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.

미로를 그래프로 생각하고 구현했을 때 사용할 Vertex 구조체를 선언했다. visited는 DFS, BFS에서 해당 vertex를 방문했는지를 marking하기 위한 변수이며, parent는 해당 vertex가 어느 위치에서 왔는지를 기록하기 위한 변수이다.

```
enum Parent{
    UP, DOWN, LEFT, RIGHT, NONE
};

struct Vertex{
    /* Can or cannot go ... */
    bool up;
    bool down;
    bool left;
    bool right;
    /* Position of this vertex */
    int x;
    int y;
    /* This vertex is visited or not */
    bool visited;
    /* Parent position */
    Parent parent;
};
```

파일로부터 읽어들인 2차원 char 배열은 DFS, BFS에 불필요한 정보를 담고 있기 때문에 미로에서 이동할 수 있는 칸만을 저장하기 위하여 Vertex 구조체의 2차원 배열 maze를 선언했다.

```
vector<vector<char>> image;     /* Saves all information from the text file. */
vector<Line> walls;            /* Saves wall information */
vector<vector<Vertex>> maze;    /* Saves maze cell information */
```

maze를 초기화하고 상, 하, 좌, 우로 이동할 수 있는지의 여부를 저장하는 함수를 작성했다.

```
void ofApp::processImage(){
    int i, j, x, y;
    Vertex tmp;

    /* Initialize maze */
    for (y = 0; y < mazeHEIGHT; y++) {
        maze.push_back(vector<Vertex>());
        for (x = 0; x < mazeWIDTH; x++) {
            tmp.visited = false;
            tmp.x = x;
            tmp.y = y;
            tmp.parent = NONE;
            maze[y].push_back(tmp);
        }
    }

    /* From image, make maze. */
    /* 홀수 인덱스마다 칸이 있음 */
    for (i = 1; i < imageHEIGHT; i += 2) {
```

```
        y = (i - 1) / 2;
        for (j = 1; j < imageWIDTH; j += 2) {
            x = (j - 1) / 2;
            /* up */
            maze[y][x].up = (image[i-1][j] == ' ') ? true : false;
            /* down */
            maze[y][x].down = (image[i+1][j] == ' ') ? true : false;
            /* left */
            maze[y][x].left = (image[i][j-1] == ' ') ? true : false;
            /* right */
            maze[y][x].right = (image[i][j+1] == ' ') ? true : false;
        }
    }

}
```

DFS는 다음과 같이 구현했다. 예비보고서에 작성한 대로 먼저 모든 vertex를 방문하지 않은 vertex로 표시하고, 시작 위치의 vertex를 스택에 push했다. 그리고 stack이 비어있지 않은 동안 stack의 top 원소에서 상, 하, 좌, 우에 아직 방문하지 않은 vertex가 있다면 해당 vertex를 스택에 push했다. 예비보고서에는 무조건 stack에서 pop하고 이웃하는 vertex를 조사했지만, 실습에서 DFS를 구현할 때는 이웃하는 vertex 중 방문하지 않은 vertex가 없다면 pop하도록 했다. stack의 top 원소를 확인할 때마다 해당 path를 그려줄 수 있도록 path를 저장해주는 getSearchedPath() 함수를 호출했고, 만약 목적지에 도달했다면 getShortestPath() 함수를 호출해 탈출 경로 path를 저장해주었다.

```
bool ofApp::DFS(){
    /* Initialize path */
    searched_path.clear();
    shortest_path.clear();

    /* 모든 노드를 방문하지 않은 노드로 표시 */
    int i, j;
    for (i = 0; i < mazeHEIGHT; i++) {
        for (j = 0; j < mazeWIDTH; j++) {
            maze[i][j].visited = false;
            maze[i][j].parent = NONE;
        }
    }

    Vertex* curr;
    i = 0; j = 0;
    curr = &maze[i][j];
    curr->visited = true; //start_node.visited = 1
    DFS_stack.push(curr); //stack.append(start_node)

    bool has_unvisited;
    /* while (stack is not empty) */
    while (!DFS_stack.empty()) {

        curr = DFS_stack.top();

        getSearchedPath(curr);

        /* Destination reached */
        if ((curr->x == mazeWIDTH - 1) && (curr->y == mazeHEIGHT - 1)) {


            getShortestPath();

            return true;
        }

        //if (S.top has an unvisited adjacent node)
        has_unvisited = false;
```

```
        i = curr->y; j = curr->x;
        if (curr->up && !maze[i-1][j].visited) {
            maze[i-1][j].visited = true;
            maze[i-1][j].parent = DOWN;
            DFS_stack.push(&maze[i-1][j]);
            has_unvisited = true;
        }
        if (curr->down && !maze[i+1][j].visited) {
            maze[i+1][j].visited = true;
            maze[i+1][j].parent = UP;
            DFS_stack.push(&maze[i+1][j]);
            has_unvisited = true;
        }
        if (curr->left && !maze[i][j-1].visited) {
            maze[i][j-1].visited = true;
            maze[i][j-1].parent = RIGHT;
            DFS_stack.push(&maze[i][j-1]);
            has_unvisited = true;
        }
        if (curr->right && !maze[i][j+1].visited) {
            maze[i][j+1].visited = true;
            maze[i][j+1].parent = LEFT;
            DFS_stack.push(&maze[i][j+1]);
            has_unvisited = true;
        }

        /* 더 이상 갈 곳이 없음 */
        if (!has_unvisited) {
            DFS_stack.pop();
        }

    }

    return false;
}
```

