2020년도 2학기 컴퓨터공학설계및실험Ⅰ

14주차 미로(Maze) 3주차 예비보고서

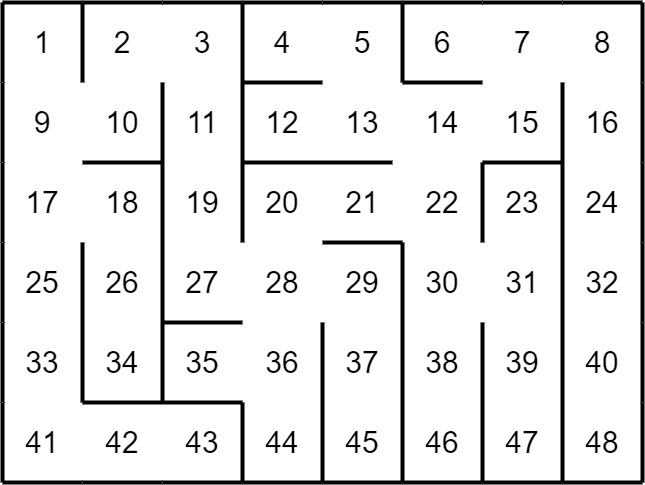
20161663 허재성

1. 실습 목적

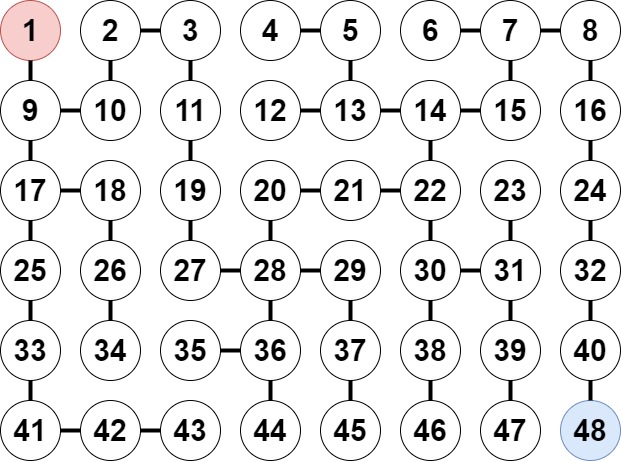
본 실험에서는 2주차의 실습에서 만든 OpenFrameWork 응용 프로그램을 수정하여 DFS 방법(실습)과 BFS 방법(과제)으로 미로를 탐색하여 그 결과를 화면에 표시하는 프로그램을 작성한다.

2. 관련 이론

(미로 찾기 문제에서의) DFS 알고리즘과 BFS 알고리즘의 시간 복잡도와 공간 복잡도를 계산하고 그 과정을 보이시오. 단 시간 복잡도와 공간 복잡도의 변수는 미로의 가로와 세로를 기준으로 한다.

12주차에서 생성하고 13주차에서 화면에 그린 미로는 완전 미로이므로 트리의 형태를 한다. 13주차에서 사용했던 다음 8\*6 미로를 고려해보자. 

위의 미로를 그래프로 만들면 다음과 같다. 빨간색 정점은 시작 정점인 1번방이고 파란색 정점은 미로에서 목적지인 마지막 방(48번 방)이다.



위의 그래프는 정점이 48개이고 간선이 47개이며 연결되지 않은 정점이 없으므로 트리이다. 따라서 시작점인 1방의 정점을 루트 노드로 하는 트리를 생각할 수 있다.

미로 찾기가 아닌 그래프의 모든 정점을 탐색하는 DFS를 생각해보자. 정점을 방문할 때 현재 정점 기준으로 탐색 순서를 방 번호가 큰 정점을 먼저 방문한다고 가정하자. 예를 들어 28번 정점에 위치할 경우, 연결된 정점은 20, 27, 29, 36이고 4개의 정점이 모두 아직 방문하지 않은 정점이라면 방문 순서의 우선 순위는 다음과 같다.

