i++) {
            int nr = cr + dr[i], nc = cc + dc[i];
            if (visited[nr][nc] == 0) { // 미방문 노드 존재 시 push() 후 종료
                Vertex nv = Vertex(nr, nc);
                nv.path = cp;   nv.path.push_back({nr, nc});
                visited[nr][nc] = 2;
                queue.push(nv);
            }
        }
    }
    return 0;
}
```

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.

BFS 알고리즘은 찾은 길이 항상 최단 경로임을 보장하지만, DFS 알고리즘은 찾은 길이 항상 최단 경로라는 것을 보장하지는 못한다. 가능한 모든 경로를 확인하지는 않기 때문이다.

반면에 DFS 알고리즘은 훨씬 더 적은 정점을 탐색하기 때문에 최단 경로는 보장하지 못하더라도, 길찾기의 속도는 보통 훨씬 빠르다. 1000 * 1000 미로 등 큰 크기의 미로를 탐색할 때 DFS에서 1초 정도면 탐색이 완료되었지만, BFS에서는 훨씬 긴 시간이 걸렸다. 단 시간 차이의 원인으로는 BFS 알고리즘에서 따로 메모리 관련 최적화를 하지 않았다는 점도 있을 것이다.

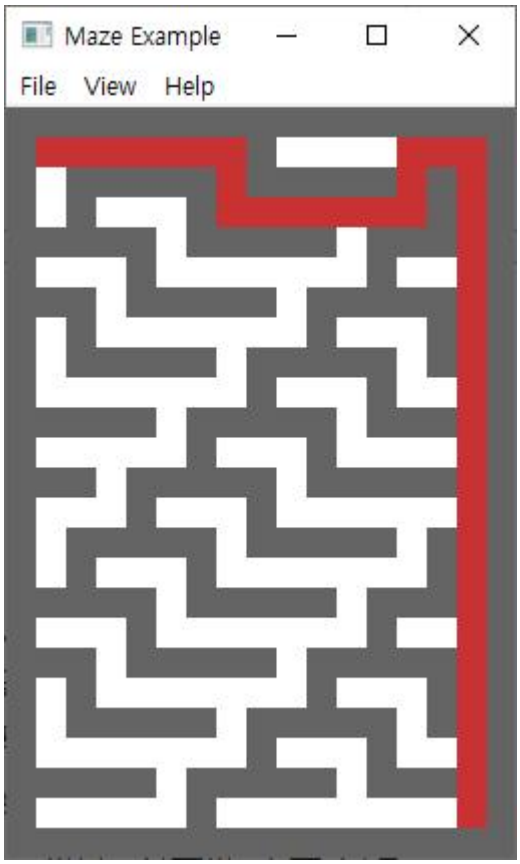
실제로 **모든 미로 탐색 문제**에 대해 DFS 알고리즘이 BFS 알고리즘에 비해 항상 우월하다고 볼 수는 없다. DFS 알고리즘이 오른쪽, 아래, 왼쪽, 위 순으로 간선을 확인한다고 가정할 때, 다음과 같은 **불완전 미로**를 예시로 들 수 있다. 빨간색이 시작점, 파란색이 도착점이라고 가정할 때, DFS 알고리즘이 확인하는 정점은 두 번째, BFS 알고리즘이 확인하는 경로는 세 번째와 같다.

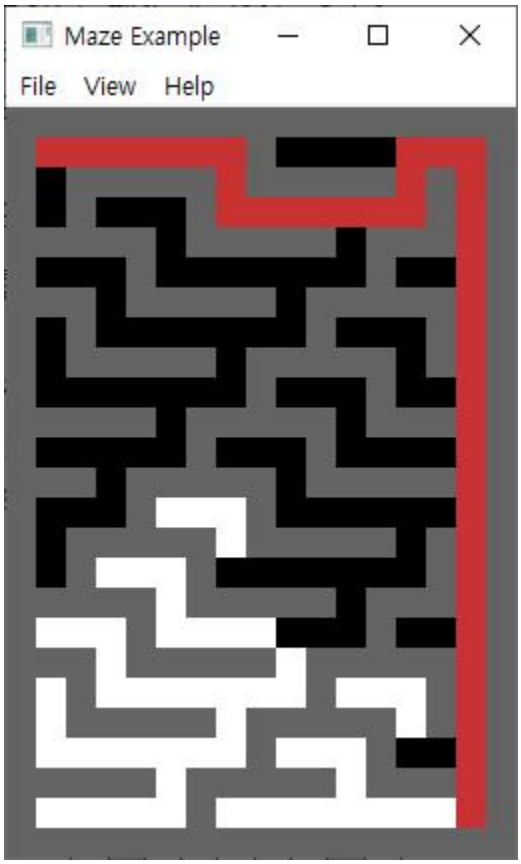




하지만 문제 상황에서는 완전 미로에서의 길 찾기를 가정하고 있으므로, DFS 길 찾기와 BFS 길 찾기는 정확히 동일한 길을 찾게 된다. 그 원리는 완전 미로의 정의에서 알 수 있는데, 두 정점 사이의 최단 경로가 오직 하나만 존재하는 미로를 완전 미로라고 정의하기 때문이다. 따라서 BFS 알고리즘이 설령 최고 수준으로 최적화되더라도, DFS 알고리즘에 비해 비효율적일 것임을 어렵지 않게 알 수 있다.

