2020년도 2학기 컴퓨터공학설계및실험Ⅰ

13주차 미로(Maze) 2주차 예비보고서

20161663 허재성

1. 실습 목적

이번 실험은 1주차 미로 만들기 프로젝트의 결과물인 ~.maz 파일을 읽어서 이를 윈도우 화면상에 표시함을 목적으로 한다. 실험 목적인 GUI(Graphical User Interface) 구축과 동시에 3주차에서 행할 프로젝트를 위하여 학생들은 창의적이고 효율적인 자료구조를 설계하여야 한다. GUI를 용이하게 설계하기 위하여 OpenFrameWorks를 사용하여 구현하도록 한다.

2. 관련 이론

2-1. DFS 알고리즘에 대하여 설명하시오.

DFS는 그래프에서의 탐색 방법 중 하나로 Depth First Search(깊이 우선 탐색)의 줄임말이다. 그래프의 임의의 정점 v에서 탐색을 시작하여 v를 먼저 방문한 후, v의 인접 정점들 중 아직 방문하지 않은 정점 w로부터 탐색을 시작한다. w를 탐색하기 전, 현재 정점인 v를 스택(Stack)에 저장한다. 만약 특정 정점 u에서 더 이상 탐색하지 않은 인접 정점이 없을 경우 스택에서 이전에 방문한 정점을 제거 후 스택에서 제거한 이전 정점의 다른 탐색하지 않은 정점에 대하여 탐색한다. 만약 스택이 비어 있고 더 이상 탐색할 정점이 없으면 DFS는 종료된다.[1]

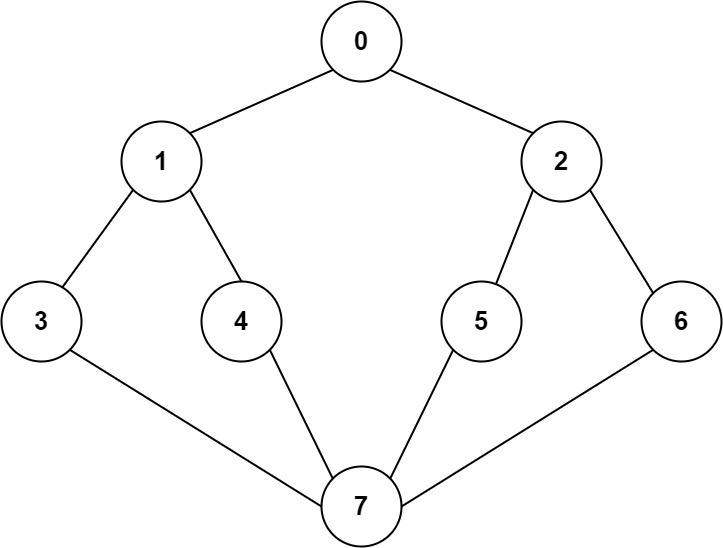
스택은 가장 나중에 들어온 원소가 먼저 나오는 LIFO 정책을 따르므로 스택에서 제거된 정점은 가장 최근에 방문한 정점이다.

DFS는 재귀함수로도 구현될 수 있다. 그래프의 임의의 한 정점 v에 대하여 DFS가 호출되면 먼저 시작 정점 v을 방문 후, v의 인접 정점 중 아직 방문하지 않은 정점 w에 대하여 재귀적으로 DFS 함수를 호출한다. 만약 인접한 정점 중에 방문하지 않은 정점이 존재하지 않을 경우 함수는 종료된다.

재귀함수로 구현한 DFS의 의사코드는 다음과 같다. 정점을 방문할 경우 해당 정점의 번호를 출력한다고 가정했다.[1]

|  |
| --- |
| void dfs(정점 v) { // 탐색을 시작할 정점 v  정점 v를 방문함;  print 정점 v의 번호;  for (v의 모든 인접한 정점 w에 대하여)  if (정점 w를 아직 방문하지 않음)  dfs(정점 w);  } |

다음과 같은 그래프에서 dfs를 수행한다고 가정하자. 정점 0에서부터 탐색을 시작한다. 색칠되지 않은 정점은 아직 방문하지 않은 정점을 의미한다. 임의의 정점에 대하여 방문하지 않은 인접 정점이 여러 개일 경우 정점의 번호가 작은 정점부터 방문한다고 가정하면 다음 그래프에 dfs를 수행했을 때 dfs의 과정과 정점의 방문 순서는 다음과 같다.



1. 정점 0을 방문한다.

2. 정점 0의 인접 정점(1, 2) 중 아직 방문하지 않은 정점 1을 방문한다.

3. 정점 1의 인접 정점(0, 3, 4) 중 아직 방문하지 않은 정점 3을 방문한다.

4. 정점 3의 인접 정점(1, 7) 중 아직 방문하지 않은 정점 7을 방문한다.

5. 정점 7의 인접 정점(3, 4, 5, 6) 중 아직 방문하지 않은 정점 4를 방문한다.

6. 정점 4의 인접 정점(1, 7) 중에서는 방문하지 않은 정점이 없으므로 이전 정점 7로 돌아간다.