36, 29, 27, 20

미로의 방은 아래로 갈 때 줄이 바뀌며 방의 번호가 가장 커지고 오른쪽으로 갈 때 방의 번호가 1 증가한다. 왼쪽으로 갈 경우 방 번호가 1 감소하고 위로 갈 때 방 번호가 크게 감소한다. 따라서 이동 순서는 아래, 오른쪽, 왼쪽, 위 이다.

위의 우선 순위를 바탕으로 DFS의 정점의 방문 순서는 다음과 같다.

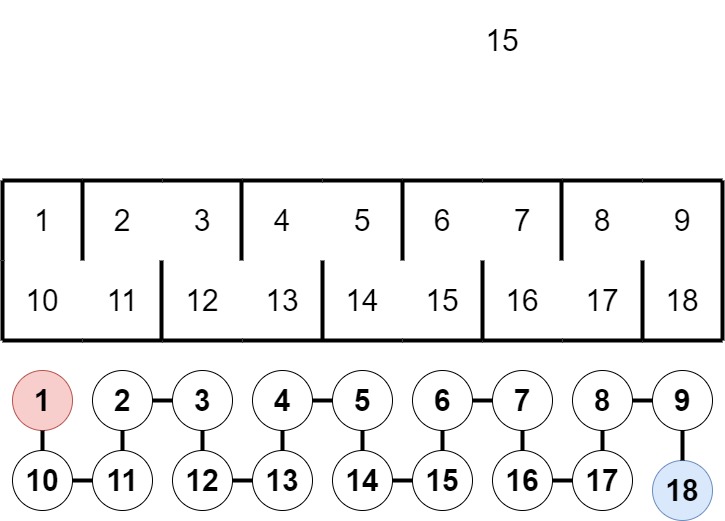
|  |
| --- |
| **1**-9-17-25-33-41-42-43-18-26-34-10-2-3-11-19-27-28-36-44-35-29-37-45-20-21-22-30-38-46-31-39-47-23-14-15-7-8-16-24-32-40-**48**-6-13-12-5-4 |

위와 같은 탐색 순서로 DFS를 수행하면 목적 정점인 48은 43번째에 도달하게 된다. 해당 미로에 맞게 탐색 우선순위를 바꾸면 더 빠르게 찾을 수도 있다. 만약 오른쪽 방을 가장 먼저 방문하고 위 방을 그 다음으로 방문하는 방법을 사용하면 48번 방까지 이동 과정은 다음과 같다. 목적점에 도달한 이후는 생략했다.

|  |
| --- |
| **1**-9-10-2-3-11-19-27-28-29-37-45-20-21-22-14-7-8-16-24-32-40-**48** |

23번째에 목적점에 도달했다. 위의 탐색 방법과 비교했을 때 훨씬 빨리 도달했다.

하지만 미로는 임의로 생성되므로 모든 미로에 통용되는 탐색 우선 순위는 존재하지 않는다.’



극단적인 예로 위와 같은 미로를 생각해보자. 위와 같은 미로의 경우 탐색의 우선 순위에 관계 없이 목적 정점인 18번 정점에 도달하기 위해선 모든 정점을 지나야 한다. 완전 미로의 경우 위와 같은 미로가 가능하다.

