2020년도 2학기 컴퓨터공학설계및실험Ⅰ

12주차 미로(Maze) 1주차 결과보고서

20161663 허재성

1. 실습 목적

미로는 복잡한 길을 찾아 출발점으로부터 시작해 도착점까지 도달하는 퍼즐로 직사각형으로 배열된 방과 이러한 방들 사이를 막고 잇는 벽으로 구성된다. 벽은 각 방의 상하좌우 사면에 존재할 수 있는데 인접한 두 방 사이에 벽이 없을 경우 두 방 사이를 자유로이 지날 수 있다. 본 프로젝트에서는 다음에 설명하는 완전미로를 만들 수 있는 여러 가지 방법을 알아보고 이를 직접 구현하기 위한 자료구조 설계 및 프로그래밍을 통해 미로를 생성해보도록 한다.

2. 실습 구현 내용

완전 미로를 구현하기 위해 Eller’s Algorithm을 이용했다. 미로의 크기를 나타내는 변수 WIDTH와 HEIGHT를 전역적으로 선언하였다. 각각 미로의 가로 방의 개수, 세로 방의 개수이다.

|  |
| --- |
| int WIDTH, HEIGHT; |

main 함수에는 미로를 한 줄씩 그리기 위해 필요한 1차원 배열을 할당하기 위한 int\* 변수 maze,heightWall, widthWall과 maze.maz 파일에 미로를 출력하기 위해 FILE 포인터 fp를 선언하였다.

|  |
| --- |
| int\* maze = NULL;  int\* heightWall = NULL;  int\* widthWall = NULL;  FILE\* fp; |

maze는 미로를 한 줄씩 생성할 때 각 방의 번호를 저장하는 동적 할당한 배열을 가리키는 포인터이다. heightWall은 방과 방 사이의 좌우의 벽 ‘|’ 의 유무를 저장하는 배열을 가리킨다. widthWall은 방과 방 사이의 위 아래 벽 ‘-‘의 유무를 저장하는 배열을 가리킨다. 벽의 유무는 0, 1로 표현하며 0(FALSE)일 경우 벽이 존재하지 않고, 1(TRUE)일 경우 벽이 존재한다.

또한 미로의 각 방의 번호를 나타내는 변수 room을 선언 후 1로 초기화하였다.

|  |
| --- |
| int room = 1; |

main 함수에서 표준 입력으로 WIDTH와 HEIGHT를 입력 받아서 필요한 1차원 배열을 동적 할당한다. maze는 미로의 한 행의 방의 개수만큼 할당되므로 공간 복잡도는 O(WIDTH)이다. heightWall의 경우 한 행의 방과 방 사이의 벽의 개수만큼 할당되므로 마찬가지로 공간복잡도는 O(WIDTH)이다. widthWall의 경우 현재 행과 위의 행의 방 사이의 벽의 개수만큼 할당되므로 공간복잡도는 O(WIDTH)이다. 따라서 전체 공간 복잡도는 O(WIDTH)이다. drawMaze 함수를 호출해 완전 미로를 파일 maze.maz에 그린 후, 동적 할당된 메모리를 해제 후 종료한다.

완전 미로를 구현하기 위해 추가한 함수는 다음과 같다.

(1) void resetHeightWall(int\* heightWall);

세로 벽 ‘|’이 방과 방 사이에 존재하는 상태로 설정한다. 한 행에서 세로 벽의 개수는 WIDTH - 1개이므로 시간 복잡도는 O(WIDTH)이다.

input : 특정 행의 방과 방 사이의 벽의 정보를 저장하는 배열을 가리키는 포인터 heigthWall

output : 없음

(2) void resetWidthWall(int\* widthWall)

가로 벽 ‘-’이 방과 방 사이에 존재하는 상태로 설정한다. 한 행에서 가로 벽의 개수는 WIDTH개이므로 시간 복잡도는 O(WIDTH)이다.

input : 현재 행의 방과 아래(다음) 행의 방 사이의 벽의 정보를 저장하는 배열을 가리키는 포인터 widthWall

output : 없음

(3) void drawCeilingFloorOutline(FILE\* fp)

미로의 외곽벽(미로 바깥과 만나는 벽) 중 가장 위와 가장 아래의 외곽벽을 그리는 함수이다. WIDTH의 개수만큼 “+-“을 출력한 후 마지막으로 “+\n”을 출력하여 줄바꿈이 되도록 한다. 시간 복잡도는 O(WIDTH)이다.

input : 미로를 저장할 파일 maze.maz에 출력하기 위한 FILE 포인터 fp

(4) void decideFirstRowWalls(int\* maze, int\* heightWall)

미로의 첫 번째 행의 방 사이의 벽을 결정하는 함수이다. 만약 HEIGHT가 1일 경우 벽이 존재하면 폐쇠 공간이 반드시 존재해 완전 미로가 될 수 없으므로 1일 경우에는 heightWall의 모든 값을 FALSE로 설정해 모든 벽을 제거한다. 2 이상일 경우, 난수 0, 1을 발생시켜 벽을 임의로 제거하거나 유지한다. 만약 벽을 제거해 두 방이 연결될 경우 두 방의 번호를 하나로 통일한다. 이 때 두 방의 번호 중 앞의 방의 번호가 더 작으므로 그 번호로 통일한다.

input : 미로의 첫 째 행의 방 번호를 저장하고 있는 배열을 가리키는 포인터 maze, 특정 행의 방과 방 사이의 벽의 정보를 저장하는 배열을 가리키는 포인터 heigthWall

output : 없음

(5) void drawHeightWalls(int\* heightWall, FILE\* fp)

미로의 방과, 방 사이의 세로 벽 ‘|’를 그리는 함수이다. 방은 공백 ‘ ‘로 그린다. heightWall이 결정되면 그에 따라 벽을 그린다. 왼쪽 외곽 벽 ‘|’을 먼저 그린다.heightWall의 i번째 원소가 TRUE이면 i번째 방과 i+1번째 방 사이의 벽이 존재하므로 ‘|’를 출력하고 FALSE면 벽이 존재하지 않으므로 ‘ ‘를 출력한다. 벽을 다 출력 후, 오른쪽 외곽 벽 ‘|’을 출력 후 줄을 바꿔준다. 시간 복잡도는 O(WIDTH)이다.

input : 세로 벽의 정보를 저장하고 있는 배열의 포인터 heightWall, 미로를 저장할 파일 maze.maz에 출력하기 위한 FILE 포인터 fp

output : 없음

(6) void drawWidthWalls(int\* widthWall, FILE\* fp)

