13주차 결과보고서

전공: 컴퓨터공학과 학년: 2학년 학번: 20231515 이름: 김다은

**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.**

**1-1. 실습(DFS)**

**1-1-1. 작성한 프로그램의 자료구조**

*// 내가 추가*

    vector<string> inputmaze;

*// 미로에서 움직일 수 있는 방향*

    int maxdir = 4;

    int dx[4] = { 0, 1, 0, -1 };

    int dy[4] = { 1, 0, -1, 0 };

    stack<tuple<int, int, short>> st;

    vector<pair<int, int>> anspath;

    vector<pair<int, int>> path;

정답 경로를 저장하기 위해 anspath vector를 선언하였다. 탐색한 모든 경로에 path vector를 선언하였다.

**1-1-2. 작성한 프로그램의 알고리즘**

먼저 필요한 변수를 선언하고 초기화를 진행한다.

*//DFS탐색을 하는 함수 ( 3주차)*

    int x, y;

*// 미로 목적지 좌표*

    int destx = inputmaze.size() - 2, desty = inputmaze[0].size() - 2;

*// printf("%d %d\n\n", destx, desty);*

    int nx, ny, dir;

    bool success = 0; *// 목적지 도착 여부*

*// stack 초기화*

    while (!st.empty()) st.pop();

*// anspath, path 초기화*

    path.clear();

    anspath.clear();

    vector<pair<int, int>>().swap(path);

    vector<pair<int, int>>().swap(anspath);

*// maze 크기에 맞추어 visited 배열 동적할당*

    visited = new bool\* [inputmaze.size()];

    for (int i = 0; i < inputmaze.size(); i++) {

        visited[i] = new bool[inputmaze[0].size()];

        for (int j = 0; j < inputmaze[0].size(); j++) {

            visited[i][j] = 0; *// 0으로 초기화*

        }

    }

이후 while문을 이용해 경로를 탐색한다. 아래 코드의 작동 방식은 대략 다음과 같다.

1. 출발지를 stack에 push한 뒤, 방문처리
2. stack.top()의 x, y, dir정보를 저장한다.
3. 저장한 x, y좌표에 방향에 따라 좌표를 더해 nx, ny를 구해 이동 가능한 칸을 탐색한다.  
   (벽으로 막히지 않았고, 미로 범위 안에 nx, ny가 존재하고, 아직 방문하지 않은 칸인 경우 이동가능하다.)
   1. 만약 이동가능한 칸 이라면, stack에 해당 칸의 위치를 push한 뒤, 방문 처리한다. 만약 해당 칸이 목적지라면 success를 1로 설정하여 while문을 탈출한다.
4. 2, 3번 과정을 반복한다.

*// 출발지점*

    st.push({ 1, 1, 0 });

    visited[1][1] = 1; *// 방문 처리*

    while (!st.empty()) {

        x = get<0>(st.top());

        y = get<1>(st.top());

        dir = get<2>(st.top());

*// st.pop();*

*// printf("%d %d %d\n", x, y, dir);*

        path.push\_back({ x, y });

*// 벽에 막힌 경로인지 확인*

        bool flag = 0;

        for (; dir < maxdir; dir++) {

            nx = x + dx[dir];

            ny = y + dy[dir];

            if (inputmaze[nx][ny] == '|' || inputmaze[nx][ny] == '-') {

                continue;

            }

            else {

                nx += dx[dir]; ny += dy[dir];

                if (!visited[nx][ny] && 0 <= nx && nx < inputmaze.size() && 0 <= ny && ny < inputmaze[0].size() && dir < maxdir) {

                    st.push({ nx, ny, 0 });

                    visited[nx][ny] = 1; *// 방문처리*

                    flag = 1;

                    if (nx == destx && ny == desty) {

                        success = 1;

                        break;

                    }

                    break;

                }

            }

        }

        if (!flag) {

            st.pop();

        }

        if (success) break;

    }

이후 visited에 동적 할당된 메모리를 해제한 뒤, 정답 경로를 anspath에 저장한다.

*// visited 동적 할당 메모리 해제*

    for (int i = 0; i < inputmaze.size(); i++) {

        delete[] visited[i];

    }

    delete[] visited;

    while (!st.empty()) {

        anspath.push\_back({ get<0>(st.top()), get<1>(st.top()) });

        st.pop();

    }

    if (success) return true;

    return false;

**1-2. 숙제(BFS)**

**1-1-1. 작성한 프로그램의 자료구조**

*// 내가 추가*

    vector<string> inputmaze;

*// 미로에서 움직일 수 있는 방향*

    int maxdir = 4;

    int dx[4] = { 0, 1, 0, -1 };

    int dy[4] = { 1, 0, -1, 0 };

vector<tuple<int, int, short>> bfs\_path;

탐색한 모든 경로를 저장하기 위해 vector bfs\_path를 선언하였다. 또한 BFS 함수 안에서 아래와 같은 queue를 선언하여 사용하였다.

queue<pair<int,int>> q;

**1-1-2. 작성한 프로그램의 알고리즘**

먼저 필요한 변수를 선언하고 초기화한다.

