

알고리즘 과제(09)

201404377 진승언

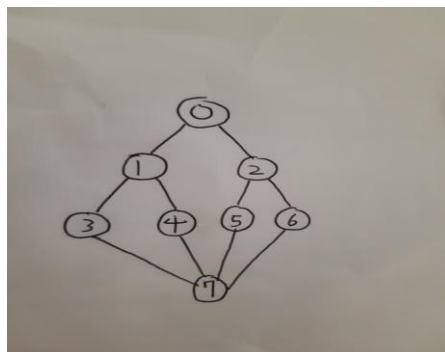
1. 과제 목표 및 해결 방법

➔ 이번 과제는 그래프에서의 너비우선탐색과(BFS) 깊이우선탐색(DFS)를 구현하고 vertex들의 탐색시작시간과 완료시간을 도출해내는 것이 목표였다. 그래프에는 인접 행렬을 이용한 방법과 인접 리스트를 이용한 방법 그리고 인접 배열과 해시테이블을 이용하는 방법이 있는데 그 중에서 인접 행렬을 이용한 방법으로 구현하였다.

루트를 시작으로 탐색을 한다면 너비우선탐색은 먼저 루트의 자식을 차례로 방문한다. 다음으로 루트 자식의 자식, 즉 루트에서 두 개의 간선을 거쳐서 도달할 수 있는 정점을 방문한다. 이에 반해 깊이 우선 탐색은 루트의 자식 정점을 하나 방문한 다음 아래로 내려갈 수 있는 곳까지 내려가고 더 이상 내려갈 수가 없으면 위로 되돌아오다가 내려갈 곳이 있으면 즉각 내려간다. 이러한 특징을 생각하고 구현하였다. 그리고 색깔을 이용해서 방문한 vertex인지 구분할 수 있게 하였다. 예를 들어 white는 아직 방문하지 않은 경우이고 gray는 방문한 vertex이고 black은 방문하고 탐색을 끝마친 vertex로 하였다.

이번 과제는 입력파일이 인접행렬이므로 인접행렬을 이용해 구현하였다. 이차원 배열로 표현되는 인접행렬을 이용해서 서로 인접한 vertex인지 확인 할 수 있었다. 행과 열은 각각의 vertex를 의미하므로 vertex크기와 같은 개수의 행과 열로 되어있고 값이 1이면 인접하다는 뜻이고 0이면 인접하지 않다는 뜻이다. 이것을 생각해서 인접한 vertex들을 찾아가면서 구현하면 되었다.

```
0 1 1 0 0 0 0 0
1 0 0 1 1 0 0 0
1 0 0 0 0 1 1 0
0 1 0 0 0 0 0 1
0 1 0 0 0 0 0 1
0 0 1 0 0 0 0 1
0 0 1 0 0 0 0 1
0 0 1 0 0 0 0 1
0 0 0 1 1 1 1 0
```



이번 입력파일 인접행렬이다. 이것을 그림으로 그려놓고 하면 더 이해하기 쉬웠다.

2. 주요 부분 코드 설명(알고리즘 부분 코드 캡처)

```
2 public class Vertex {
3
4     final int INF = Integer.MAX_VALUE;
5     int id;
6     String color;
7     int distance; //도착시간(탐색하는데 걸린시간)
8     int f; //빠져나온 시간(DFS에서 쓰임)
9     Vertex parent;
10
11 Vertex(){
12     color = "WHITE";
13     distance = INF;
14     parent = null;
15 }
16
17 Vertex(int id){
18     this.id = id;
19     color = "WHITE";
20     distance = INF;
21     parent = null;
22 }
23
24 public void init() { //vertex 초기화
25     color = "WHITE";
26     distance = INF;
27     parent = null;
28 }
29 }
30
```

정점(Vertex) 구현한 코드이다. 초기 색은 WHITE, distance(탐색시간)은 INF값, parent는 (해당 vertex를 방문 하거나 인접vertex) null로 초기화 하였다. 그리고 vertex를 생성할 때 id값을 지정할 수 있게 하였다.

그리고 init()함수로 해당 vertex를 초기화 할 수 있게 해주었다.

```
// BFS 수행
BFS bfs = new BFS();
System.out.println("BFS의 시작 vertex를 입력하시오 (정해진 vertex크기보다 작게입력하시오)");
Scanner sc = new Scanner(System.in);
startVertex = sc.nextInt();

bfs.BFSrun(graphMatrix, vertex, startVertex); // 맨 마지막 매개변수에 탐색을 시작할 vertex를 설정가능, 0으로하였다.
for (i = 0; i < vertex.length; i++) {
    System.out.println("id값: " + vertex[i].id + " 탐색횟수: " + vertex[i].distance);
}
}
```