현재 행의 방과, 아래(다음) 행의 방 사이의 가로 벽 ‘-‘과 방의 모서리 ‘+’를 그리는 함수이다. widthWall이 결정되면 그에 따라 벽을 그린다. 왼쪽 가장자리 모서리 ‘+’을 먼저 그린다. widthWall의 i번째 원소가 TRUE이면 현재 행의 i번째 방과 다음 행의 i번째 방 사이의 벽이 존재하므로 ‘-’와 모서리 ‘+’를 출력하고 FALSE면 벽이 존재하지 않으므로 ‘ ‘과 모서리 ‘+’를 출력한다. 벽과 모서리를 다 출력 후, 줄을 바꿔준다. 시간 복잡도는 O(WIDTH)이다.

input : 가로 벽의 정보를 저장하고 있는 배열의 포인터 widthWall, 미로를 저장할 파일 maze.maz에 출력하기 위한 FILE 포인터 fp

output : 없음

(7) void decideInterRowWalls(int\* maze, int\* widthWall, int\* room)

새로 그릴 행과 이전 행의 방 사이의 벽을 결정하는 함수이다. 0, 1 둘 중 하나를 임의로 설정하여 0일 경우 벽을 없애고 1일 경우 벽을 유지한다. 만약 이전 행의 방이 같은 행의 다음 방과 연결되지 않았고 다음 행의 방 마저도 연결되지 않았을 경우, 폐쇄공간이 되므로 flag의 값과 상관 없이 벽을 없앤다.

만약 현재 행의 마지막 방이 이전 행의 어떤 방하고도 연결되지 않았을 경우 벽을 강제로 없앤다.

벽이 존재할 경우 벽을 기준으로 구분되는 현재 행의 방의 번호를 증가시킨다.

한 행의 방의 개수가 WIDTH개이므로 시간 복잡도는 O(WIDTH)이다.

input : 이전 행의 방의 번호를 저장하고 있고 현재 행의 방의 번호를 저장할 배열의 포인터 maze, 이전 행과 현재 행의 방 사이의 벽의 정보를 저장할 배열의 포인터 widthRoom, 방의 번호를 가리키는 포인터 변수 room

output : 없음

(8) void decideMidRowWalls(int\* maze, int\* heightWall)

현재 행의 방과 방 사이 벽의 유무를 결정하는 함수이다. 만약 두 방의 번호가 같다면 이미 어떤 경로로 연결된 상태이므로 벽을 없앨 경우 경로가 하나 더 생긴다. 따라서 이럴 경우 벽을 유지해야 한다. 두 방의 번호가 다르다면 flag에 난수를 저장해 flag에 따라 벽을 제거할 지 결정한다. 벽을 제거하기로 결정했으면 더 작은 방의 번호로 방 번호를 통일한다. 통일 후 합쳐진 방과 같은 번호를 가진 방들도 모두 연결되므로 해당 방들의 번호도 변경한다. 방과 방 사이 벽의 개수가 최대 WIDTH - 1개이고 벽을 제거했을 경우 번호가 같은 방을 찾기 위해 WIDTH개의 방을 모두 확인해야 한다. 벽을 제거하는 횟수가 k라면 시간 복잡도는 O(kWIDTH)이다. 벽을 제거하는 횟수 k는 flag를 어떻게 설정할지에 따라 달라지며 이전 미로 상태에 따라서도 달라지는 임의의 값이다. k가 WIDTH에 비해 충분히 작을 경우, 시간 복잡도는 O(WIDTH)로 생각할 수 있다.

input : 현재 행의 방의 번호를 저장하고 있는 배열의 포인터 maze, 세로 벽의 정보를 저장하고 있는 배열의 포인터 heightWall

output : 없음

(9) void decideLastWalls(int\* maze, int\* heightWall)

마지막 행의 방 사이의 벽을 결정하는 함수이다. 모든 벽 ‘|’이 존재하도록 설정 후, 만약 인접한 두 방의 번호가 같을 경우 이미 어떤 경로로 연결되어 있으므로 벽을 유지한다. 만약 두 방의 번호가 다르다면 벽을 제거하지 않을 경우 더 이상 두 방이 연결될 수 없으므로 반드시 벽을 제거한다. 벽을 제거하여 두 방이 연결되면 더 작은 방의 번호로 번호를 통일한다. 번호를 통일 후 이이전에 같은 방 번호를 가진 방을 모두 찾아 번호를 통일해줘야 한다. 번호가 같은 방을 찾기 위해 WIDTH개의 방을 모두 확인해야 한다. 벽을 제거하는 횟수가 k라면 시간 복잡도는 O(kWIDTH)이다. 벽을 제거하는 횟수 k는 인접한 두 방의 번호가 다를 경우의 수이며 이전까지의 미로 상태에 따라서 달라지는 임의의 값이다. k가 WIDTH에 비해 충분히 작을 경우, 시간 복잡도는 O(WIDTH)로 생각할 수 있다.

input : 현재 행의 방의 번호를 저장하고 있는 배열의 포인터 maze, 세로 벽의 정보를 저장하고 있는 배열의 포인터 heightWall

output : 없음

(10) void drawMaze(int\* maze, int\* heightWall, int\* widthWall, int\* room, FILE\* fp)

완전 미로를 그리는 함수로 위에 정의한 함수들을 호출하는 역할을 한다. 먼저 미로의 첫 번째 행의 방 번호를 초기화한다. 각 방의 번호를 1씩 증가하게 설정한다. drawCeilingFloorOutline 함수를 호출하여 미로의 위쪽 외곽선을 그리고 decideFirstRowWalls 함수를 호출하여 첫 번째 행의 방 사이의 벽을 결정한 후, drawHeightWalls 함수를 호출하여 첫 번째 행의 방 사이의 벽을 그린다.

만약 HEIGHT가 1일 경우 미로는 첫 번째 행만 존재한다. 1일 경우 바로 drawCeilingFloorOutline 함수를 호출하고 종료한다. HEIGHT가 2 이상일 경우 행과 행 사이의 방의 벽을 결정하는 decideInterRowWalls 함수를 호출하고 drawWidthWalls 함수를 호출하여 가로벽 ‘-‘를 그린다. HEIGHT가 2 이상일 경우부터 첫 째 행과 구분되는 마지막 행이 있으므로 decideLastWalls 함수를 호출하여 마지막 행의 방 사이의 벽을 결정 후 drawHeightWalls 함수를 호출하여 벽을 그린다.