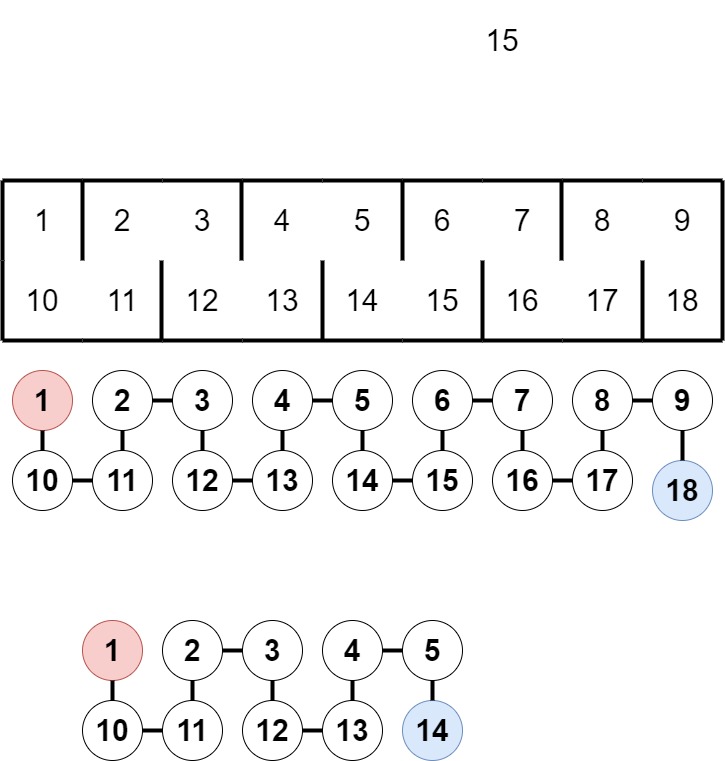
따라서 DFS 알고리즘의 최악의 경우 시간 복잡도는 O(정점의 개수)이다. 정점의 개수는 미로의 가로 방 개수 WIDTH와 세로 방 개수 HEIGHT의 곱이므로 시간 복잡도는 O(WIDTH\*HEIGHT)이다.

재귀함수의 호출 횟수는 트리의 최대 깊이에 비례한다. DFS를 재귀함수를 이용해 구현할 경우 트리가 위와 같이 skewed tree가 될 경우 트리의 노드의 개수만큼 재귀함수를 호출해야 한다. 함수가 자기 자신을 호출할 때, 함수의 인자(parameter)와 return address를 넘겨주기 위한 스택 메모리를 사용하는데 이 또한 공간 복잡도에 포함된다.[1] 따라서 공간 복잡도의 경우 최대 O(정점의 개수)=O(WIDTH\*HEIGHT)이다. 만약 정점의 개수가 커지게 되면 재귀함수의 호출 횟수가 많아져 호출 스택에서 이용 가능한 스택 메모리를 초과해서 사용하게 되어 스택 오버 플로(Stack Overflow)가 발생하여 프로그램이 충돌이 발생하게 된다.[2]

스택 오버플로를 피하기 위해 재귀 함수가 아닌 반복문과 스택 자료구조를 사용하여 DFS를 구현할 수 있다. 시작 정점을 방문 후 시작 정점을 스택에 삽입한다. 스택의 가장 최근에 삽입한 정점이 간선으로 연결된 정점 중 아직 방문하지 않은 정점이 있다면 해당 정점을 방문 후 스택에 삽입한다. 만약 스택의 가장 최근에 삽입한 정점에 연결된 정점 중 방문하지 않은 정점이 없다면 해당 정점을 스택에서 제거한다.

위의 과정을 스택이 빌 때까지 반복한다. 만약 중간에 목적 정점을 만나면 종료한다.

위의 스택을 이용한 DFS를 다음 그래프에 적용한다고 가정해보자. 방문 과정은 다음과 같다.