7. 정점 7의 인접 정점(3, 4, 5, 6) 중 아직 방문하지 않은 정점 5를 방문한다.

8. 정점 5의 인접 정점(2, 7) 중 아직 방문하지 않은 정점 2를 방문한다.

9. 정점 2의 인접 정점(0, 5, 6) 중 아직 방문하지 않은 정점 6을 방문한다.

10. 정점 6의 인접 정점(2, 7) 중 방문하지 않은 정점이 없으므로 이전 정점 2로 돌아간다.

11. 정점 2의 인접 정점(0, 5, 6) 중 방문하지 않은 정점이 없으므로 이전 정점 5로 돌아간다.

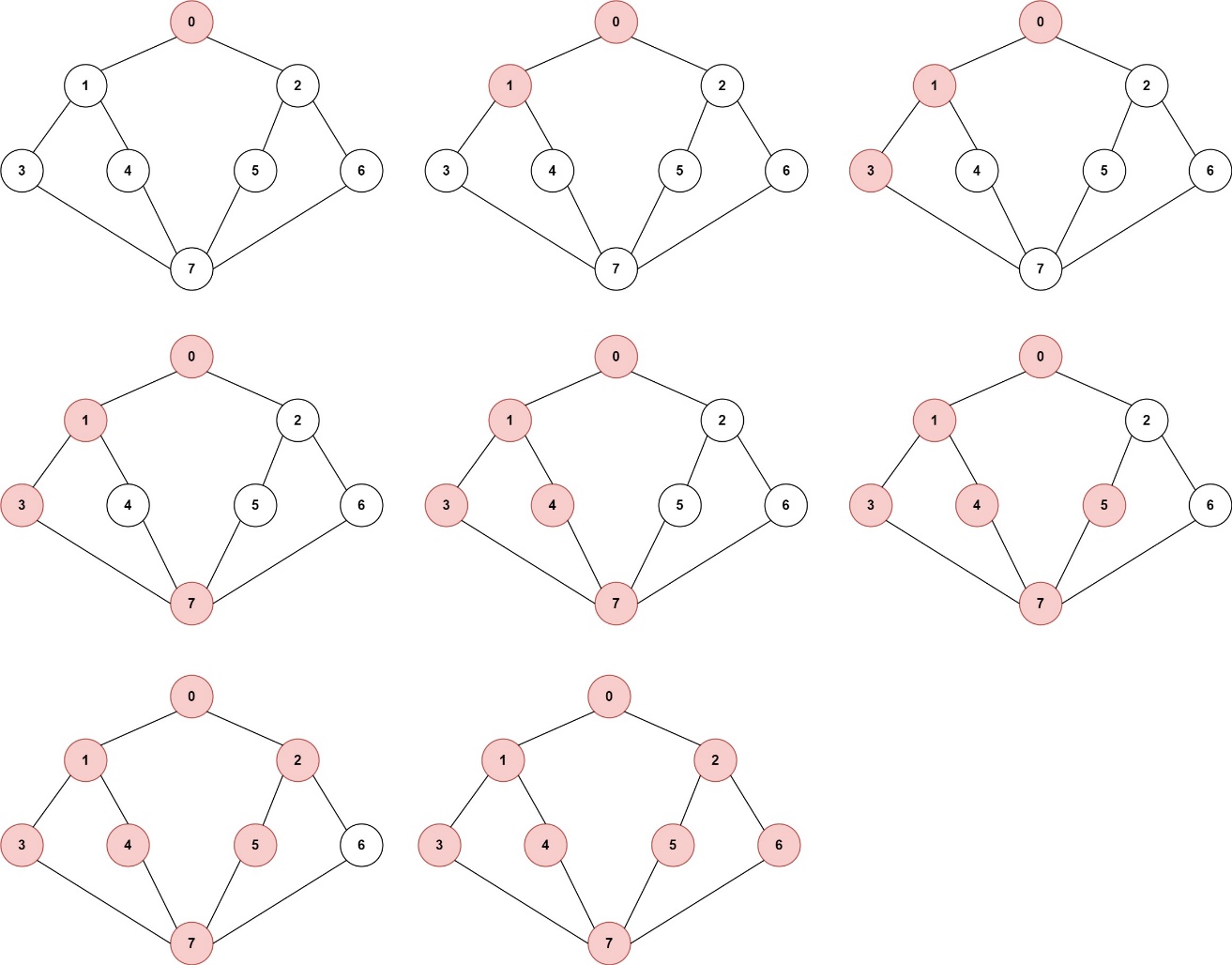
12. 정점 5의 인접 정점(2, 7) 중 방문하지 않은 정점이 없으므로 이전 정점 7로 돌아간다.

13. 정점 7의 인접 정점(3, 4, 5, 6) 중 방문하지 않은 정점이 없으므로 이전 정점 3으로 돌아간다.

14. 정점 3의 인접 정점(1, 7) 중 방문하지 않은 정점이 없으므로 이전 정점 1으로 돌아간다.

15. 정점 1의 인접 정점(0, 3) 중 방문하지 않은 정점이 없으므로 이전 정점 0으로 돌아간다.