BFS 함수는 다음과 같이 작성했다. 예비보고서에 작성한 대로 알고리즘을 구현했다. 먼저 모든 vertex를 방문하지 않은 vertex로 표시하고, 시작 위치의 vertex를 queue에 push했다. 그리고 queue가 비어있지 않은 동안 queue의 front 원소를 pop하고, 그 원소에서 상, 하, 좌, 우에 아직 방문하지 않은 vertex가 있다면 해당 vertex를 queue에 push했다. BFS에서 확인하는 path를 저장해주기 위해 front 원소를 확인할 때 getSearchedPath() 함수를 호출해주었고, 목적지에 도달했을 때는 탈출 경로 path를 저장하기 위해 getShortestPath() 함수를 호출해주었다.

```
bool ofApp::BFS(){
    /* Initialize path */
    searched_path.clear();
    shortest_path.clear();

    /* 모든 노드를 방문하지 않은 노드로 표시 */
    int i, j;
    for (i = 0; i < mazeHEIGHT; i++) {
        for (j = 0; j < mazeWIDTH; j++) {
            maze[i][j].visited = false;
            maze[i][j].parent = NONE;
        }
    }

    Vertex* curr;
    i = 0; j = 0;
    curr = &maze[i][j];
    curr->visited = true; //start_node.visited = True
    BFS_queue.push(curr); //queue.append(start_node)

    /* while (queue is not empty) */
    while (!BFS_queue.empty()) {
```

```
        //curr_node = queue.popleft()
        curr = BFS_queue.front();
        getSearchedPath(curr);
        BFS_queue.pop();

        /* Destination reached */
        if ((curr->x == mazeWIDTH - 1) && (curr->y == mazeHEIGHT - 1)) {

//          cout << "BFS success!!!" << endl;

            getShortestPath();

            return true;
        }

        i = curr->y; j = curr->x;
        if (curr->up && !maze[i-1][j].visited) {
            maze[i-1][j].visited = true;
            maze[i-1][j].parent = DOWN;
            BFS_queue.push(&maze[i-1][j]);
        }
        if (curr->down && !maze[i+1][j].visited) {
            maze[i+1][j].visited = true;
            maze[i+1][j].parent = UP;
            BFS_queue.push(&maze[i+1][j]);
        }
        if (curr->left && !maze[i][j-1].visited) {
            maze[i][j-1].visited = true;
            maze[i][j-1].parent = RIGHT;
            BFS_queue.push(&maze[i][j-1]);
        }
        if (curr->right && !maze[i][j+1].visited) {
            maze[i][j+1].visited = true;
            maze[i][j+1].parent = LEFT;
            BFS_queue.push(&maze[i][j+1]);
        }

    }

    return false;
}
```