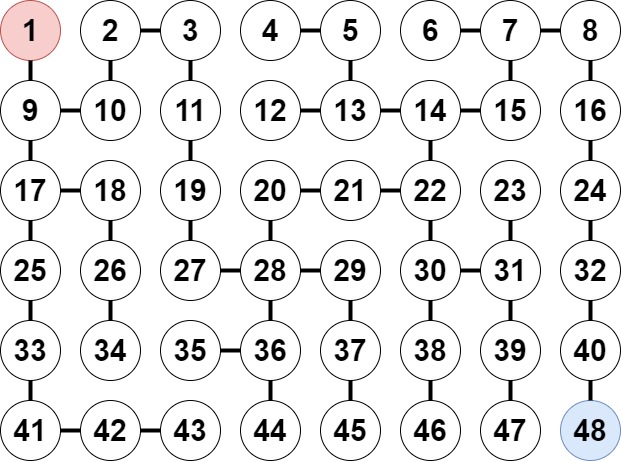


|  |
| --- |
| 1번 정점 방문 후 스택에 삽입->스택에 가장 최근에 쌓인 1번 정점에 인접한 10번 정점 방문 후 스택에 삽입-> 스택에 가장 최근에 쌓인 10번 정점에 인접한 11번 정점 방문 후 스택에 삽입->…-> 스택에 가장 최근에 쌓인 4번 정점에 인접한 5번 정점 방문 후 스택에 삽입-> 스택에 가장 최근에 쌓인 5번 정점에 인접한 14번 정점 방문 후->종료 |

위와 같은 그래프의 경우 DFS를 수행할 경우 목적 정점에 도달할 때까지 스택에 정점을 쌓기만 한다. 따라서 모든 정점이 스택에 쌓이게 되고 스택은 모든 정점을 담을 수 있을 만큼 커야 한다.

따라서 스택의 크기에 대한 공간 복잡도는 최대 O(정점의 개수)=O(WIDTH\*HEIGHT)이다.

DFS에서 시간 복잡도와 공간 복잡도는 모두 O(WIDTH\*HEIGHT)이다.



DFS에서와 같은 그래프에 BFS를 적용해보자. 탐색 우선 순위는 DFS와 같이 방 번호가 큰 방을 먼저 방문한다.

|  |
| --- |
| **1**->9->17->10->25->18->2->33->26->3->41->34->11->42->19->43->27->28->36->29->20->44->35->37->21->45->22->30->14->38->31->15->13->46->39->23->7->12->5->47->8->6->4->16->24->32->40->**48** |

목적 정점 48에 가장 마지막으로 도착하는 것을 알 수 있다. DFS와 같은 이유로 BFS의 시간 복잡도 또한 O(WIDTH\*HEIGHT)이다.

BFS는 정점을 방문 후 큐에 삽입한다. 큐에 삽입 후 큐에 가장 먼저 들어왔던 정점을 꺼내어 꺼낸 정점의 인접한 정점들 중 방문하지 않은 정점들을 방문 후 큐에 삽입한다. 따라서 큐에 삽입된 이전 정점들이 큐에서 제거되면서 제거된 정점의 인접 정점들이 삽입되므로 트리의 경우 이전 층(Level, depth)의 정점들이 큐에서 제거될 때 다음 층의 정점들이 모두 큐에 삽입된다. 이와 같은 경우 큐의 최대 크기는 트리의 최대 폭(트리의 각 층의 노드의 개수 중 최대 개수)가 된다.

DFS에서는 스택의 최대 크기를 정점의 개수 로 쉽게 알 수 있었지만, 미로가 임의로 생성되므로 트리의 최대 폭은 예측하기 어렵다. 따라서 필요한 큐의 크기를 정확히 알기는 어렵다.

정점을 구현하는 자료구조에서 각 정점은 최대 4개의 정점과 연결될 수 있다. 루트 노드를 시작 정점이라 하면 시작 정점은 최대 2개의 정점과 연결될 수 있고 나머지 정점들은 최대 4개의 정점과 연결될 수 있으며 트리의 경우 그 중 1개는 루트 노드로부터 연결된 부모 노드와 연결되어 있으므로 하나의 노드가 자식 노드로 가질 수 있는 노드의 개수는 최대 3개이므로 3진 트리로 생각할 수 있다.