HEIGHT가 3 이상일 경우 첫 째 행과 마지막 행 사이에 HEIGHT - 2개의 행이 존재한다. 각 행마다 decideInterRowWalls 함수를 호출하여 이전 행의 방과의 벽을 결정 후 drawWidthWalls 함수를 호출하여 벽을 그린다. 또한 각 행마다 decideMidRowWalls 함수를 호출하여 방 사이의 벽을 결정 후, drawHeightWalls 함수를 호출하여 벽을 그린다.

호출하는 함수들의 시간 복잡도는 O(WIDTH)이고 반복문은 HEIGHT – 2번 반복한다. 또한 처음 행과 마지막 행을 위해 함수들을 호출한다. 따라서 시간 복잡도는 O(WIDTH \* HEIGHT)이다.

input : 현재 행의 방의 번호를 저장할 배열의 포인터 maze, 세로 벽의 정보를 저장할 배열의 포인터 heightWall, 가로 벽의 정보를 저장할 배열의 포인터 widthWall, 방의 번호를 나타내는 포인터 room, 미로를 저장할 파일 maze.maz에 출력하기 위한 FILE 포인터 fp

output : 없음

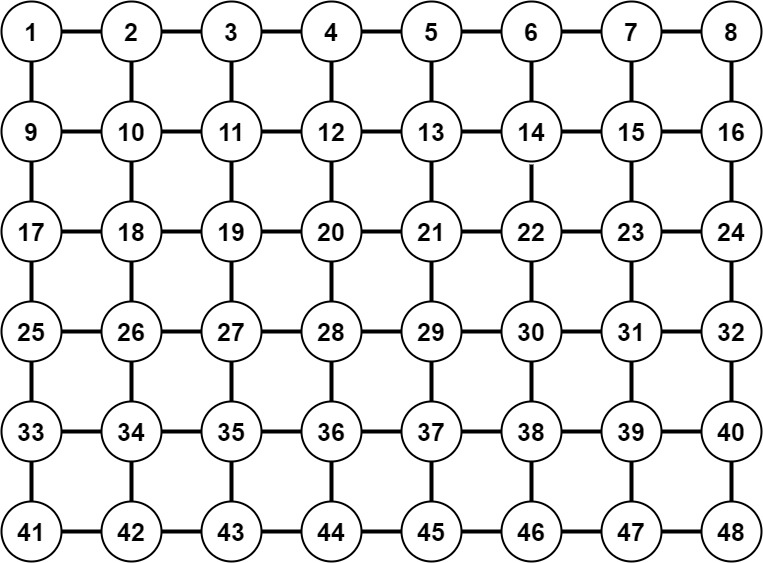
3. 실습 환경

CSPRO 서버 : CSPRO.sogang.ac.kr

4. 실습 결과 및 분석

가로의 방의 개수가 W, 세로의 방의 개수가 H개인 미로는 총 방의 개수가 WH개이다. 미로는 그래프 자료구조로 나타낼 수 있다. 방을 그래프의 정점(vertex)로, 방과 방 사이의 벽의 유무를 그래프의 간선(edge)로 표현할 수 있다. 만약 미로의 어떤 두 방 사이에 벽이 없어서 두 방 사이를 이동할 수 있다면 두 방에 대응되는 두 정점 사이에는 간선이 존재한다. 반대로 두 방 사이에 벽이 있어 두 방 사이를 이동할 수 없다면 두 정점 사이에는 간선이 존재하지 않는다.

만약 미로에서 인접한 방끼리 벽이 전혀 존재하지 않는다면 그래프로 다음과 같이 나타낼 수 있다. 미로에서 W=8, H=6인 경우이다.



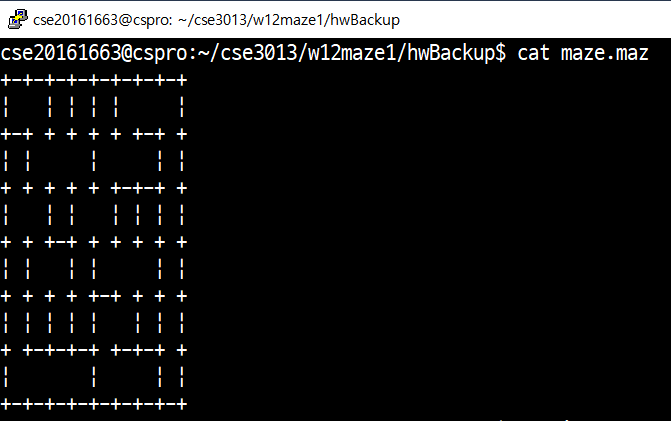
위, 아래, 좌, 우로 인접한 방 사이에 벽이 존재하지 않을 경우 그래프의 간선의 개수는 82개이다. 각 행의 방을 나타내는 정점 사이의 간선은 7개로 총 6개의 행이 존재하므로 42개가 존재한다. 첫 번째 행부터 시작해서 바로 아래 행의 방과 연결되는 간선의 개수는 8개이고 마지막 행을 제외하면 모두 다음 행과 연결되어 있으므로 총 8\*5 = 40개가 존재한다. 따라서 전체 간선의 개수는 82개이다.

일반적으로 가로로 방이 W개 있고, 세로로 방이 H개가 존재하는 미로를 나타내는 그래프에서 정점 사이의 간선의 개수는 최대 (W-1) \* H + W \* (H – 1) = 2\*WH – W – H 개이다.

완전 미로의 경우 미로의 임의의 두 방 사이의 경로가 반드시 하나만 존재해야 한다. 도달할 수 없는 방인 폐쇄공간이나 임의의 두 방 사이의 또 다른 경로가 존재하면 완전 미로가 아닌 불완전미로가 된다. 폐쇄공간은 그래프에서 연결되지 않은 정점들의 집합을 의미한다. 완전 미로의 그래프는 모든 정점이 서로 연결되어 있어야 한다. 또한 임의의 두 방 사이에 경로가 2개 이상일 경우 그래프에는 사이클이 존재한다. 완전 미로의 그래프에서는 사이클이 존재해서는 안된다.

