**13주차 결과보고서**

전공: 아트&테크놀로지학과 학년: 4학년 학번: 20191048 이름: 김도솔

**1.**

**1) DFS (깊이 우선 탐색)**

bool ofApp::DFS(): DFS 알고리즘을 사용하여 미로의 탈출 경로를 찾는 함수이다.

* 사용한 자료구조: 스택(Stack)
* 함수 내부에서는 시작 위치와 도착 위치를 초기화하고, 방문 여부를 나타내는 visited 벡터와 스택 path를 선언한다.
* 스택이 비어있지 않은 동안 다음 과정을 반복한다.

스택의 맨 위에 있는 좌표를 가져온다.

가져온 좌표가 도착 위치와 일치하는지 확인한다. 일치하면 탈출 경로를 찾았다는표시를 하고 함수를 종료한다.

가져온 좌표에서 상하좌우로 이동할 수 있는지 확인하고, 이동 가능한 좌표를 스택에 추가한다. 이때, 방문한 좌표를 표시하고 방문한 순서대로 스택에 추가한다.

이동할 수 있는 좌표가 없으면 스택에서 제거한다.

* 스택이 모두 비었을 때까지 도착 위치를 찾지 못한 경우, 탈출 경로를 찾지 못했다는 의미로 false를 반환한다.

**2) DFS 탐색 경로 그리기**

void ofApp::dfsdraw(): DFS 탐색 경로를 그래픽으로 그려주는 함수이다.

* 미로를 그리는데 사용되는 rectLong과 rectShort 변수 등을 설정한다.
* 방문한 좌표를 표시한 visited 벡터를 기반으로 DFS 탐색 경로를 그래픽으로 표현한다. 방문한 좌표를 기준으로 사각형을 그리는 방식으로 탐색 경로를 표시한다.
* 탐색 경로를 표시한 result 벡터를 생성하고, 스택 path에서 좌표를 꺼내서 해당 좌표를 result 벡터에 표시한다.
* 최종적으로 result 벡터를 기준으로 탐색 경로를 그래픽으로 표시한다.

**3) BFS (너비 우선 탐색)**

bool ofApp::BFS(): BFS 알고리즘을 사용하여 미로의 탈출 경로를 찾는 함수이다.

* 사용한 자료구조: 큐(Queue)
* 함수 내부에서는 시작 위치와 도착 위치를 초기화하고, 방문 여부를 나타내는 visited 벡터와 각 좌표까지의 거리를 나타내는 dist 벡터를 선언한다.
* 큐에 시작 위치를 넣고, 해당 좌표를 방문했다고 표시하고 거리를 0으로 설정한다.
* 큐가 비어있지 않은 동안 다음 과정을 반복한다.

큐에서 맨 앞에 있는 좌표를 가져온다.

가져온 좌표가 도착 위치와 일치하는지 확인한다. 일치하면 탈출 경로를 찾았다는 표시를 하고 함수를 종료한다.

가져온 좌표에서 상하좌우로 이동할 수 있는지 확인하고, 이동 가능한 좌표를 큐에 추가한다. 이때, 방문한 좌표를 표시하고 이전 좌표의 거리에 1을 더한 값을 저장한다.

* 큐가 모두 비어있을 때까지 도착 위치를 찾지 못한 경우, 탈출 경로를 찾지 못했다는 의미로 false를 반환한다.

**4) BFS 탐색 경로 그리기**

void ofApp::bfsdraw(): BFS 탐색 경로를 그래픽으로 그려주는 함수이다.

* 미로를 그리는데 사용되는 rectLong과 rectShort 변수 등을 설정한다.
* 방문한 좌표를 표시한 visited 벡터를 기반으로 BFS 탐색 경로를 그래픽으로 표현한다. 방문한 좌표를 기준으로 사각형을 그리는 방식으로 탐색 경로를 표시한다.
* BFS 탐색 경로를 찾았을 경우, 탐색 경로를 표시한 bfspath 벡터를 생성한다.
* dist 벡터를 이용하여 역으로 시작 위치부터 도착 위치까지의 경로를 찾고, 해당 좌표를 bfspath 벡터에 표시한다.
* 최종적으로 bfspath 벡터를 기준으로 탐색 경로를 그래픽으로 표시한다.

**각 함수 시간 및 공간 복잡도**

DFS (깊이 우선 탐색):

시간 복잡도: O(V + E), 공간 복잡도: O(V)

BFS (너비 우선 탐색):

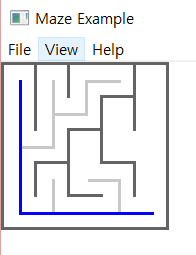
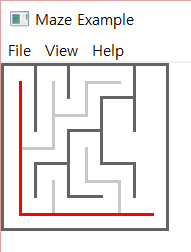
시간 복잡도: O(V + E), 공간 복잡도: O(V)

(V: 미로 내의 셀(정점) 수, E: 미로 내에서 연결된 경로(간선) 수)

전반적인 구현 방법은 예비 보고서에 작성했던 것과 거의 동일하다. 다른 점이 있다면 DFS, BFS에 따라 노드를 방문하는 방법까지는 일치하지만 경로를 그리는 방법에서는 실제 구현할 때 예비보고서와 조금 차이가 있었던 것 같다. DFS 경로 그릴 때 stack에 저장했던 경로를 2차원 벡터로 옮겨서 그리거나, bfs 경로를 그릴 때 방문했던 노드들을 다시 추적하는 방법으로 변형시켜 코드를 완성했다.

**2.**

**알고리즘 별 실행 결과**

 [DFS 실행 결과] [BFS 실행 결과]

**DFS (깊이 우선 탐색)의 장단점:**

메모리 사용량이 BFS에 비해 적고 구현이 쉽다. 현재 경로에서 더 이상 갈 수 없을 때까지 탐색한다. 경로가 하나만 존재하는 경우, 빠르게 찾을 수 있다. 그러나 해가 존재하지 않는 경우 전체를 탐색하고 최단 경로를 보장하지 못한다는 점에서 단점을 가진다. 목표까지의 경로가 여러 개인 경우, 처음으로 발견된 경로가 최단 경로가 아닐 수도 있다.

**BFS (너비 우선 탐색)의 장단점:**

DFS에 비해 덜 깊게 탐색한다. 탐색해야 하는 공간이 넓고 깊이가 깊은 경우에도 효율적이다. 경로가 존재한다면, 항상 최단 경로를 보장한다. 해가 존재하지 않는 경우, DFS보다는 빠르게 해가 없다는 것을 확인할 수 있다. 그러나 구현이 상대적으로 복잡하고 메모리 사용량이 많다. 큐에 모든 경로를 저장해야 하므로, 탐색 공간의 크기에 비례하는 메모리가 필요하다.

미로의 탈출 경로를 찾아야 하는 것이 우리의 목표이고, DFS나 BFS 경로도 모두 비슷한 자료구조(2차원 벡터)로 구현했기 때문에 시간복잡도와 공간복잡도도 비슷하다는 점에서 항상 최단 경로를 찾아주고 DFS에 비해 더 적은 수의 노드를 탐색하는 BFS 알고리즘이 더 적합하다고 판단할 수 있다.