꽉찬 3진 트리(단말 노드를 제외한 모든 노드의 자식 노드의 개수가 3개)의 경우 루트 노드의 깊이를 0이라 했을 때 깊이 d일 때의 노드의 개수는 3d개이며 트리의 모든 노드의 개수는 1+31+32+…+3d=(3d+1-1)/(3-1) = WIDTH\*HEIGHT이다. 따라서 3d=(2\*WIDTH\*HEIGHT+1) / 3이다. 이 때 트리의 최대폭은 3d이므로 최대 폭은 O(WIDTH\*HEIGHT)가 된다. 미로의 그래프가 꽉찬 3진 트리가 될 수는 없으므로 최대폭은 꽉찬 3진 트리일 경우보다 작다. 따라서 큐의 크기의 상한은 O(WIDTH\*HEIGHT)이다. 따라서 BFS의 공간 복잡도는 O(WIDTH\*HEIGHT)이다.

3. 실습 방법

3-1. DFS 알고리즘으로 이 문제를 어떻게 해결할지 서술하시오.

2에서 언급한 것처럼 재귀 함수를 이용한 DFS의 구현은 미로의 크기가 커지면 스택 오버 플로우로 인하여 프로그램이 정상적으로 작동하지 않을 수 있다. 따라서 반복문과 스택 자료구조를 이용하여 DFS를 구현한다.

13주차에서 구현한 정점 구조체인 Vertex 구조체를 원소로 하는 스택을 정의해야 한다.

|  |
| --- |
| typedef struct \_Vertex {  int name;  float x, y;  bool visited;  \_Vertex \*up, \*down, \*left, \*right;  } Vertex; |

Vertex 구조체를 저장할 스택을 배열로 저장할 경우, 정점의 개수 WIDTH\*HEIGHT를 크기로 가지는 배열을 생성해야 한다. 하지만 트리의 최대 깊이가 WIDTH\*HEIGHT보다 훨씬 얕을 경우 이와 같은 배열은 공간 낭비가 된다. 따라서 연결 리스트를 이용해 스택을 구현한다.

연결 리스트의 노드는 다음과 같이 정의할 수 있다.

|  |
| --- |
| typedef struct \_StackNode {  Vertex elm;  \_StackNode\* next;  } StackNode; |

노드의 변수 중 elm은 정점 구조체 Vertex를 저장하는 변수이다.

연결 리스트를 이용해 스택을 구현할 경우 트리의 최대 깊이만큼 StackNode를 할당하면 되므로 공간 복잡도가 최소화된다. 또한 스택에 삽입과 삭제 모두 O(1) 시간에 이루어진다. 따라서 연결 리스트로 구현한 스택을 이용할 경우 시간 복잡도는 최대 O(WIDTH\*HEIGHT)가 된다.

반복문과 스택을 이용한 DFS의 의사 코드는 다음과 같다.

|  |
| --- |
| void dfs(Vertex v, Vertex target)  { /\* 탐색을 시작할 정점 v, 목표 정점 target \*/  StackNode\* top = NULL; // 스택 초기화  정점 v를 방문;  push(top, v); // 스택에 삽입  while(!stackEmpty(top)) {  if(top->elm과 target이 같은 정점일 경우) return;  if(top->elm의 인접 정점 중 방문하지 않은 인접 정점이 있을 경우) {  u = 아직 방문하지 않은 인접 정점;  u를 방문함;  }  else pop(top); // 경로에 포함되지 않음  } |

만약 어떤 정점을 방문하여 스택에 쌓였는데 해당 정점의 인접 정점이 존재하지 않을 경우, 해당 방은 막다른 방에 해당한다. 또한 정점의 인접 정점 중 방문하지 않은 정점이 없다면 해당 방은 미로의 경로에 포함되지 않는다. 만약 이 방까지 올 동안 목적방이 존재했다면 이미 목적 방을 찾았으므로 DFS는 종료되었을 것이다. 따라서 해당 방은 시작 방에서 목적 방까지의 경로에 포함되지 않는다. 따라서 스택에서 제거된 방들은 미로 찾기 과정에서 방문은 했으니 경로에 포함되지 않는, 방문하지 않았어도 되는 방임을 알 수 있다.

목적 방에 도달했을 때 스택에 남아 있는 방만이 경로에 포함되는 방이다.

