전공 : 컴퓨터공학과 학년 : 4 학번 : 20212020 이름 : 박민준

1. 실습 및 숙제로 작성한 프로그램의 알고리즘과 자료구조를 요약하여 기술한다. 완성한 알고리즘의 시간 및 공간 복잡도를 보이고 실험 전에 생각한 방법과 어떻게 다른 지 아울러 기술한다.

1) 자료구조

(1) 2차원 배열 - 미로의 상태를 저장하기 위해 사용된다.

- char\*\* input: 텍스트 파일의 모든 정보를 담는 이차원 배열.

- vector<vector<char>> maze: 파일의 미로를 저장할 자료구조.

(2) 스택 - DFS에서 사용된다.

- stack<pair<int, int>> s: DFS 탐색을 위한 스택.

(3) 큐 - BFS에서 사용된다.

- queue<pair<int, int>> q: BFS 탐색을 위한 큐.

(4) 벡터

- vector<pair<int, int>> dfsFullPath: DFS 전체 경로를 저장하는 벡터.

- vector<pair<int, int>> dfsShortestPath: DFS 최단 경로를 저장하는 벡터.

- vector<pair<pair<int, int>, pair<int, int>>> bfsFullPath: BFS 전체 경로를 저장하는 벡터.

- vector<pair<int, int>> bfsShortestPath: BFS 최단 경로를 저장하는 벡터.

- vector<vector<bool>> visited: 방문 여부를 저장하는 2차원 벡터.

2) 알고리즘

(1) DFS (깊이 우선 탐색)

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- 먼저, 시작 노드를 스택에 넣고 방문 표시를 한다.

s.push({ startX, startY })

visited[startX][startY] = true

텍스트, 스크린샷, 소프트웨어, 디스플레이이(가) 표시된 사진

자동 생성된 설명

- 스택의 맨 위 노드를 꺼내 현재 위치로 설정한다.

auto [x, y] = s.top()

- 현재 위치가 미로의 끝에 도달하면 최단 경로를 저장한다.

- 그렇지 않으면, 인접한 노드 중 방문하지 않은 노드를 스택에 넣고, 현재 노드를 경로에 추가한다.

- 모든 인접 노드를 탐색하면 스택에서 꺼낸다.

s.pop()

- 모든 노드를 탐색한 후, DFS 경로와 최단 경로를 저장한다.

dfsFullPath.push\_back({ x, y })

dfsShortestPath = currentPath

(2) BFS (너비 우선 탐색)

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

- 먼저, 시작 노드를 큐에 넣고 방문 표시를 한다.

q.push({ startX, startY })

visited[startX][startY] = true

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- 그 다음, 큐의 맨 앞 노드를 꺼내 현재 위치로 설정한다.

auto [x, y] = q.front()

- 현재 위치가 미로의 끝에 도달하면 최단 경로를 저장한다.

- 그렇지 않으면, 인접한 노드 중 방문하지 않은 노드를 큐에 넣고, 부모 노드를 경로에 추가한다.

- 모든 인접 노드를 탐색하면 큐에서 꺼낸다.

q.pop()

- 모든 노드를 탐색한 후, BFS 경로와 최단 경로를 저장한다.

bfsFullPath.push\_back({ {x, y}, {nx, ny} })

bfsShortestPath.push\_back(current)

3) 시간 및 공간 복잡도

(1) DFS

- 시간 복잡도: O(V + E), 여기서 V는 노드의 수, E는 간선의 수이다.

- 공간 복잡도: O(V), 방문 여부를 저장하는 배열과 스택의 크기 때문이다.

(2) BFS

- 시간 복잡도: O(V + E), 여기서 V는 노드의 수, E는 간선의 수이다.

- 공간 복잡도: O(V), 방문 여부를 저장하는 배열과 큐의 크기 때문이다.