16. 정점 0의 인접 정점(1, 2) 중 방문하지 않은 정점이 없으므로 탐색을 종료한다.



2-2. BFS 알고리즘에 대하여 설명하시오.

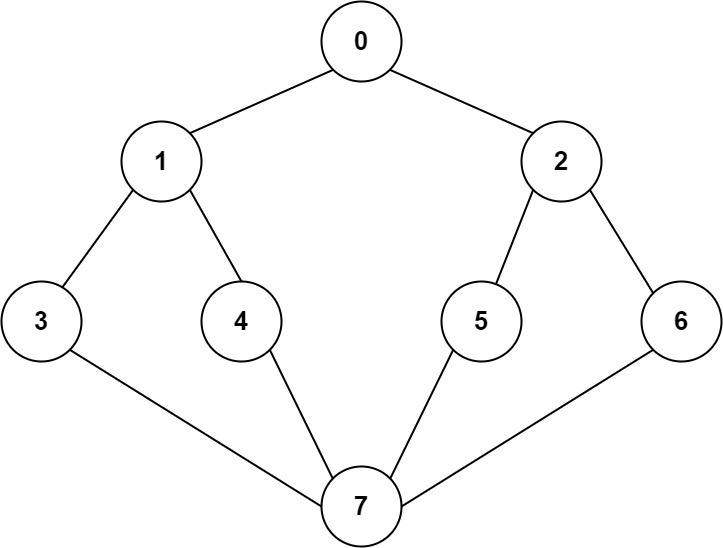
BFS는 그래프에서의 탐색 방법 중 하나로 Breadth First Search(너비 우선 탐색)의 줄임말이다. 그래프의 임의의 정점 v에서 탐색을 시작하여 v를 먼저 방문한 후, v의 모든 방문하지 않은 인접 정점들을 한번씩 방문한다. 더 이상 방문할 인접 정점이 없을 경우, 가장 먼저 방문한 인접 정점의 인접 정점들 중 방문하지 않은 정점들을 차례로 방문한다. 이 과정을 구현하기 위해 큐(Queue) 자료구조를 이용한다. 정점을 방문할 때마다 큐에 저장한다. 큐에서 정점을 삭제하여 가져오고 가져온 정점의 모든 인접 정점들을 방문한 뒤 각각 큐에 저장한다. 더 이상 방문할 인접 정점이 없으면 큐에서 정점을 삭제하여 첫 번째로 방문한 인접 정점을 가져온 후, 해당 정점의 인접 정점들을 방문하다. 큐가 비었고 더 이상 방문할 인접 정점이 없을 경우 종료된다.[1]

큐는 가장 먼저 들어온 원소가 먼저 나오는 FIFO 정책을 따르므로 큐에서 제거된 정점은 큐에 저정된 인접 정점 중 가장 처음 방문한 인접 정점이다.

BFS의 의사코드는 다음과 같다. 정점을 방문할 경우 해당 정점의 번호를 출력한다고 가정했다. [1]

|  |
| --- |
| void bfs(정점 v) { // 탐색을 시작할 정점 v  큐 Q를 초기화;  정점 v를 방문함;  print 정점 v의 번호;  Q에 v를 저장;  While(큐가 비어 있지 않을 경우) {  v = Q에서 제거한 정점;  for(v의 모든 인접 정점 w에 대하여) {  if(정점 w를 아직 방문하지 않음) {  정점 w를 방문함;  print 정점 w의 번호;  Q에 w를 저장;  }  }  }  } |

다음과 같은 그래프에서 bfs를 수행한다고 가정하자. Dfs와 같은 방법으로 탐색할 경우 탐색 과정과 정점의 방문 순서는 다음과 같다.



1. 정점 0을 방문한다.

2. 정점 0을 큐에 삽입한다.

3. 큐에서 정점 0을 제거한다.

4. 큐에서 제거한 정점 0의 인접 정점 1, 2를 각각 방문 후 큐에 삽입한다.

5. 큐에서 정점 1을 제거한다.

6. 큐에서 제거한 정점 1의 인접 정점 3, 4를 각각 방문 후 큐에 삽입한다.

7. 큐에서 정점 2를 제거한다.

8. 큐에서 제거한 정점 2의 인접 정점 5, 6를 각각 방문 후 큐에 삽입한다.

9. 큐에서 정점 3을 제거한다.

10. 큐에서 제거한 정점 3의 인접 정점 중 아직 방문하지 않은 정점 7을 방문 후 큐에 삽입한다.

11. 큐에서 정점 4를 제거한다.

12. 정점 4의 인접 정점 중 더 이상 방문할 정점이 없으므로 아무 일도 하지 않는다.

15. 큐에서 정점 5를 제거한다.

13. 정점 5의 인접 정점 중 더 이상 방문할 정점이 없으므로 아무 일도 하지 않는다.