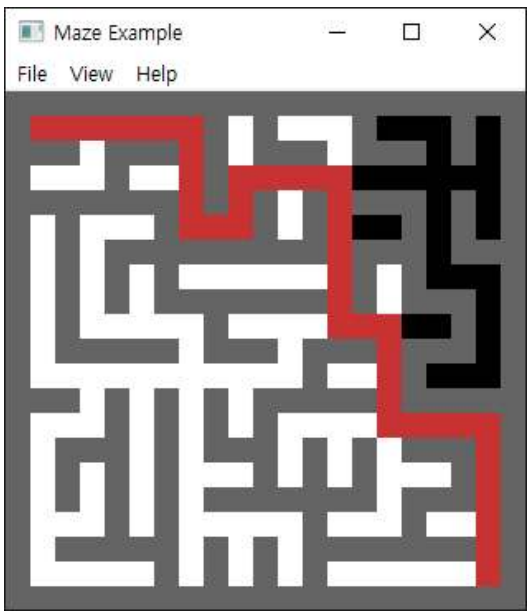
아래는 일반적인 문제 상황을 표현하기 위한 작은 미로들을 실행한 이미지와, 800 * 400 크기의 큰 미로를 실행한 이미지를 첨부했다.

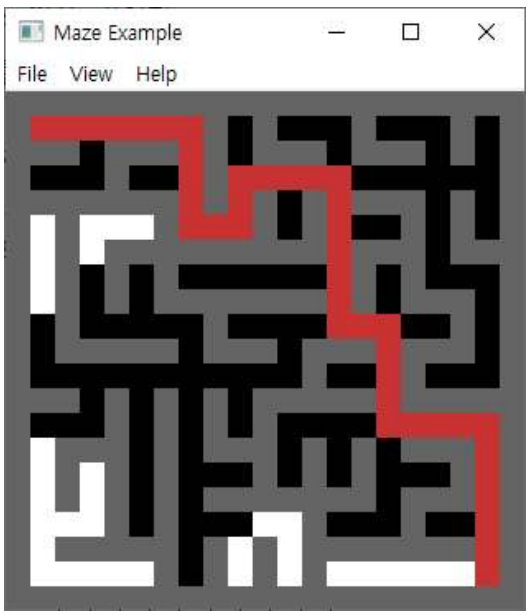
12 * 8 미로



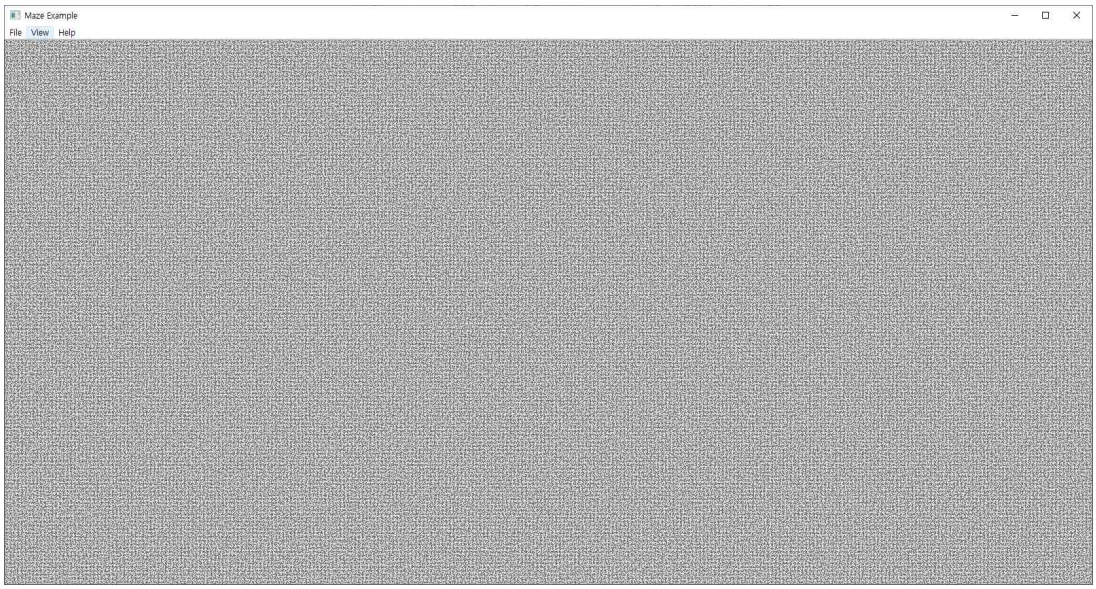