따라서 스택에서 제거된 방들과 스택에 포함된 방들을 분류하여 미로에 다른 색으로 표시하면 시작 방에서 목적 방까지의 경로와 방문했지만 포함되지는 않는 방을 알 수 있다.

완전 미로이므로 시작점과 목적점까지의 경로는 유일하다. 경로 이외의 방을 방문할 경우 경로 이외의 방들은 목적점에 도달하기 위해 필수적으로 방문하지 않아도 되는 불필요한 방들이다.

3-2. BFS 알고리즘으로 이 문제를 어떻게 해결할지 서술하시오.

반복문과 큐 자료구조를 이용하여 BFS를 구현한다. Vertex 구조체를 저장할 큐를 배열로 저장할 경우, 정점의 개수 WIDTH\*HEIGHT를 크기로 가지는 배열을 생성하면 공간이 부족할 일은 없다. 하지만 트리의 최대 폭은 WIDTH\*HEIGHT보다 항상 작으므로 이와 같은 배열은 공간 낭비가 발생한다. 스택과 비슷한 방법으로 연결 리스트를 이용해 큐를 구현한다.

연결 리스트의 노드는 다음과 같이 정의할 수 있다.

|  |
| --- |
| typedef struct \_QueueNode {  Vertex elm;  \_QueueNode\* next;  } QueueNode; |

트리의 최대 폭 만큼만 연결 리스트의 노드가 큐에 저장되므로 연결 리스트를 이용해 큐를 구현할 경우 트리의 최대 폭만큼 QueueNode를 할당하면 되므로 공간 복잡도를 최소화할 수 있다. 또한 큐에 삽입과 삭제 모두 O(1) 시간에 이루어진다. 따라서 연결 리스트로 구현한 큐를 이용할 경우 시간 복잡도는 최대 O(WIDTH\*HEIGHT)가 된다.

반복문과 큐를 이용한 BFS의 의사 코드는 다음과 같다.

|  |
| --- |
| void bfs(Vertex v, Vertex target)  { /\* 탐색을 시작할 정점 v, 목표 정점 target \*/  QueueNode\* front = NULL; // 큐 초기화  QueueNode\* rear = NULL;;  정점 v를 방문함;  enque(rear, v); // 큐에 삽입  While(!queueEmpty(front)) {  u = deque(front) // 큐에서 제거한 정점 u;  if(u가 target일 경우) return;  for(u의 모든 인접 정점 w에 대하여) {  if(정점 w를 아직 방문하지 않음) {  정점 w를 방문함;  enque(rear, w);  }  }  }  } |

큐에서 제거된 정점들은 이미 방문한 정점들이다. 또한 목표 정점에 도달했을 때 큐에 남아 있는 정점들은 방문했지만 트리에서 도달한 정점의 부모 정점과 형제 정점이므로 경로에 포함되지 않는다. 또한 큐에서 제거된 정점들 중에서도 도달한 정점으로부터 루트 노드까지의 조상 정점을 제외한 모든 정점은 포함되지 않는다.

따라서 큐에 남은 정점과 제거된 정점 중 루트 노드로부터 목적 노드까지 경로에 포함되지 않은 정점은 미로에 방문했지만 경로에 포함되지 않는 정점으로 표시하고, 그 외 제거된 정점들은 경로에 포함된 정점이므로 따로 표시한다.

Vertex 노드에서 부모 노드(BFS 과정에서 현재 노드로 오기 이전 노드)를 알 수 있으면 경로를 알 수 있다.

4. 기타

불완전 미로의 경우, 사이클이 존재하여 시작점과 끝 점 사이의 경로가 하나가 아닐 수 있다. 이 경우 여러 개의 경로 중 지나는 방의 수가 최소인 경로를 최소 경로로 생각할 수 있다.

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, “스택 오버플로”, https://ko.wikipedia.org/wiki/%EC%8A%A4%ED%83%9D\_%EC%98%A4%EB%B2%84%ED%94%8C%EB%A1%9C