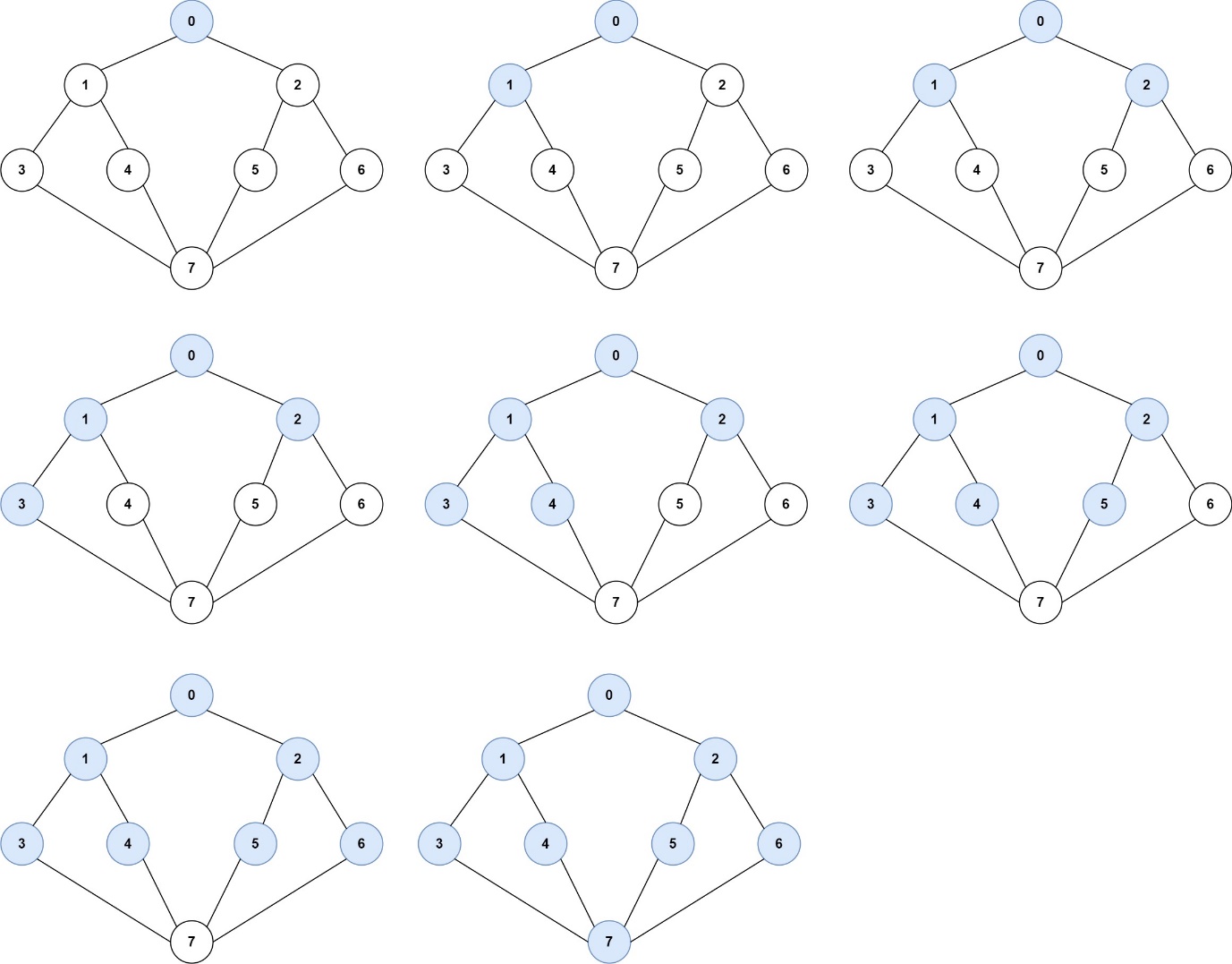
14. 큐에서 정점 6을 제거한다.

15. 정점 6의 인접 정점 중 더 이상 방문할 정점이 없으므로 아무 일도 하지 않는다.

16. 큐에서 정점 7을 제거한다.

17.. 정점 7의 인접 정점 중 더 이상 방문할 정점이 없으므로 아무 일도 하지 않는다.

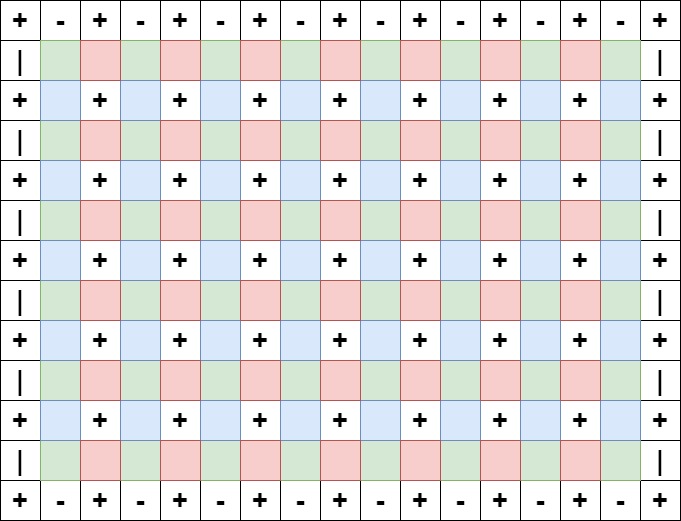
.18. 큐가 비었으므로 탐색이 종료된다.