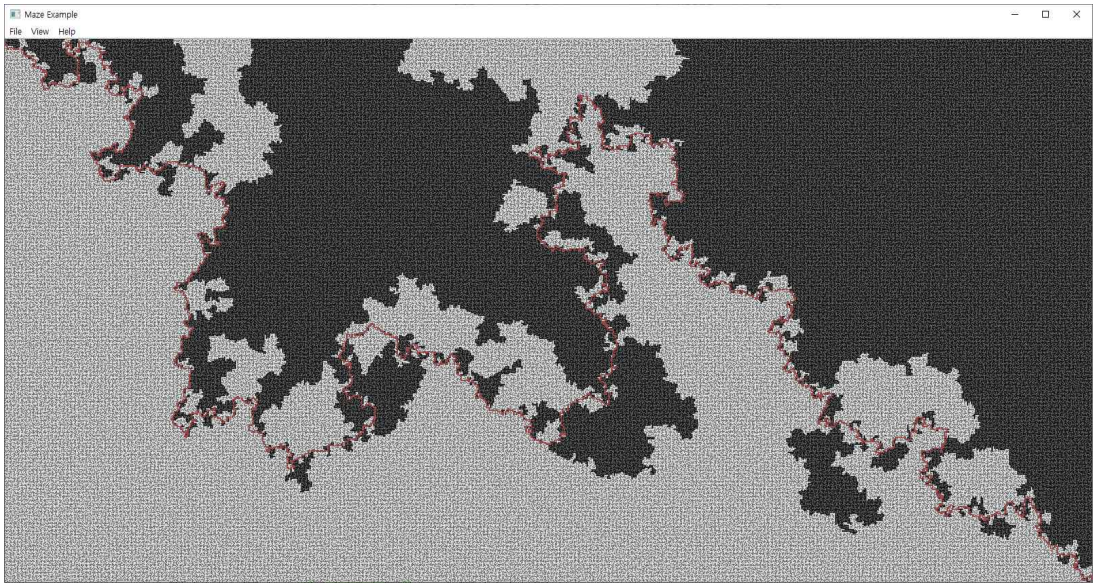
10 * 10 미로

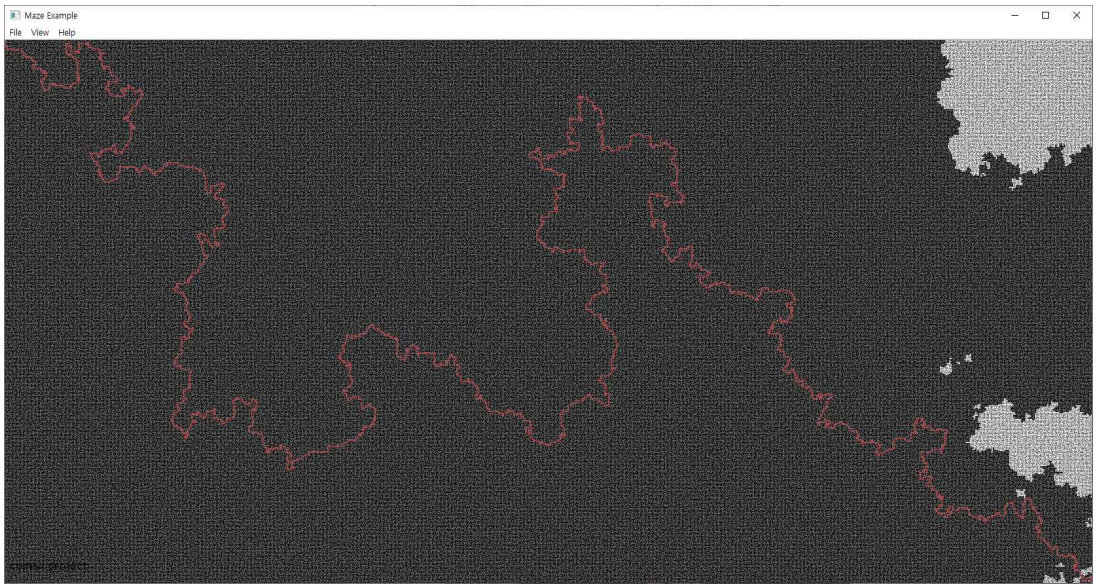




400 * 800 미로


Maze Example
File View Help


Maze Example
File View Help


Maze Example
File View Help