13주차 결과보고서

전공 : 경영학과 학년 : 4학년 학번 : 20190808 이름 : 방지혁

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.

우선 본 프로젝트에 대해 설명하자면, .maz파일로부터 미로를 읽어 그래프로 변환한 후, DFS와 BFS 알고리즘을 이용해 경로를 탐색한다.

이를 위해 사용한 자료구조에 대해 설명하겠다.

Cell_info 구조체의 경우 4개의 Boolean 변수로 각 방향별로 벽의 존재 여부를 저장한다. 이를 2차원배열로 선언하여 미로 전체를 저장할 수 있도록 한다.

그래프 인접리스트

vector<vector<vector<pair<int, int>>>> graph;

각 좌표마다 이동 가능한 인접좌표를 pair<int, int>형태로 저장한다. 공간복잡도에 대해 분석하자면, O(HEIGHT * WIDTH * 평균 이웃 개수)로 볼 수 있겠다. 이웃이 실질적으로 상수 시간복잡도에 근사하므로 O(HEIGHT * WIDTH)라고 볼 수 있다.

탐색 저장 자료구조

DFS

```
vector<vector<bool>> visited;                  // 방문 여부
vector<vector<pair<int, int>>> parent;         // 경로 역추적용 부모 노드
vector<pair<int, int>> visitOrder;             // 탐색 순서
vector<pair<int, int>> finalPath;              // 최종 경로 저장
```

BFS

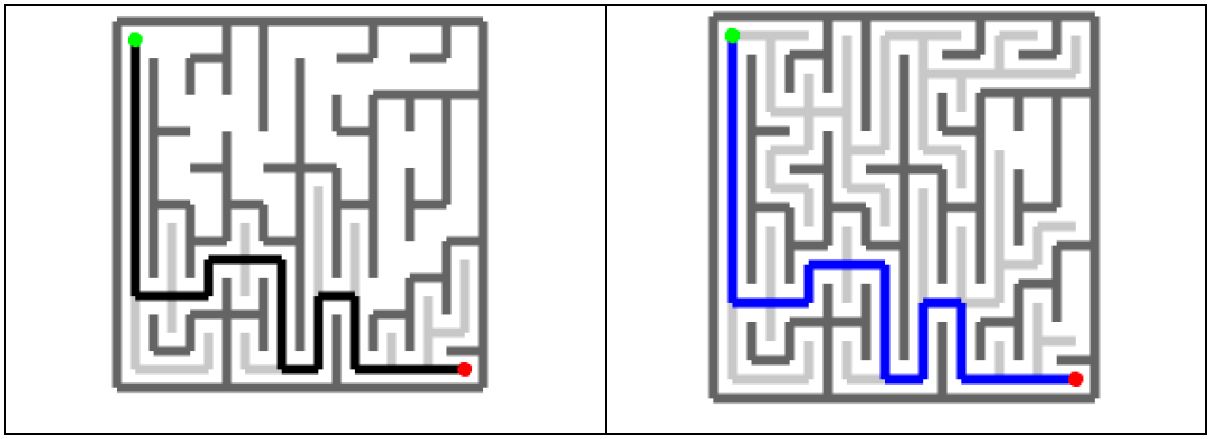
```
vector<vector<bool>> visited_bfs;              // 방문 여부
vector<vector<pair<int, int>>> parent_bfs;     // 경로 역추적용 부모 노드
vector<pair<int, int>> visitOrder_bfs;         // 탐색 순서
vector<pair<int, int>> finalPath_bfs;          // 최종 경로 저장
```

자료구조는 둘 다 변수 이름만 다르고 동일하다. 공간복잡도는 이전과 마찬가지로 O(HEIGHT * WIDTH)라고 볼 수 있다.

알고리즘 설명

.maz 파일을 찾아 줄별로 읽어 vector<string> lines에 저장한다. 그리고 WIDTH와 HEIGHT을 계산한다. 그리고 cells 2차원 배열에 각 셀마다 벽 정보를 저장한다. 그리고 buildGraph함수로 인접리스트를 구성한다. DFS의 경우 stack을 사용하며 한 방향으로 끝까지 파고드는 특성을 가지고 있다. 시작하는 좌표에 대해 방문했다고 체크하고 스택에 넣는다. 도착지가 아닌 경우, 움직일 수 있는 방향이 있다면 해당 좌표를 스택에 더해주며 계속 진행한다. 만약 더 이상 이동할 수 없다면 스택에 빼고, 다시 이전의 이동 가능 좌표로 돌아간다. 최악의 경우 O(V+E)의 시간 복잡도를 가진다. 공간복잡도의 경우 긴 경로를 계속 따라갈 때를 고려하면 O(V)가 될 것이다. 반면 BFS의 경우, queue를 사용하고 레벨 단위로 탐색하여 최단 경로가 보장된다. Queue에 시작 좌표를 넣고, 도착지인지 확인한다. 아니라면 계속 움직일 수 있는 인접한 모든 좌표를 큐에 넣으며 반복한다. 모든 노드를 한 번씩 방문하고 모든 간선을 한 번씩 방문하기에 O(V + E)의 시간 복잡도를 가진다. 기존 구현에서는 좌표를 1차원 노드 번호로 변환하려고 했었다. 그런데 더 복잡해지는 것 같아서 바꿨다. 그리고 초기에는 하나의 visited 및 parent 배열만 사용하려 했으나 편의상 이름만 바꿔 하나 더 추가하여 사용하였다.

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.



DFS는 말 그래도 깊이 우선 탐색이기 때문에 한 방향으로 길게 뻗어나가는 형태이다. 막다른 길

을 만나면 되돌아 가서 다른 경로를 시도한다. BFS의 경우 너비 우선 탐색이기 때문에 시작점에서부터 물에 잉크를 탄 것처럼 퍼져나가는 형태이다. 모든 방향을 골고루 탐색한다고 볼 수 있다.

DFS의 경우 메모리 효율적이다 stack의 크기가 경로의 깊이에 비례하기에 긴 미로에서도 메모리 사용량이 낮다고 볼 수 있다. 또한, 재귀나 스택으로 구현할 수 있기에 간단하다. 모든 방향으로 가는 것이 아니기에 특정 방향으로 운 좋게 시도하면 빠르게 해를 찾을 수 있다. 그러나, 최단 경로 보장이 되지 않는다. 처음 찾은 경로를 반환하기 때문에 BFS보다 긴 경로를 반환할 수 있다. 그리고 앞서 서술했던 빠른 해 찾기라는 장점이 단점이 될 수 있다. 어느 방향부터 탐색하느냐에 따라 결과 나오는 시간이 달라질 수 있다.

BFS의 경우 장점으로는 최단경로를 보장한다. 또한, 레벨 별로 체계적으로 탐색한다. 그러나 메모리 사용량이 높고, 목표가 멀리 있으면 많은 노드를 레벨별로 탐색하는 것이기에 탐색 시간이 오래 걸릴 수도 있다. C++의 경우 queue가 라이브러리가 제공되기에 구현이 쉽지만 c에서는 귀찮을 수도 있다.

이 프로젝트에서는 BFS가 더 적합하다고 생각한다. 주 목적은 최단 경로 찾기이며, 미로가 그렇게 크지 않아 메모리에 크게 제한사항이 없는 상태이기 때문이다. 개선 사항을 추가한다면 A* 알고리즘으로 휴리스틱한 측면을 고려하여 개선한다던지, 단순하게 선언한 visited배열을 비트맵으로 바꿀수도 있다.