3. 실습 방법

3-1. 14주 실험에서 화면에 그린 미로를 찾는 기능을 구현해야 하는데, 이를 고려하기 위해 어떻게 .maz 파일을 프로그램에서 읽어 저장할 것인지 자료구조를 상세히 제시하시오.

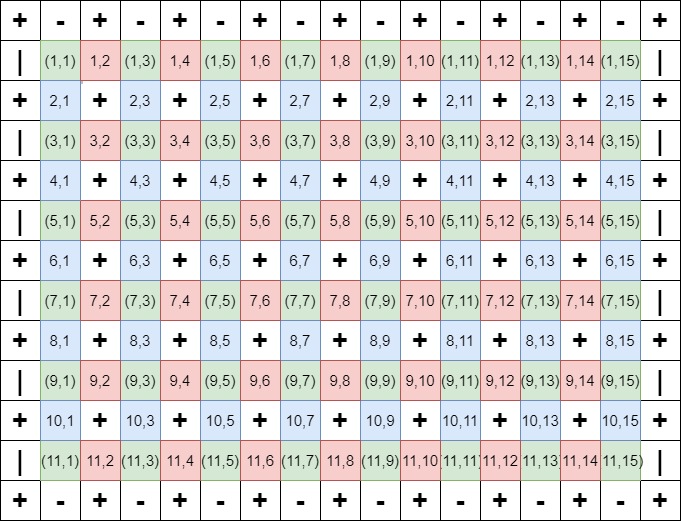
12주차에서 생성한 미로는 파일 maze.maz에 다음 그림과 같이 저장되어 있다. 그림은 가로로 방이 8개, 세로로 방이 6개인 미로의 기본 틀을 나타낸 것으로 외곽벽을 제외한 벽은 나타내지 않았다. 그림에는 나타나지 않았지만 한 행이 끝나고 줄바꿈을 위한 ‘\n’ 문자도 저장도 되어 있어야 하므로 실제로는 각 행마다 그림에 나타나지 않은 한 칸이 더 존재한다. 따라서 8\*6 미로를 필요하기 위한 배열의 크기는 열 18, 행 13의 2차원 배열이 필요하다.



임의의 W\*H 미로를 저장하기 위해서는 열 2\*W + 2, 행 2\*H + 1의 2차원 배열이 필요하다.

흰색 칸은 미로의 외곽 벽 및 모서리를 나타내는 칸으로 ‘+’, ‘-‘, ‘|’이 저장되어 있다. 초록색 칸은 미로의 방이 있는 칸으로 ‘ ‘만이 저장되어 있다. 빨간색 칸은 같은 행의 방 사이의 벽이 존재할 수 있는 공간으로 벽이 존재하면 ‘|’, 존재하지 않으면 ‘ ‘이 저장되어 있다. 마찬가지로 파란색 칸은 같은 열의 방과 방 사이의 벽이 존재할 수 있는 공간으로 벽이 존재하면 ‘-‘, 존재하지 않으면 ‘ ‘이 저장되어 있다. 미로의 모든 정보를 저장하기 위해 char형 2차원 배열을 동적할당해야 한다. 크기는 2\*H + 1 개의 행과 2\*H + 2개의 열을 가진다.

미로의 크기를 알아내기 위해 파일을 한 번 읽어 ‘\n’이 처음 나올 때까지의 문자의 개수를 알아내 미로의 폭을 알아낼 수 있고, 미로의 전체 문자의 개수를 알아낸 후 파일을 닫는다. 알아낸 미로의 폭과 전체 문자 개수를 이용해 미로의 높이도 알아낼 수 있고 미로의 한 행의 방의 개수와 한 열의 방의 개수도 알아내어 배열을 알아낸 크기로 동적 할당할 수 있다. 배열을 동적 할당 후 파일을 다시 열어서 미로의 내용을 배열에 저장하면 된다.



미로를 저장한 2차원 배열에서 방(초록색)이 존재하는 인덱스를 표시하면 위와 같다. 일반화하면 행의 인덱스를 i, 열의 인덱스를 j라 할 때, 1<= i <= 2 \* H – 1, 1 <= j <= 2 \* W – 1이고 i, j가 모두 홀수인 배열의 원소는 방을 나타낸다. 방의 개수는 W\*H개가 된다.