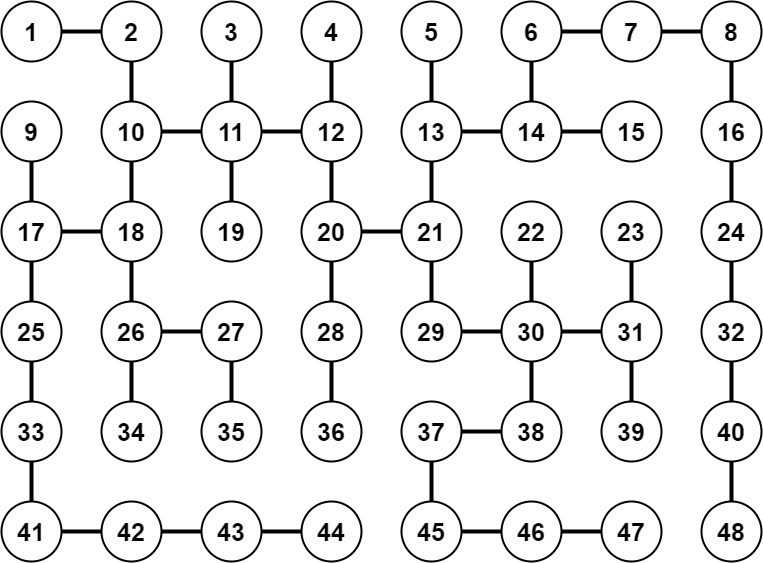
완전 미로의 그래프는 모든 정점이 연결되어 있고 사이클이 존재하지 않는 트리(Tree)가 되어야 한다. 트리의 노드(정점)의 개수가 N개일 때 간선의 개수는 N-1개이다. 따라서 완전 미로일 경우 그래프의 간선의 개수는 WH - 1개가 되어야 한다.

완전 미로를 생성하는 프로그램을 작성 후 실행한다. 미로의 크기로 가로 8, 세로 6을 입력하였다.



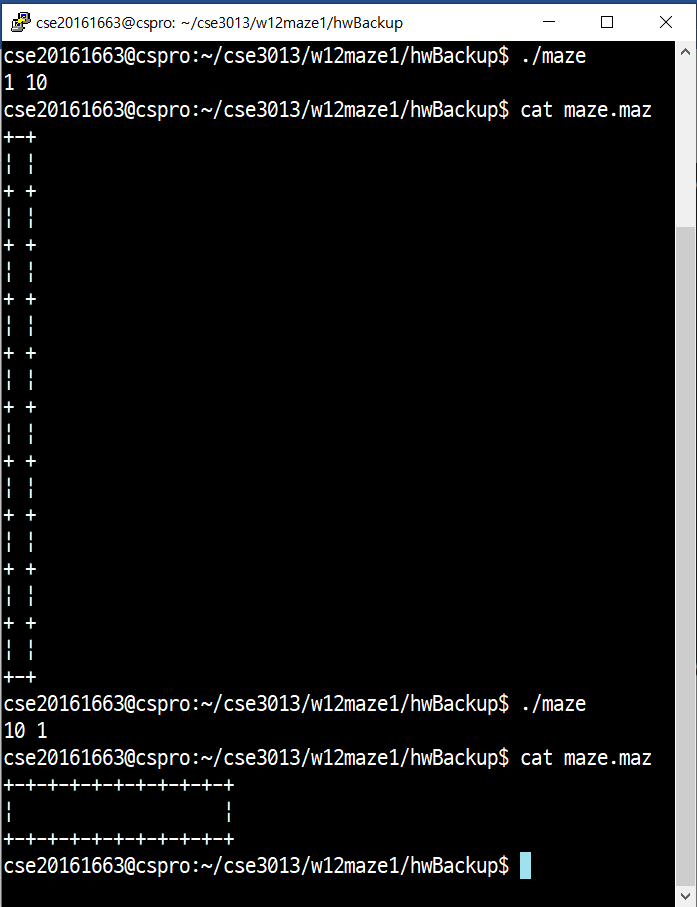
maze.maz 파일이 생성되었다. cat 명령어로 해당 파일의 내용을 출력해보면 위와 같다.

미로가 모든 방이 서로 연결되어 있고 두 방 사이의 경로가 단 하나만 존재하는지 확인해야 한다. 이를 위해 위의 미로를 그래프 형태로 표현 후, 해당 그래프가 트리가 되는지 확인한다. 트리가 되면 해당 미로는 완전 미로이다.

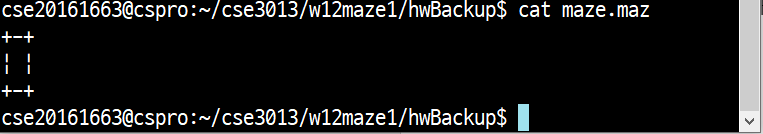


그래프로 표현하면 위와 같다. 그래프의 간선은 47개로 그래프는 트리가 된다. 따라서 완전 미로이다. 모든 정점이 연결되어 있고, 어떤 두 정점을 선택해도 두 정점 사이의 경로는 1가지뿐이다.

특수한 크기의 완전 미로가 잘 생성되는지 확인한다. 가로 또는 세로의 방 개수가 1일 경우 벽이 하나라도 존재하면 도달할 수 없는 방이 반드시 생긴다. 이와 같은 경우 방 사이의 벽이 존재해서는 안된다.

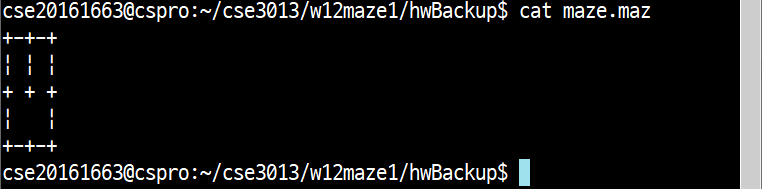
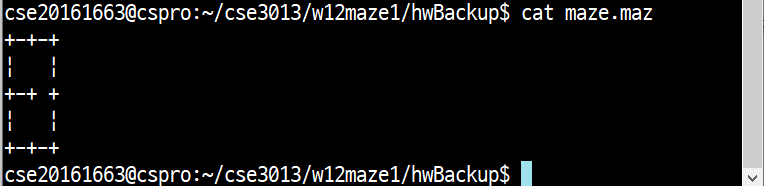


위의 미로는 가로의 크기가 1, 세로의 크기가 10인 미로이고, 아래의 미로는 가로의 크기가 10, 세로의 크기가 1인 미로이다. 두 경우 모두 방 사이의 벽이 전혀 존재하지 않는다. 가로 또는 세로의 길이가 1인 경우 두 방 사이의 경로가 2개 이상인 경우는 발생할 수 없으므로 고려하지 않는다.



