컴퓨터공학실험1 13주차 결과보고서

전공: 컴퓨터공학과 학년: 2 학번: 20191559 이름: 강상원

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.

`vector<vector<char>> maze;` 에 .maz 파일의 데이터를 불러왔다. 이차원 배열 (vector)에 저장된 데이터는 '+', ' ', '-' 중 하나이다. `vector<vector<bool>> visited;` 에 '+', '-'로 표현되어 있는 벽을 visited 처리해준다. (알고리즘의 단순성)

<<DFS>>

```
s_blocks.push(v);
all_path.push(v);

while(!s_blocks.empty()){
    all_path.push(s_blocks.top());

    cout << "top : " << s_blocks.top().y << ' ' << s_blocks.top().x << endl;


    if(s_blocks.top().x == realWIDTH-2 && s_blocks.top().y == realHEIGHT-2) {
        cout << "search complete!" << endl << "Press D one more time to show
diagonal line" <<endl;
        return true;
    }
    move_possible=false;
    for(int i=0; i<4; i++){
        u.y = s_blocks.top().y + y_move[i];
        u.x = s_blocks.top().x + x_move[i];


        if(!visited[u.y][u.x]){
            all_path.push(u);

            visited[ s_blocks.top().y + y_move[i] ][ s_blocks.top().x +
x_move[i]] = true;
            s_blocks.push(u);

            move_possible=true;
            break;
        }
    }
    if(!move_possible){
        s_blocks.pop();
    }
}
```

<stack>을 이용하여 스택 s_blocks를 만들고 DFS를 수행한다. s_blocks에 첫 노드를 넣고, 첫 원소 s에 대해 아직 방문하지 않은 노드(칸)를 스택에 추가하고, 스택의 첫 원소를 제거한다. 전체 과정은 스택이 비기 전까지 반복한다. 최소 경로(초록색)에 대한 정보는 s_blocks로 충분하지만, 전체 탐색 경로 (빨간색)을 추후에 표현하기 위해서 all_path라는 스택을 만들었다.

<<BFS>>

```
q_blocks.push(v);
all_path_2.push(v);

while(!q_blocks.empty()){
    path_block[q_blocks.front().y][q_blocks.front().x]=true;

    if(q_blocks.front().x == realWIDTH-2 && q_blocks.front().y == realHEIGHT-
2)
    {
        cout << "search complete!" << endl << "Press B one more time to show
diagonal line" <<endl;
        return true;
    }
    move_possible=false;
    for(int i=0; i<4; i++){
        u.y = q_blocks.front().y + y_move[i];
        u.x = q_blocks.front().x + x_move[i];
        u.count = q_blocks.front().count + 1;

        if(!visited[u.y][u.x]){
            cout << u.y << ' ' << u.x << '\t' << u.count << endl;
            visited[ q_blocks.front().y + y_move[i] ][ q_blocks.front().x +
x_move[i]] = true;
            q_blocks.push(u);
            //
            all_path_2.push(u);
            //
            path_block[ u.y ][ u.x ] = true;

            move_possible=true;
        }
    }
    if(!move_possible)
        q_blocks.pop();
}
```

<queue>를 이용하여 q_blocks를 만들고 BFS를 수행한다. q_blocks에 첫 원소를 넣고, 그 원소에 대해 인접한 노드 중 아직 방문하지 않은 노드를 모두 큐에 넣고, q_block의 첫 원소를 제거한다. 전체 과정을 큐가 빌 때까지 반복한다.

```
int x_move[4]={0, 0, 1, -1};
int y_move[4]={1, -1, 0, 0};
```

위와 같은 x_move, y_move 배열을 미리 ofApp.h에 선언하여 한 칸에서 가능한 4가지 움직임의 경우를 for문을 사용해 손쉽게 체크할 수 있도록 하였다.

실험 전에 생각한 점과 달라진 점은 ofPolyline 사용(하단 기술), BFS의 최단 경로를 화면에 나타내기 위해 BLOCK 구조체에 count라는 변수를 추가한 점이다.

미로를 그리는 과정에 있어서는

```
ofPolyline path;
ofPolyline allpath;
```

를 사용하였는데, ofPolyline은 입력된 점들의 좌표값을 잇는 직선들을 연속해서 그려준다.
이는 다음과 같이 표현 가능하다.