저장된 미로의 원소가 방에 해당할 때, 그래프에서 방에 대응되는 정점을 만들어준다. 정점의 자료구조는 다음과 같이 정의할 수 있다.

|  |
| --- |
| typedef struct \_Vertex {  int name; // 방의 번호 (1~W\*H)  bool visited; // 해당 방을 방문한 적이 있는지를 나타내는 변수  bool up, down, left, right; // 인접한 방과의 연결 상태  } Vertex; |

Vertex 자료구조는 각 방의 번호(이름)를 의미하는 name 변수를 가지고 있다. 필요에 따라 사용할 수도 있고, 필요하지 않다면 생략할 수 있다.

bool형(또는 int형으로도 구현 가능) visited는 해당 정점이 나타내는 방을 방문한 적이 있는지를 나타내는 변수이다. 처음 정점을 만들 때에는 어떤 방도 방문한 적이 없으므로 visited를 모두 false로 설정한다. 미로를 탐색해 나가며 벽이 없어서 방을 이동할 떄마다 이동한 방에 해당하는 정점의 visited를 true로 변경해 준다. 미로를 탐색할 때 DFS 또는 BFS에서 어떤 방의 인접한 방 중 이미 탐색한 방을 다시 탐색하지 않기 위해 이와 같은 변수가 필요하다.

bool형 변수 up, down, left, right는 한 방의 주변에 존재할 수 있는 4개의 방과의 연결 상태를 나타내는 변수로 up, down, left, right 순으로 각각 현재 방의 같은 열의 바로 위 방, 바로 아래 방, 같은 행의 왼쪽 방, 오른쪽 방과의 연결 상태를 의미한다. 만약 현재 방과 인접 방 사이의 벽이 존재하지 않아서 직접 연결되어 있다면 변수의 값을 true로 설정하고, 벽이 존재해서 직접 연결되지 않았다면 false로 설정한다. true일 경우 방을 나타내는 정점과 인접한 방의 정점 사이에 간선이 존재한다는 의미이다. 그래프는 방향 그래프가 아니므로(Undirected graph) 현재 정점에서 다른 정점으로의 간선이 있으면 그 다른 정점에서 현재 정점으로 가는 간선도 존재해야 한다.

두 방이 같은 행에서 인접하고 두 방 사이 벽이 없으면 왼쪽 방을 나타내는 Vertex 구조체의 right는 true이고, 오른쪽 방을 나타내는 Vertex 구조체의 left도 true여야 한다.

위의 그림에서 인덱스 i가 1일 경우 i=1인 행의 방들은 위의 방이 존재하지 않는다. 따라서 해당 Vertex 구조체의 up은 무조건 false가 되어야 한다. 마찬가지로 i=2\*H-1일 경우, 해당 행의 방들은 아래 방이 존재하지 않으므로 Vertex 구조체의 down을 무조건 false로 설정해야 한다.

같은 방법으로 인덱스 j가 1일 경우, j=1인 열의 방들은 왼쪽 방이 존재하지 않으므로 해당 Vertex 구조체의 left는 무조건 false가 되어야 한다. 마찬가지로 j=2\*W-1일 경우, 해당 열의 방들은 오른쪽 방이 존재할 수 없으므로 Vertex 구조체의 right를 false로 설정해야 한다.

구석에 존재하는 네 개의 방의 경우 연결될 수 있는 방이 최대 2개이므로 나머지 2개는 무조건 false이다.

미로를 찾기 위해 DFS 또는 BFS를 이용할 경우, 현재 방과 연결된 방이 여러 개일 때 어떤 방을 먼저 탐색할지 우선순위가 필요하다. 우선 순위를 구현하는 예로 인접한 방의 번호를 비교하여 더 큰 방(또는 더 작은) 방을 먼저 탐색하는 것을 생각할 수 있다. 일반적으로 오른쪽 아래로 갈수록 방의 번호가 커지고 보통 미로의 시작점은 왼쪽 위, 목표점은 오른쪽 아래인 경우가 많으므로 번호가 큰 방을 먼저 탐색하도록 구현하는 것이 하나의 방법이 될 수 있다. 하지만 시작점과 도착점이 꼭 왼쪽 위, 오른쪽 아래로 정해진다고 보장할 수 없으면, 다른 우선 순위가 더 유용할 수도 있다.

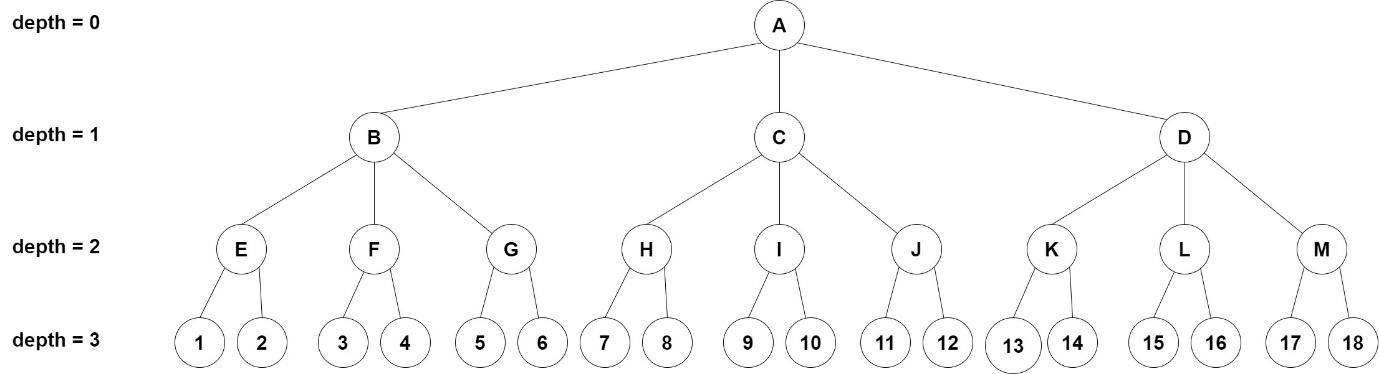
3-2. 1에서 설계한 자료구조에서 DFS와 BFS를 구현할 방법을 설명하고, 어떤 방법이 더 미로 찾기에 적합한지 그 이유와 함께 설명하시오.(적합한 예시를 드는 것이 좋음)