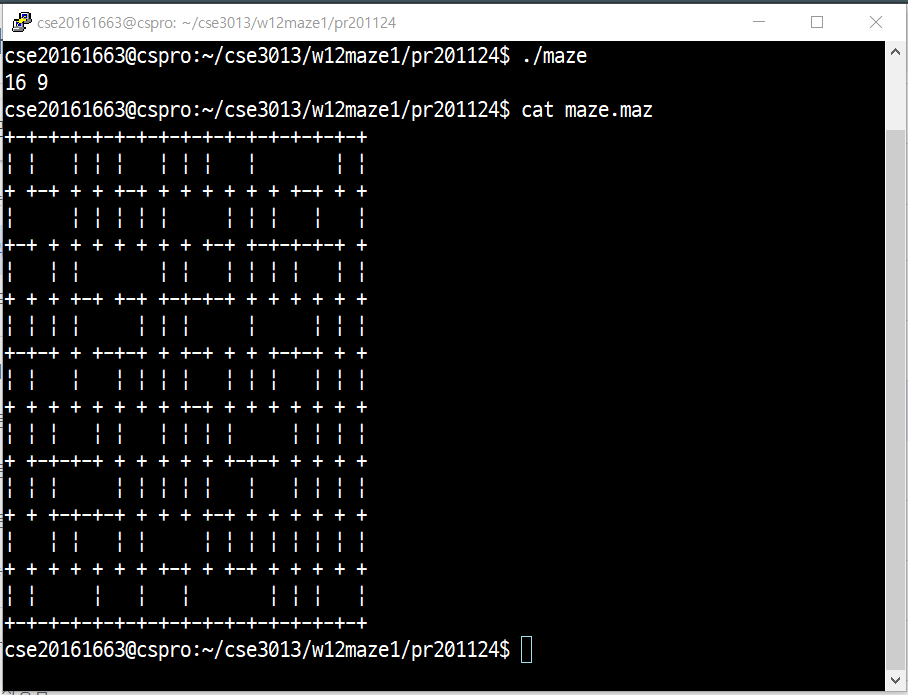
가로, 세로가 모두 1일 경우 방 하나와 둘러싸인 벽만 그려진다.

위와 같은 경우를 제외한 미로 중 크기가 가장 작은 미로는 가로, 세로의 길이가 2인 미로이다. 4개의 방이 존재하며 그래프의 간선의 개수는 최대 2\*2\*2-2-2 =4개가 존재할 수 있다. 완전 미로가 되기 위해서 간선의 개수는 2\*2-1=3개가 되어야 하고, 벽의 개수는 1개가 되어야 한다.

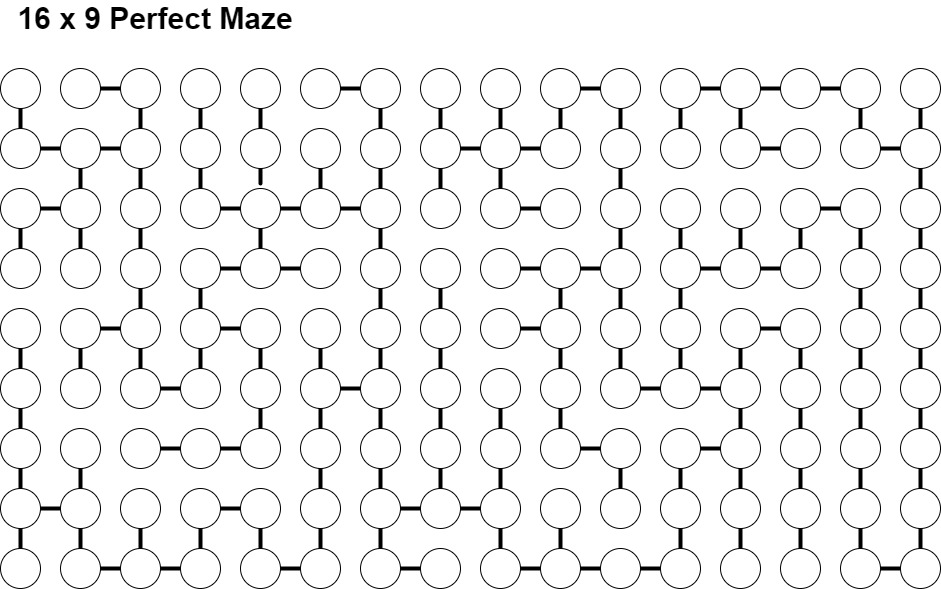


위와 같이 벽이 단 1개 존재하는 완전 그래프가 출력된다.

좀 더 큰 크기의 완전 미로를 확인해본다. 16\*9 크기의 완전 미로가 다음과 같이 만들어진다.



해당 미로를 그래프로 표현하면 다음과 같다.



그래프의 모든 정점이 연결되어 있고, 사이클이 존재하지 않는다. 연결되지 않은 정점이 없고 간선의 개수가 16\*9-1= 143개임을 확인하면 미로가 완전 미로임을 알 수 있다.

5. 과제

불완전 미로는 완전 미로와 다르게 임의의 두 정점 사이의 경로가 하나도 존재하지 않거나, 두 정점 사이의 경로가 2개 이상일 경우의 미로이다. 전자의 경우 도달할 수 없는 방인 폐쇄 공간이 존재하며 그래프에서는 연결되지 않는 정점이 존재하게 된다. 후자의 경우 그래프에서는 사이클이 생기게 된다.

과제에서는 후자의 경우인 사이클이 존재하는 불완전 미로를 생성한다. 완전 미로의 그래프는 트리로 모든 방들이 최소의 간선의 개수로 연결되어 있다. 만약 트리의 임의의 두 정점 사이에 간선을 하나 더 추가하면 이미 두 정점 사이에는 경로가 하나 존재하는데 경로가 하나 더 추가되어 사이클이 생긴다. 과제에서는 가로 길이 W, 세로 길이 H의 미로에서 W와 H 중 더 작은 값을 2로 나눈 몫(소수점 이하는 버린 값)만큼의 벽을 제거하여 불완전 미로를 생성한다.

미로에서 벽을 제거하는 것은 그래프에서 간선을 추가하는 것과 같다. 완전 미로에서 임의의 두개의 방 사이의 벽을 제거하는 것은 트리에서 임의의 두 정점 사이의 간선을 추가하는 것으로 반드시 사이클이 발생하고 미로는 불완전 미로가 된다. 따라서 완전 미로에서 필요한 개수만큼 임의로 벽을 제거하면 불완전 미로가 생성된다.