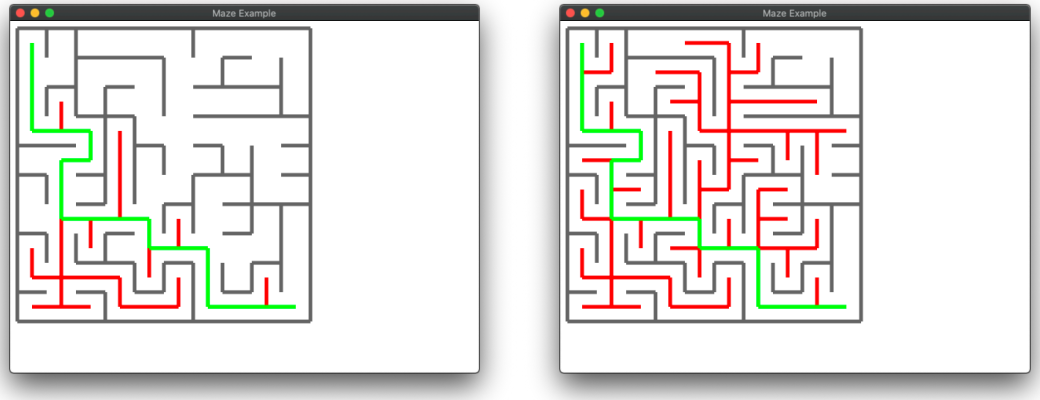
```
ofSetColor(255, 0, 0);
int iterateall=all_path.size();

for(int i=0; i<iterateall; i++){
    allpath.addVertex(20*all_path.top().x+10, 20*all_path.top().y+10);
    all_path.pop();
}
allpath.draw();


ofSetColor(0, 255, 0);
int iterate=s_blocks.size();

for(int i=0; i<iterate; i++){
    path.addVertex(20*s_blocks.top().x+10, 20*s_blocks.top().y+10);
    s_blocks.pop();
}
path.draw();
```

freeMemory는 선언된 모든 스택, 큐, 구조체 배열 등의 메모리를 해제하는 방향으로 하였다.

```
void ofApp::freeMemory(){
    this->maze.clear();
    vector<vector<char>>().swap(this->maze);

    this->LINES.clear();
    vector<MazeLines>().swap(this->LINES);

    this->PATHS.clear();
    vector<MazeLines>().swap(this->PATHS);

    while(!s_blocks.empty())
        s_blocks.pop();

    while(!s_blocks_2.empty())
        s_blocks_2.pop();

    while(!s_path.empty())
        s_path.pop();
    while(!all_path.empty())
        all_path.pop();

    while(!q_blocks.empty())
        q_blocks.pop();
    while(!q_path.empty())
        q_path.pop();
    while(!all_path_2.empty())
        all_path_2.pop();
    while(!all_path_3.empty())
        all_path_3.pop();

    this->draw_flag=0;
    this->dfs_flag=0;

    path.clear();
    allpath.clear();
    this->dfs_flag=0;
    this->bfs_flag=0;
}
```

시간복잡도는 DFS, BFS 모두 O(n^2)로 동일하며, 공간복잡도도 O(WIDTH*HEIGHT)가 되겠다.

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.



같은 미로에 대하여 왼쪽이 DFS, 오른쪽이 BFS를 수행한 결과이다. DFS 알고리즘이 탐색한 전체 경로가 BFS보다 상대적으로 적음을 알 수 있다. 이는 이 미로에만 국한되는 것이 아니라 다른 테스트 케이스에서도 동일한 양상을 보인다. 아마도 Eller's Algorithm으로 만든 완전 미로에서는 이런 결과가 나타나는 것 같다. 하지만 불완전 미로이거나, 미로의 규칙성이 기존 테스트 케이스와 달라진다면 적합성에 차이가 있을 수 있다. BFS는 최단 경로를 보장한다는 장점이 있다. 하지만 경로가 1가지인 경우에는 별다른 장점이 되지 못한다.

어느덧 마지막 결과레포트네요.
사이버 강의라 조교님들이 중간에서 많이 힘드셨을 것 같습니다 ㅠㅠ
조교님 한 학기동안 정말 수고많으셨습니다.
아직 프로젝트와 기말 시험이 남았으나 남은 일정 최선을 다해보겠습니다..
다시 한번 감사드립니다.