queue<pair<int,int>> q;

    int x, y;

*// 미로 목적지 좌표*

    int destx = inputmaze.size() - 2, desty = inputmaze[0].size() - 2;

*// printf("%d %d\n\n", destx, desty);*

    int nx, ny;

    bool success = 0; *// 목적지 도착 여부*

*// queue 초기화*

*// while (!q.empty()) q.pop();*

*// bfs\_path 초기화*

    bfs\_path.clear();

    vector<tuple<int, int, short>>().swap(bfs\_path);

*// maze 크기에 맞추어 visited 배열 동적할당*

    visited = new bool\* [inputmaze.size()];

    for (int i = 0; i < inputmaze.size(); i++) {

        visited[i] = new bool[inputmaze[0].size()];

        for (int j = 0; j < inputmaze[0].size(); j++) {

            visited[i][j] = 0; *// 0으로 초기화*

        }

    }

이후 while문을 이용해 미로의 경로를 찾는다. 대략적인 코드의 흐름은 아래와 같다.

1. 출발 지점의 좌표 (1, 1)을 queue에 push하고 방문 처리한다.
2. queue.front()의 좌표를 변수 x, y에 저장한 뒤, queue.pop()을 해준다.
3. x, y에서 이동가능한 4가지 방향에 대해 for문으로 이동가능한 칸이 있는지 확인한다.
   1. 만약 미로의 범위 안에 칸이 존재하고, 벽에 막히지 않았고, 아직 방문하지 않은 칸 이라면, queue에 해당 칸의 좌표를 push하고 방문 처리한다.
   2. 만약 해당 칸이 목적지의 좌표와 일치한다면 success를 1로 설정하여 for문과 while문을 탈출한다.
4. 2, 3번 과정을 반복한다.

*// 출발지점*

q.push({ 1, 1 });

visited[1][1] = 1; *// 방문 처리*

bfs\_path.push\_back({ 1, 1, -1 });

while (!q.empty()) {

    x = q.front().first; y = q.front().second; q.pop();

    for (int k = 0; k < maxdir; k++) {

        nx = x + dx[k];

        ny = y + dy[k];

        if ((!(0 <= nx && nx < inputmaze.size() && 0 <= ny && ny < inputmaze[0].size()))

            || (inputmaze[nx][ny] == '|' || inputmaze[nx][ny] == '-')) {

            continue;

        }

        else {

            nx += dx[k]; ny += dy[k];

            if (!visited[nx][ny]) {

                q.push({ nx, ny });

                visited[nx][ny] = 1;

                bfs\_path.push\_back({ nx, ny, k });

*// 목적지에 도달한 경우*

                if (nx == destx && ny == desty) {

                    success = 1;

                    break;

                }

            }

        }

    }

*// 경로를 찾은 경우 break*

    if (success) break;

}

마지막으로 visited 변수에 동적 할당된 메모리를 해제하고, success가 1이라면 true를 리턴한다. 만약 1이 아니라면 false를 리턴한다.

*// visited 동적 할당 메모리 해제*

    for (int i = 0; i < inputmaze.size(); i++) {

        delete[] visited[i];

    }

    delete[] visited;

    if (success) return true;

    return false;

**1-3. 시간복잡도와 공간복잡도**

실습에서 작성한 DFS, 숙제로 작성한 BFS 알고리즘 모두, 미로의 모든 칸을 탐색하게 되는 경우가 최악의 경우이다. 따라서 미로의 가로, 미로의 세로 길이를 각각 COL, ROW라고 표현할 때, 시간복잡도는 O(ROW\*COL)로 나타난다.

또한 미로 정보를 저장하고 목적지까지의 경로를 탐색하기 위해 inputmaze vector가 사용된다. inputmaze는 ROW\*COL 크기를 가진다. 따라서 공간복잡도는 대략 O(ROW\*COL)로 나타난다.

실험 전에 생각한 방법과 동일하게 구현하였다.

**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**.

아래는 동일한 미로를 DFS, BFS 알고리즘으로 경로를 탐색한 결과이다.

|  |  |
| --- | --- |
| DFS | BFS |
| 패턴, 직사각형, 디자인이(가) 표시된 사진  자동 생성된 설명 |  |

BFS는 DFS와 비교했을 때, 목적지 노드까지의 깊이만을 탐색하게 된다. 따라서 보통 최적의 해를 찾는 알고리즘에서는 BFS가 적합한 경우가 많다.

다만 이번 미로 실습의 경우는 좀 달랐다. 미로의 경우 출발지와 목적지가 정해져 있으며, 이는 목적지까지의 깊이가 정해져 있다는 뜻이다. 따라서 다른 경우에 비해 BFS가 강점을 드러내지 못한다.

또한 미로의 경우 확률적으로 아래, 오른쪽으로 이동할 경우 목적지에 빠르게 도달할 가능성이 높다. 따라서 DFS에서 아래, 오른쪽을 위, 왼쪽보다 먼저 탐색하도록 함으로서 빠르게 최적의 경로를 찾을 수 있었다. 따라서 미로 프로그램에서는 DFS가 더 적합하다고 생각된다.