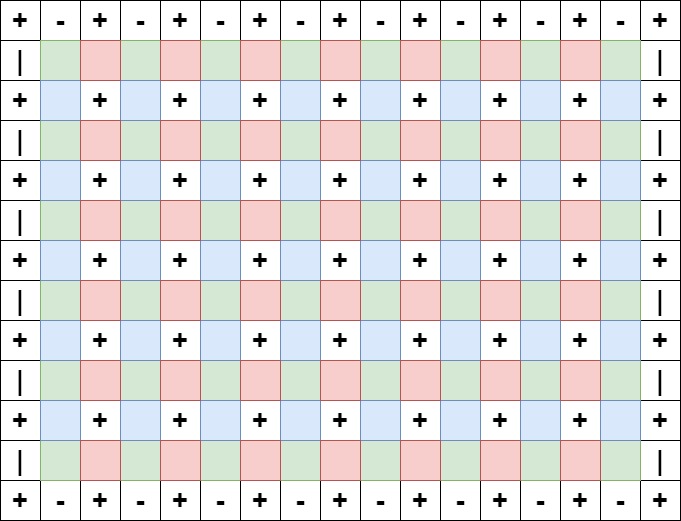
완전 미로를 생성할 때에는 미로의 정보를 별도의 배열에 저장하지 않고 바로 maze.maz 파일에출력하였다. 불완전 미로를 생성하기 위해 main 함수에 미로의 정보를 저장할 char형 2차원 배열을 동적 할당할 char\*\* mazeArray를 추가하였다.

|  |
| --- |
| char\*\* mazeArray = NULL; |

전역 변수로 동적 할당할 2차원 배열의 크기인 mazeHeight와 mazeWidth를 선언하였다. 각각 배열 mazeArray의 행의 크기와 열의 크기가 된다. 또한 불완전 미로를 생성하기 위해 추가적으로 삭제할 벽의 개수를 나타내는 delNum을 전역 변수로 추가한다.

|  |
| --- |
| int mazeHeight, mazeWidth;  int delNum; |

표준입력으로 WIDTH와 HEIGHT를 입력 받으면 mazeHeigth, mazeWidth, delNum은 main 함수에서 결정되고 그 크기에 따라 mazeArray도 동적 할당된다.



만약 8\*6 미로를 저장하려면 위와 같은 공간이 필요하다. 흰색으로 표시된 칸은 미로의 외곽 벽과 모서리를 나타내는 ‘+’를 저장하기 위한 공간이고 초록색으로 표시된 칸은 방을 표현하는 공간이다. 빨간색 칸과 파란색 칸은 각각 방 사이의 좌우 벽 ‘|’이 놓일 수 있는 공간과 위 아래 벽 ‘-‘이 놓일 수 있는 공간이다. 벽이 존재할 경우 ‘|’ 또는 ‘-‘이, 벽이 존재하지 않을 경우 공백 ‘ ‘이 저장되어 있다.

일반적으로 가로 W, 세로 H인 미로를 저장하기 위해 필요한 2차원 배열의 크기는 열 2\*W + 1, 행 2\*H + 1이다. 또한 미로에서 한 줄을 출력 후 줄바꿈을 위한 ‘\n’ 문자를 저장해야 하므로 행당 하나의 공간이 더 필요하다. 따라서 mazeHeight = 2\*H + 1, mazeWidth = 2\*W + 2로 결정된다.

mazeArray를 위한 추가적인 공간의 공간 복잡도는 O(mazeHeight \* mazeWidth) = O((2\*Height + 1) \* (2\*WIDTH + 2)) = O(HEIGHT \* WIDTH)이다.

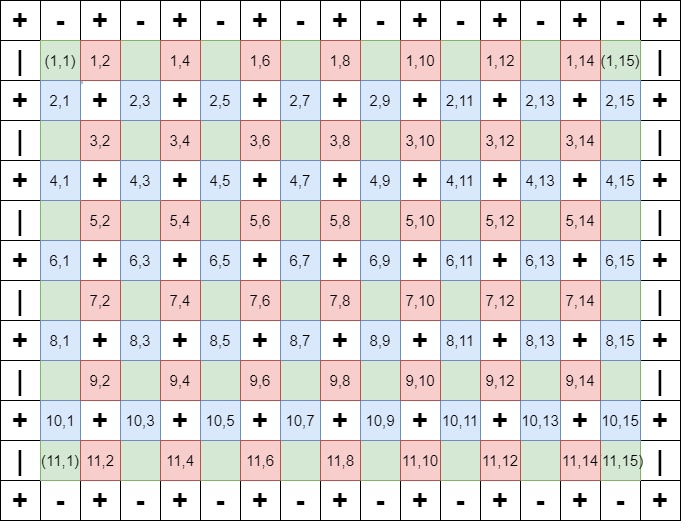
main 함수에서는 완전 미로가 저장된 maze.maz 파일을 읽기 모드로 열어서 미로의 정보를 동적할당한 2차원 배열 mazeArray에 저장하고 새로 정의한 함수 makeImperfectMaze 함수를 호출한다. makeImperfectMaze 함수에서 불완전 미로를 생성 후 maze.maz 파일에 출력 후 종료되면 동적 할당된 2차원 배열을 해제해주고 종료한다.

새로 추가된 함수 makeImperfectMaze는 다음과 같다.

(1) void makeImperfectMaze(char\*\* mazeArray, FILE\* fp);

완전 미로에서 delNum개의 벽을 없애 불완전 미로를 만드는 함수이다. 미로에서 최대로 존재 가능한 그래프에서 최대로 존재 가능한 간선의 개수와 같으므로 2WH – W - H이다. 완전 미로의 그래프에서 간선의 수는 WH - 1이므로 완전 미로에서의 벽의 개수는 WH – W – H + 1이다. 이 값을 walls라고 한다. mazeArray를 탐색하며 지나온(지운 경우, 지우지 않는 경우 둘 다 해당) 벽의 개수를 passed, 지운 벽의 개수를 deleted, 앞으로 지워야 할 벽의 개수를 toDelete로 선언한다. passed와 deleted는 각각 0으로 초기화하고 toDelete는 delNum으로 초기화된다.

완전 미로를 저장하고 있는 2차원 배열 mazeArray의 원소들을 이중 for문으로 돌며 하나씩 확인한다.



8\*6 미로에서 벽이 존재할 수 있는 인덱스의 범위는 1<= 열 <= 15, 1 <= 행 <= 11이다. 일반화하면 행 인덱스 i와 열 인덱스 j의 범위는 다음과 같다.

1 <= i < 2 \* HEIGHT, 1 <= j < 2 \* WIDTH

이 중 세로 벽 ‘|’이 존재할 수 있는 인덱스와 가로 벽 ‘-‘이 존재할 수 있는 인덱스는 다음과 같다.