4) 실험 전후의 차이점

- 실험 전에는 DFS와 BFS의 경로 탐색이 각각의 전체 경로와 최단 경로를 정확하게 구할 것으로 예상하였다.

- 실험 후, DFS와 BFS가 예상대로 작동했지만, 경로 시각화 과정에서 벽과 노드의 구분이 필요하다는 것을 알게 되었다. 이로 인해, 경로를 그릴 때 노드 공간과 벽 공간을 정확하게 구분하여 표시하도록 수정하였다.

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.

1) 프로그램 실행 결과

- 프로그램을 실행하여 DFS와 BFS 알고리즘을 통해 미로를 탐색하고, 각각의 경로를 시각적으로 표시한 결과는 다음과 같다

(1) DFS (깊이 우선 탐색):

- 전체 경로(DFS Full Path)는 한 경로를 끝까지 탐색한 후 돌아와 다른 경로를 탐색하는 방식으로 진행된다.

- 최단 경로(DFS Shortest Path)는 우연히 가장 먼저 발견한 경로가 된다.

(2) BFS (너비 우선 탐색):

- 전체 경로(BFS Full Path)는 같은 깊이의 노드들을 모두 탐색한 후 다음 깊이의 노드들을 탐색하는 방식으로 진행된다.

- 최단 경로(BFS Shortest Path)는 출발지에서 목표지까지의 최단 경로를 보장한다.

2) 알고리즘 장단점

(1) DFS (깊이 우선 탐색)

[장점]

- 구현이 간단하다.

- 스택을 사용하기 때문에 메모리 사용량이 상대적으로 적다.

- 경로가 매우 깊은 경우, 빠르게 해결할 수 있다.

[단점]

- 최단 경로를 찾는다는 보장이 없다.

- 깊이가 깊을 경우, 무한히 깊어질 수 있어 비효율적이다.

- 사이클이 있는 그래프와 같이 특정한 경우에는 모든 노드를 탐색할 가능성이 있다.

(2) BFS (너비 우선 탐색)

[장점]

- 항상 최단 경로를 보장한다.

- 경로의 길이가 짧을 때 효율적이다.

[단점]

- 큐를 사용하기 때문에 메모리 사용량이 많다.

- 노드와 간선이 많은 경우, 많은 메모리와 시간이 필요하다.

- 깊이가 깊은 경우, 매우 비효율적일 수 있다.

3) 자료구조에 적합한 알고리즘

- 미로 탐색 문제에서는 BFS가 더 적합하다. 그 이유는 다음과 같다.

(1) 최단 경로 보장: 미로의 출발지에서 목표지까지 최단 경로를 찾는 것이 주된 목표이기 때문에, BFS가 이를 보장할 수 있어 더 유리하다.

(2) 명확한 경로 탐색: BFS는 같은 깊이의 노드들을 먼저 탐색하므로, 경로의 깊이가 깊어지지 않고 명확하게 탐색할 수 있다.

- DFS는 특정 상황(ex: 경로가 매우 깊은 경우나 메모리 사용이 제한된 경우)에 유리할 수 있지만, 미로 탐색 문제에서는 BFS가 주로 더 나은 선택이 된다.

- 결론적으로 프로그램을 실행하여 DFS와 BFS 알고리즘을 비교해본 결과, BFS가 미로 탐색 문제에서 더 적합하다는 것을 알 수 있었다. BFS는 최단 경로를 보장하며, 경로 탐색이 명확하고 효율적이다. 반면, DFS는 구현이 간단하고 메모리 사용량이 적지만, 최단 경로를 찾지 못할 가능성이 크다. 미로 탐색 문제와 같은 경우에는 BFS 알고리즘을 사용하는 것이 더 바람직하다.

직사각형, 패턴, 사각형, 대칭이(가) 표시된 사진

자동 생성된 설명직사각형, 패턴, 사각형, 대칭이(가) 표시된 사진

자동 생성된 설명

[BFS 실행 결과] [DFS 실행 결과]