자료구조는 위에 언급한 자료구조를 사용한다. 알고리즘은 DFS와 BFS에서 설명한 알고리즘을 자료구조에 맞게 구현하면 된다. 간략하게 설명하면 DFS의 경우 미로의 시작점이 정해지면 시작점에 해당하는 방의 정점의 visited를 true로 변경한다. up, down, left, right는 미로를 저장할 때 이미 결정되어 있으므로 저장된 값을 확인하면 된다. 각 변수를 확인하여 true인 경우의 인접 방 중 설정한 우선순위가 더 높은 방에 대하여 재귀적으로 DFS를 적용하면 된다. true인 인접한 방 중 visited가 false인 방이 없다면 더 이상 탐색할 방이 없으므로 DFS를 종료한다.

BFS의 경우 정점 Vertex를 저장할 큐를 구현해야 한다. 미로의 시작점이 정해지면 시작점에 해당하는 방의 정점의 visited를 true로 변경한 후, 해당 정점을 구현한 큐에 삽입한다. 큐에 저장 후, 큐에서 가장 먼저 저장된 정점을 제거 후 가져온다. 가져온 정점에 대하여 인접 정점(최대 4개)의 방과의 연결 상태를 비교 후 연결 상태가 true인 방 중 아직 방문하지 않은 방들을 방문한다. 방문 순서는 우선 순서에 따라 방문하며 방문할 때마다 해당 방의 visited를 true로 설정 후 큐에 삽입한다. 모든 방을 방문 후, 큐에 남아 있는 정점 중 가장 먼저 저장된 정점을 제거하여 해당 정점의 방에 대해서 같은 과정을 반복한다. 큐가 비었고, 큐에서 제거한 정점의 방의 인접 방 중 더 이상 방문할 방이 없으면 종료된다.

12주차에서 생성한 미로는 완전 미로 또는 폐쇄 공간 없이 사이클이 존재하는 불완전 미로이므로 모든 임의의 두 방 사이에는 경로가 반드시 존재하고, DFS, BFS 어떤 방법을 사용해도 목적지에 도착할 수 있다.

트리에서의 BFS 알고리즘의 단점은 다음과 같다. 다음과 같은 트리에 BFS를 적용해보자. (다음 트리는 미로와는 무관하다.)



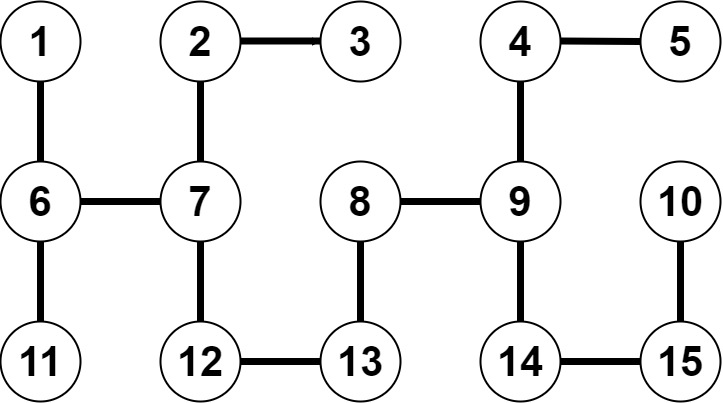
루트 노드 A를 시작점으로 BFS를 적용하면 처음 큐에 노드 A가 저장된다. A를 큐에서 제거하면 B, C, D가 저장된다. B를 제거하면 이미 큐에 남아 있는 C, D에 이어 E, F, G가 저장되고, C를 제거하면 큐에 D, E, F, G에 이어 H, I, J가 저장되며 D를 제거하면 E, F, G, H, I, J에 이어 K, L, M이 저장된다.

이와 같은 방식으로 BFS의 큐에서 제거한 노드의 깊이가 lv이면 깊이가 lv+1인 노드들이 모두 큐에 저장되어야 한다.