세로 벽 : i가 홀수이고, j가 짝수

가로 벽 : i가 짝수이고, j가 홀수

이를 토대로 2중 for문을 작성할 수 있다. 2중 for문은 i=1부터 2\*HEIGHT-1까지 반복하며 각각의 i에 대하여 j=1부터 2\*WIDTH-1까지 반복한다. 위의 벽이 존재할 수 있는 인덱스에서는 벽을 지울지 결정하고 벽이 존재할 수 없는 인덱스의 경우 continue로 다음 인덱스로 넘어간다.

벽이 존재할 경우 다음과 같다.

1. 지워야 할 벽의 개수 toDelete가 앞으로 미로에 남은 벽의 개수 walls – passed보다 작을 경우 충분한 벽이 남았으므로 벽을 꼭 지우지 않아도 된다. 따라서 flag에 임의의 난수를 할당하여 해당 수가 특정 조건을 만족할 경우 벽을 지운다. 여러 가지 방법이 있지만 여기서는 0~100 사이의 난수를 발생시켜 그 중 10의 배수가 나올 경우 벽을 지우도록 설정했다. 벽을 지울 확률은 11/101로 벽을 발견하면 10% 정도의 확률로 벽을 제거한다. 벽을 지우지 않을 경우 벽을 그냥 지나치며 passed만 1 증가시킨다.
2. 지워야 할 벽의 개수 toDelete가 앞으로 미로에 남은 벽의 개수 walls – passed와 같다면 앞으로 만나는 모든 벽은 반드시 제거되어야 한다. 따라서 flag에 난수를 발생시키는 과정 없이 무조건 벽을 지운다.

i), ii) 모두 벽을 제거하면 passed와 deleted는 1 증가시키고, toDeleted를 1 감소시킨다.

만약 deleted가 지워야 할 벽의 개수 delNum과 같다면 반복을 종료한다.

반복문의 시간 복잡도는 최악의 경우 마지막 남은 벽을 지워야 하는 경우이다. 이 때는 배열 mazeArray의 거의 모든 인덱스를 다 탐색하게 된다. 이 때의 시간 복잡도는 O((2\*WIDTH – 1) \* (2\*HEIGHT – 1)) = O(WIDTH \* HEIGHT)이다.

flag를 이용해 벽을 지울 확률을 더 높이면 미로에서 먼저 만나는 벽을 지울 가능성이 높아지므로 반복 횟수가 감소하는 대신 벽이 앞쪽에서만 지워질 가능성이 높다.

delNum만큼 벽을 지운 후 FILE 포인터 fp로 기존의 maze.maz 파일을 쓰기 모드로 열어서 불완전 미로가 저장된 mazeArray의 내용을 maze.maz에 출력한다.

반복의 횟수는 mazeHeight\*mazeWidth이므로, 시간 복잡도는 O(WIDTH \* HEIGHT)이다.

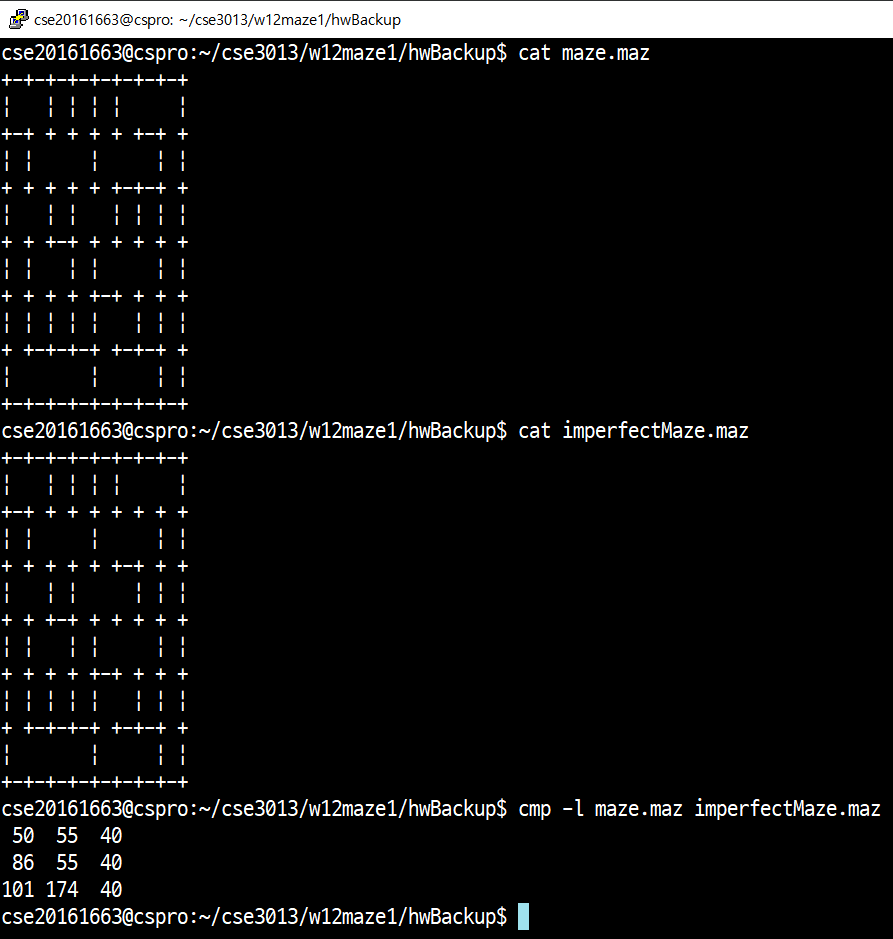
따라서 makeImperfectMaze 함수의 시간 복잡도는 O(WIDTH \* HEIGHT)이다.

만약 WIDTH 또는 HEIGHT가 1일 경우 실습에서 알아봤듯이 벽이 존재하지 않는다. 이 상태에서는 벽을 추가하지 않는 한 불완전미로를 생성할 수 없다. WIDTH 또는 HEIGHT가 1일 경우 delNum은 0이 된다. 이 경우 아무 일도 하지 않고 함수를 종료한다.

input: 완전 미로의 정보를 저장하고 있는 동적 할당된 배열을 가리키고 있는 char형 이중 포인터 mazeArray, maze.maz 파일을 열어 불완전 미로를 파일에 출력하기 위한 FILE 포인터 fp