getSearchedPath() 함수는 전달받은 vertex에 대해, 어느 방향에서 왔는지에 따라 선분을 만들어 searched_path에 push한다. searched_path는 DFS 또는 BFS가 조사한 path를 저장하는 변수이다.

```
void ofApp::getSearchedPath(Vertex* curr){

    Line tmp;

    switch (curr->parent) {
        case UP:
            tmp.startX = 40 * curr->x + 30;
            tmp.startY = 40 * (curr->y - 1) + 80;
            tmp.endX = 40 * curr->x + 30;
            tmp.endY = 40 * curr->y + 80;
            searched_path.push_back(tmp);
            break;
        case DOWN:
            tmp.startX = 40 * curr->x + 30;
            tmp.startY = 40 * (curr->y + 1) + 80;
            tmp.endX = 40 * curr->x + 30;
            tmp.endY = 40 * curr->y + 80;
            searched_path.push_back(tmp);
            break;
        case LEFT:
            tmp.startX = 40 * (curr->x - 1) + 30;
            tmp.startY = 40 * curr->y + 80;
            tmp.endX = 40 * curr->x + 30;
            tmp.endY = 40 * curr->y + 80;
```

```
            searched_path.push_back(tmp);
            break;
        case RIGHT:
            tmp.startX = 40 * (curr->x + 1) + 30;
            tmp.startY = 40 * curr->y + 80;
            tmp.endX = 40 * curr->x + 30;
            tmp.endY = 40 * curr->y + 80;
            searched_path.push_back(tmp);
            break;
        default:
            ;
    }

}
```

getShortestPath() 목적지부터 parent를 쫓아가면서 path를 만들어서 shortest_path에 push한다. shortest_path는 DFS 또는 BFS를 통해 알아낸 탈출 경로 path를 저장하는 변수이다.

```
void ofApp::getShortestPath(){

    Vertex* curr;
    Line tmp;
    int i = mazeHEIGHT-1;
    int j = mazeWIDTH-1;

    while (!(i == 0 && j == 0)) {
        curr = &maze[i][j];

        /* 부모 노드로 올라가기 */
        switch(curr->parent) {
            case UP:
                tmp.startX = 40 * curr->x + 30;
                tmp.startY = 40 * curr->y + 80;
                tmp.endX = 40 * curr->x + 30;
                tmp.endY = 40 * (curr->y - 1) + 80;
                shortest_path.push_back(tmp);
                i--;
                break;

            case DOWN:
                tmp.startX = 40 * curr->x + 30;
                tmp.startY = 40 * curr->y + 80;
                tmp.endX = 40 * curr->x + 30;
                tmp.endY = 40 * (curr->y + 1) + 80;
                shortest_path.push_back(tmp);
                i++;
                break;

            case LEFT:
                tmp.startX = 40 * curr->x + 30;
                tmp.startY = 40 * curr->y + 80;
                tmp.endX = 40 * (curr->x - 1) + 30;
                tmp.endY = 40 * curr->y + 80;
                shortest_path.push_back(tmp);
                j--;
                break;

            case RIGHT:
                tmp.startX = 40 * curr->x + 30;
                tmp.startY = 40 * curr->y + 80;
                tmp.endX = 40 * (curr->x + 1) + 30;
                tmp.endY = 40 * curr->y + 80;
                shortest_path.push_back(tmp);
                j++;
                break;

            default:
                ;
        }
    }
```

```
}
```

getSearchedPath(), getShortestPath() 함수는 DFS와 BFS 두 모드에서 모두 사용 가능하다. 두 함수에서 저장한 path를 바탕으로 모드에 따라 draw() 함수에서 drawDFS(), drawBFS()를 호출하고 이는 다음과 같이 저장된 path를 빨간색, 초록색으로 그려주는 것으로 나타냈다.