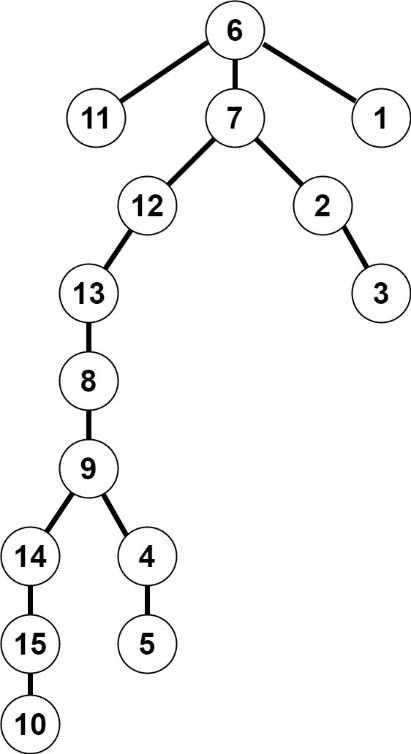
꽉찬 이진 트리(Full Binary Tree)의 경우 단말 노드(leaf node)를 제외한 모든 노드의 자식 노드가 2개이다.[1] 따라서 루트 노드의 깊이가 0이고 이진 트리의 깊이가 d이면 최대 깊이의 노드의 개수는 2d개이다. 이 경우 루트 노드에 BFS를 적용하면 필요한 큐의 크기는 최대 2d이므로 큐의 공간 복잡도는 O(2d)이다. 만약 이진 트리가 아닌 3진 트리, 4진 트리, 또는 테트리스의 추천 시스템에 사용한 34진 트리를 사용할 경우 큐의 공간 복잡도는 각각 O(3d), O(4d), O(34d)로 매우 커지게 된다.

DFS의 경우 스택의 크기는 트리의 최대 깊이에 비례한다. 따라서 트리의 최대 깊이가 d일 경우 시간 복잡도는 d이다. 트리의 노드의 개수가 N일 때, 트리가 균형 잡힌 트리일 경우 스택의 공간 복잡도는 O(logN)이다.

미로의 경우 한 방의 인접한 방의 개수는 최대 4개이다. 실제로 완전 미로를 생성해 트리로 만들어 보면 특정 방의 자식 노드가 최대 4개인 경우는 루르 노드를 제외하고 존재할 수 없다.



위와 같은 완전 미로의 그래프를 6을 루트로 하는 트리로 나타내면 다음과 같은 모양이 된다.



트리의 노드가 가지고 있는 자식 노드의 개수가 많지 않으므로 각 층마다 트리의 노드의 개수도 많지 않음을 알 수 있다. 따라서 BFS의 큐의 공간 복잡도가 크게 문제되지 않음을 알 수 있다. 반대로 트리의 깊이가 깊어져 DFS를 사용 시 필요한 스택의 공간 복잡도가 더 큰 것을 알 수 있다. DFS를 스택을 사용하지 않고 재귀를 이용해 구현할 경우 재귀 함수의 호출 횟수가 커져서 Stack Overflow 에러가 발생할 가능성이 높아진다.

또한 DFS로 미로를 탐색할 경우 잘못된 경로의 깊이가 깊을 경우 목표에 도달하는데 오랜 시간이 걸린다. 위의 미로에서 시작점을 6, 도착점을 3이라 설정하고, 인접한 방이 여러 개일 경우 번호가 큰 방부터 탐색한다고 가정하자.

DFS의 경우 탐색 순서는 다음과 같다. 목적지에 도달하면 탐색을 종료한다.

6->11->7->12->13->8->9->14->15->10->4->5->2->**3**

반면 BFS의 경우 탐색 순서는 다음과 같다.

6->11->7->1->12->2->13->**3**

DFS의 경우 목적지가 없는 경로를 탐색하기 시작하면 목적지가 없어도 인접한 모든 방을 탐색하게 되므로 목적지에 도달하는 시간이 오래 걸리게 된다. 반면 BFS의 경우 특정 경로로 계속 탐색하는 것이 아닌 여러 경로로 얕게 탐색하므로 잘못된 경로로 빠져 불필요한 탐색을 할 가능성이 줄어 든다.

또한 만약 미로가 무한할 경우 DFS로 잘못된 경로를 탐색하기 시작하면 무한한 잘못된 경로에 빠져 목적지에 도달할 수 없을 가능성이 생긴다. BFS의 경우 미로가 무한한 경우에도 목적지를 찾을 수 있다.

4. 기타

DFS에서 무한한 잘못된 경로에 빠질 경우를 대비해 탐색 깊이에 제한(depth bound)을 두는 방법이 존재한다. 깊이 제한까지 탐색했는데도 목적지에 도달하지 못하면 바로 이전 노드로 돌아가 다른 인접 노드를 탐색한다.[2]

5. 참고 문헌

[1] [Ellis Horowitz](http://www.yes24.com/SearchCorner/Result?domain=ALL&author_yn=Y&query=Ellis+Horowitz), [Sartaj Sahni](http://www.yes24.com/SearchCorner/Result?domain=ALL&author_yn=Y&query=Sartaj+Sahni), [Susan Anderson-Freed](http://www.yes24.com/SearchCorner/Result?domain=ALL&author_yn=Y&query=Susan+Anderson-Freed), “Fundamentals of Data Structures in C”, [*Silicon Valley Publishers Group*](javascript:void(0);), 2007

[2] 위키백과, “깊이 우선 탐색”, https://ko.wikipedia.org/wiki/%EA%B9%8A%EC%9D%B4\_%EC%9A%B0%EC%84%A0\_%ED%83%90%EC%83%89