13주차 결과보고서

전공: 컴퓨터공학과 학년: 2학년 학번: 20221532 이름: 김민기

1.

1) DFS 함수

bool flag = false;

int x = 0, y = 0, width = WIDTH/2, height = HEIGHT/2;

visited.resize(height, vector<int>(width, 0));

visited[y][x] = 1;

S.push(y \* width + x);

bestLine.push\_back(make\_pair(0, 0));

while (!S.empty()) {

int top = S.top();

int i = top / width;

int j = top % width;

y = i \* 2 + 1;

x = j \* 2 + 1;

if (top == (height - 1) \* width + width - 1) {

printf("finished\n");

isdfs = true;

tmpFlag = false;

return true;

}

if (상하좌우 이동할 수 있는지 판단) {

S.push((i + 1) \* width + j);

bestLine.push\_back(make\_pair(i+1, j));

visited[i + 1][j] = 1;

flag = true;

}

if (flag) {

flag = false;

}

else {

bestLine.pop\_back();

S.pop();

}

}

return false;

방문했는지 표시하기 위한 배열 visited는 미로의 HEIGHT\*WIDTH 크기의 2차원배열로 선언했다. 최적경로를 저장하기 위한 bestLine은 경로 좌표들을 저장한 벡터로 선언했다. Stack 자료형인 S를 사용하여 DFS 탐색을 구현하였다.

시간 복잡도는 while 루프는 스택이 비어 있지 않은 동안 계속 반복하므로, 스택의 크기에 비례하는 N번의 반복이 수행된다. 즉 최대 O(HEIGHT\*WIDTH)이다. 공간 복잡도는 visited에 의해 결정되므로 O(HEIGHT\*WIDTH)이다.

실험 전의 방법과 달라진 점은 윈도우에 그릴 좌표를 계산하기 위해 x와 y를 만들어 저장했다는 점과 bestLine을 pair를 사용해 좌표로 저장한 점이 달라졌다.

2) dfsdraw 함수

fSetColor(0, 255, 0);

for (int i = 0; i < (HEIGHT / 2) - 1; i++) {

for (int j = 0; j < (WIDTH / 2) - 1; j++) {

if (visited[i][j] == 1 && visited[i][j + 1] == 1 && maze[i\*2+1][j\*2+2] != '|')

ofDrawLine(j \* 30 + 15, i \* 30 + 15, j \* 30 + 45, i \* 30 + 15);

if (visited[i][j] == 1 && visited[i+1][j] == 1 && maze[i \* 2 + 2][j \* 2 + 1] != '-')

ofDrawLine(j \* 30 + 15, i \* 30 + 15, j \* 30 + 15, i \* 30 + 45);

}

}

for (int j = 0; j < (WIDTH / 2) - 1; j++) {

if (visited[HEIGHT / 2 - 1][j] == 1 && visited[HEIGHT / 2 - 1][j + 1] == 1 && maze[HEIGHT - 2][j \* 2 + 2] != '|') {

ofDrawLine(j \* 30 + 15, (HEIGHT / 2 - 1) \* 30 + 15, j \* 30 + 45, (HEIGHT / 2 - 1) \* 30 + 15);

}

}

tmpFlag = true;

int size = bestLine.size();

ofSetColor(255, 0, 0);

for (int i = 0; i < size - 1; i++) {

ofDrawLine(bestLine[i].second \* 30 + 15, bestLine[i].first \* 30 + 15, bestLine[i+1].second \* 30 + 15, bestLine[i + 1].first \* 30 + 15);

}

Visited를 기준으로 선을 그린다. 첫번째 이중 for 문에서는 방문한 모든 방들을 초록색 선으로 이어준다. 이때 maze 텍스트파일 상에서 벽이 존재한다면 선을 잇지 않는다. 마지막 가로줄은 이중 for 문에서 그려지지 않으므로 for 문으로 추가적으로 그려준다. 이후 색을 빨간색으로 변경하여 최적 경로를 bestLine을 기준으로 그려준다.

추가적인 자료저장은 없으며 draw함수에서 30을 기준으로 미로를 그렸기에 30을 기준으로 선을 그려주었다.

시간 복잡도는 O(HEIGHT\*WIDTH)이며 공간복잡도는 O(1)이다.

3) BFS함수

int x = 0, y = 0, width = WIDTH / 2, height = HEIGHT / 2;

vector< vector< pair<int, int> > > prev(HEIGHT, vector< pair<int, int> >(WIDTH, make\_pair(-1, -1)));

visited.resize(height, vector<int>(width, 0));

bestLine.push\_back(make\_pair(0, 0));

visited[y][x] = 1;

Q.push(y \* WIDTH + x);

while (!Q.empty()) {

int front = Q.front();

Q.pop();

int i = front / width;

int j = front % width;

int curY = i \* 2 + 1;

int curX = j \* 2 + 1;

if (front == (height - 1) \* width + width - 1) {

int prevY = height - 1;

int prevX = width - 1;

while (prevY != -1 && prevX != -1) {

bestLine.push\_back(make\_pair(prevY, prevX));

int tempY = prev[prevY][prevX].first;

int tempX = prev[prevY][prevX].second;

prevY = tempY;

prevX = tempX;

}

reverse(bestLine.begin(), bestLine.end());

bestLine.pop\_back();

isdfs = true;

return true;

}

if (상하좌우 이동할 수 있는지 판단){

Q.push((i + 1) \* width + j);

visited[i + 1][j] = 1;

prev[i + 1][j] = make\_pair(i, j);

}

}

return false;

dfs와 동일하게 visited와 bestLine을 사용한다. 추가적으로 필요한 자료구조는 dfs와 다르게 탐색을 하며 최적경로를 찾는 것이 힘들기에 target점에 도착하면 역추적으로 최적 경로를 찾기 위해서 prev라는 이차원 벡터를 선언했다. Prev의 각 좌표에는 이전 좌표의 값이 저장된다. 이를 통해 역추적하여 bestLine에 저장할 수 있다.

시간, 공간 복잡도는 최악의 경우 DFS와 동일하다.

실험 전의 예상과 달라진 점은 최적 경로를 탐색하는 것이 BFS알고리즘과 동시에 진행될 수 있을 것이라 예상했으나 불가능하여 역추적으로 구현했다는 점이다.

2.

A maze with a red line

Description automatically generated with low confidence A maze with a red line

Description automatically generated with low confidence

- DFS - BFS

위의 두 사진은 DFS와 BFS로 실행한 결과이다. DFS알고리즘에서는 아래와 오른쪽으로 진행하는 것을 더 우선순위에 두었기에 빠르게 target에 도착한 것을 알 수 있었다. 하지만 BFS 알고리즘은 모든 방에 가깝게 탐색한 것을 알 수 있었다. 현재 실험에서 진행 하는 미로는 왼쪽 위의 끝점에서 시작하여 오른쪽 아래 끝점으로 도착해야 하기에 같은 레벨의 모든 방들을 탐색하는 BFS는 적합하지 않은 알고리즘이라 할 수 있기에 깊게 탐색하는 DFS알고리즘이 더 적합하다 볼 수 있다. 만약 도착점이 랜덤하게 주어진다면 넓게 탐색하는 BFS알고리즘이 더 적합할 것이다.