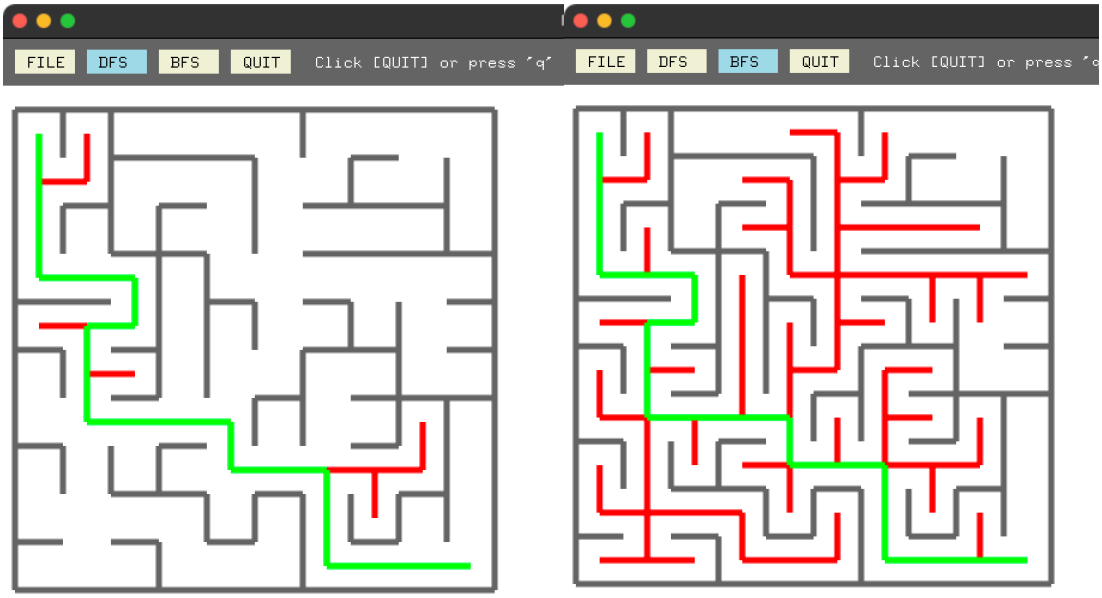
```
//--------------------------------------------------------------
void ofApp::drawDFS(){
    int i;

    /* searched path */
    ofSetColor(ofColor::red);
    for (i = 0; i < searched_path.size(); i++)
        ofDrawLine(searched_path[i].startX, searched_path[i].startY, searched_path[i].endX,
searched_path[i].endY);

    /* shortest path */
    ofSetColor(ofColor::green);
    for (i = 0; i < shortest_path.size(); i++)
        ofDrawLine(shortest_path[i].startX, shortest_path[i].startY, shortest_path[i].endX,
shortest_path[i].endY);
}

//--------------------------------------------------------------
void ofApp::drawBFS(){
    int i;

    /* searched path */
    ofSetColor(ofColor::red);
    for (i = 0; i < searched_path.size(); i++)
        ofDrawLine(searched_path[i].startX, searched_path[i].startY, searched_path[i].endX,
searched_path[i].endY);

    /* shortest path */
    ofSetColor(ofColor::green);
    for (i = 0; i < shortest_path.size(); i++)
        ofDrawLine(shortest_path[i].startX, shortest_path[i].startY, shortest_path[i].endX,
shortest_path[i].endY);
}
```

구현한 알고리즘의 시간복잡도는 예비보고서에서 예상한 바와 같이 O(HEIGHT * WIDTH)이고, 공간복잡도 역시 O(HEIGHT * WIDTH)이다.

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.

DFS, BFS를 실행해 본 결과, 모두 같은 path를 찾아냈다. 한편 DFS가 BFS보다 다른 path를 탐색하는 횟수가 적어 DFS가 더 적합할 것으로 예상된다. Eller's algorithm으로 생성된 완전 미로가 아닌 불완전 미로나 여러가지 탈출 경로가 존재한다면, 실험 결과가 달라질 수 있다.

- 예시 1) DFS(왼쪽), BFS(오른쪽)



- 예시 2) DFS(왼쪽), BFS(오른쪽)