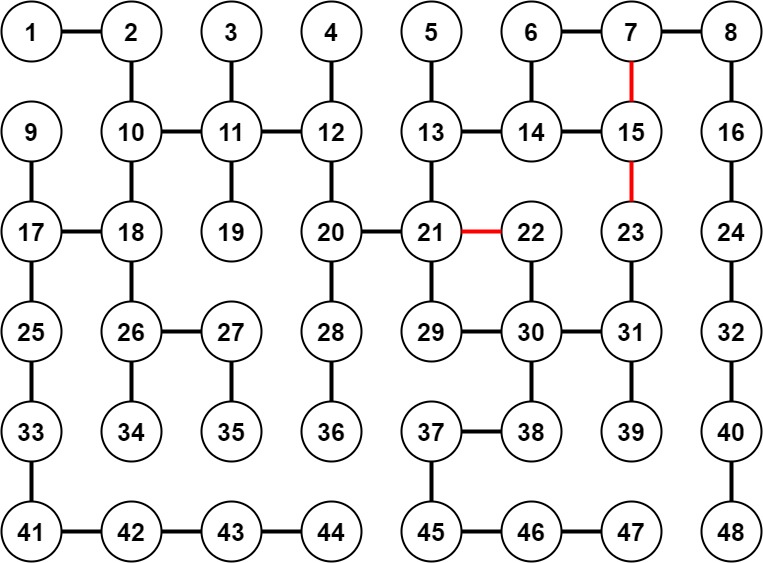
output: 없음

완전 미로 생성 프로그램에서 만든 8\*6 크기의 완전 미로를 바탕으로 만들어진 불완전 미로는 다음과 같다. 실습 결과에서 보인 완전 미로를 토대로 만들었다.

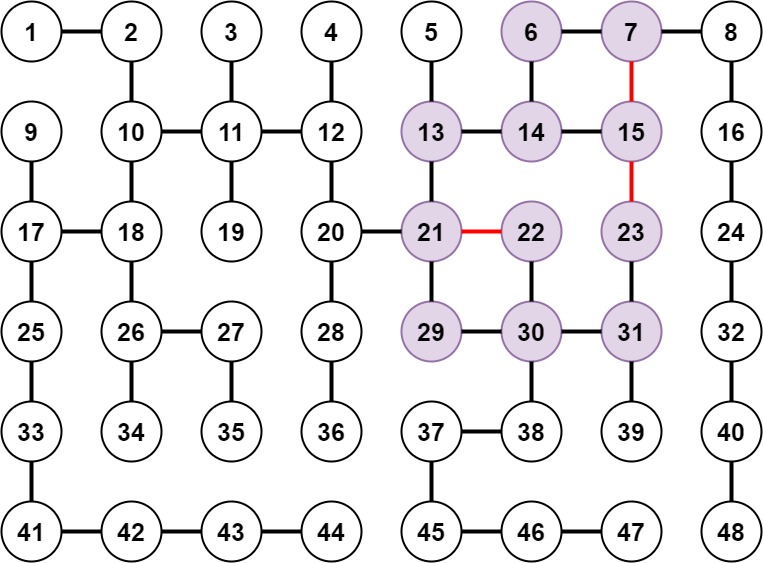


maze.maz가 완전 미로, imperfecMaze.maz가 불완전 미로이다. 두 파일을 cmp -l 명령어로 비교해보면 변경된 문자가 3개임을 알 수 있다. 50번째 바이트와 86번째 바이트가 가로 벽 ‘-‘이 공백 ‘ ‘로 변경되었고, 101번째 바이트가 세로 벽 ‘|’에서 공백 ‘ ‘로 변경되었다. 이로 인해 사이클이 발생하였다. 문제의 조건에 맞게 8, 6 중 작은 값 6을 2로 나눈 값에 해당하는 3개의 벽이 지워졌다.

불완전 미로를 그래프로 나타내면 다음과 같다. 완전 미로의 그래프에서 추가된 간선은 빨간색으로 표시했다.

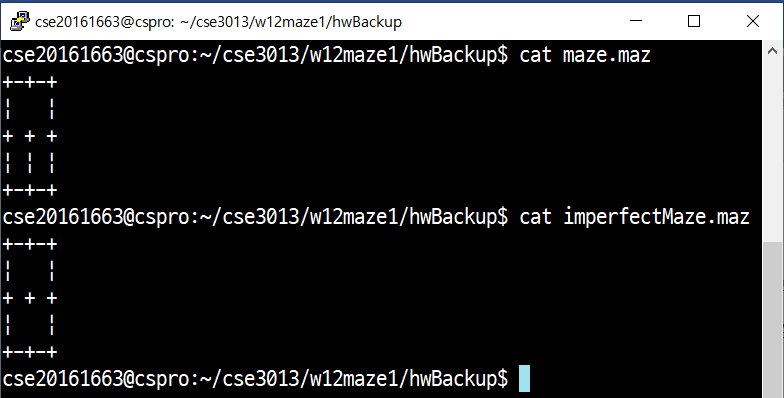


발생한 사이클을 표시하면 다음과 같다.



만약 WIDTH, HEIGHT 둘 중 하나라도 1일 경우 완전 미로를 그대로 유지한다.

WIDTH, HEIGHT 둘 다 2 이상일 경우 최소 크기의 미로 2\*2의 경우 불완전 미로는 다음과 같다.



유일하게 존재하던 벽 하나가 제거되어 벽이 존재하지 않는다.

기타

완전 미로의 경우, 그래프 자료구조를 이용하여 Prim’s Algorithm, Kruskal’s Algorithm을 이용하여 모든 벽이 허물어진 미로에서 벽을 추가해 나가는 방식으로 생성할 수 있다. 이는 그래프에서 이미 방을 나타내는 정점들끼리 간선으로 연결되어 있을 때 간선을 지워 나가 최소 개수의 간선으로 모든 정점들이 연결되도록 하는 신장 트리를 찾는 문제와 같다. 미로의 방과 방 사이 연결에 특별한 가중치가 존재하지 않을 경우, 모든 방과 방 사이의 연결의 값을 같다고 생각할 수 있고 그래프의 모든 간선의 비용이 같다고 생각할 수 있다. 이 경우 그래프의 모든 신장 트리가 최소 비용 신장 트리이므로 Prim’s Algorithm 또는 Kruskal’s Algorithm으로 임의의 최소 비용 신장 트리를 찾아내면 모든 신장 트리를 찾아낼 수 있다. 따라서 임의의 완전 미로를 생성할 수 있다.

일반적으로 가로로 방이 W개 있고, 세로로 방이 H개가 존재하는 미로를 나타내는 그래프에서 정점 사이의 간선의 개수는 최대 (W-1) \* H + W \* (H – 1) = 2\*WH – W – H 개이다. 최대 간선의 개수일 때 벽의 개수는 0개이고, 벽의 개수가 최대일 때 간선의 개수는 0개이다. 간선이 하나 생기면 벽은 1개 감소한다. 따라서 다음과 같은 관계가 성립한다.

(벽의 개수) + (간선의 개수) = 2\*WH – W - H