BFS를 할 때는 탐색시작 vertex를 입력 받을 수 있게 구현하였다.

```

1 import java.util.LinkedList;
2
3 public class BFS {
4     final int INF = Integer.MAX_VALUE;
5
6     void BFSrun(int[][] G, Vertex[] vertex, int startVertex) {
7         vertex[startVertex].color = "GRAY"; //시작 vertex 값 설정
8         vertex[startVertex].parent = null;
9         vertex[startVertex].distance = 0;
10
11         Queue<Vertex> q = new LinkedList<Vertex>();
12         q.add(vertex[startVertex]); //시작인덱스 큐에 넣어들
13         while(!q.isEmpty()) { //큐가 빌때까지 반복
14             Vertex u = q.poll(); //u에 큐에서 빼온 값 저장
15             for(int column = 0 ; column < vertex.length; column++) { //큐에서 빼온 vertex의 인접한 vertex들 값 설정 하고 큐에 넣어줄(모든 vertex
16                 Vertex v = vertex[column];
17                 if(isAdj(u, v, G) && v.color.equals("WHITE") && (v.distance == INF)) { //인접하고 색이 흰색이고 (방문색안함) 탐색하지않은(INF) vertex인 경우
18                     v.color = "GRAY"; //회색으로 변경
19                     v.distance = u.distance + 1; //탐색시간 저장
20                     v.parent = u; //인접 노드들끼리 연결해줄
21                     q.offer(v); //새로 탐색한 vertex를 큐에 삽입
22                 }
23             }
24             u.color = "BLACK"; //완료한 vertex 색 black으로 변경
25         }
26     }
27
28     boolean isAdj(Vertex u, Vertex v, int[][] graphMatrix) {
29         int tmp = graphMatrix[u.id][v.id]; //v와 u vertex가 인접하면 1, 아니면 0
30         if(tmp == 0) { //인접 아닌 경우 false반환
31             return false;
32         }
33         else { //인접 vertex인 경우 true 반환
34             return true;
35         }
36     }
37 }

```

BFS구현한 코드이다.(주석에 자세히 써놨습니다.) 먼저 탐색시작 vertex의 색과 distance를 각각 GRAY와 0으로 저장해주는 걸로 탐색을 시작준비를 한다. 그 후 시작 vertex를 큐에 넣어주고 큐가 비게 될 때까지 반복문을 돌린다. 반복문안에서는 큐에 들어진 vertex를 뽑아 이 vertex와 인접한 vertex를 모두 조사하고 vertex 값을 변경주고 해당 vertex를 큐에 넣어주게 반복하게 구현하였다. (큐에 들어있는 vertex는 회색이고 저장한 vertex를 뽑아와서 그 vertex기준으로 다시 너비우선탐색을 계속해서 진행하면 된다.) 인접한 vertex 인지는 isAdj함수를 이용해 알 수 있다. isAdj함수는 인접행렬을 이용해서 인접한지 알려주게 구현하였다. 인접행렬 값이 0이면 인접하지 않고 1이면 인접 하다는 뜻이다. 만약 인접한 vertex이고 해당 vertex가 아직 방문하지 않은 WHITE이며 distance가 INF인 경우 (distance까지 조사할 필요가 없긴 하다.) 해당 vertex의 색을 GRAY로 하고 알맞은 distance값을 저장하고 parent에 이전의 인접vertex를 저장해 이어주었다.

이런 식으로 한 vertex의 근접한 vertex들을 찾고 이 근접한 vertex들 중 하나를 기준으로 이것과 인접한 vertex를 찾아가면서 반복해 해결하게 하면 되었다.

```

public class DFS {
    final int INF = Integer.MAX_VALUE;
    int time;

    public void DFSrun(int[][] graphMatrix, Vertex[] vertex) {
        time = 0;
        for(int i = 0; i < vertex.length; i++) {
            if(vertex[i].color.equals("WHITE")) {
                visit(graphMatrix, vertex[i], vertex);
            }
        }
    }

    public void visit(int[][] graphMatrix, Vertex u, Vertex[] vertex) {
        time = time + 1; //처음 도착시간은 1부터 시작하고 1씩 더해짐
        u.distance = time;
        u.color = "GRAY";
        Queue<Integer> q = new LinkedList<Integer>();
        int row = u.id;
        for(int column = 0; column < vertex.length; column++) { //모든 vertex와 비교(인접한지)

            if(graphMatrix[row][column] != INF && graphMatrix[row][column] != 0) { //아직 탐색하지 않고 인접한 경우 큐에 인접한 vertex 삽입
                q.offer(column);
            }
        }

        for(int i : q) { //큐에 들어진 vertex를 visit 재귀
            Vertex v = vertex[i];
            if(v.color.equals("WHITE")) {
                v.parent = u;
                visit(graphMatrix, v, vertex);
            }
        }

        //탐색완료한 vertex색을 black으로 변경 및 빠져나온 시간(완료시간) 저장
        u.color = "BLACK";
        time = time + 1;
        u.f = time;
    }
}

```

DFS 구현코드이다. 과제대로 0번 vertex를 시작점으로 하였다. 먼저 DFSrun함수에서 time을 0으로 초기화 해주고 모든 탐색하지 않은 vertex를 차례대로 visit함수를 호출해준다. (코드에는 확인 차 for문으로 돌려봤으나 시작점으로 할 vertex만 넘겨줘도 된다.)

visit 함수에서는 먼저 방문했으니 time을 1증가시키고 해당 vertex의 탐색시작 시간을 저장하고 색을 GRAY로 저장해준다. 그리고 해당 vertex와 나머지 vertex들을 조사해 인접하고 아직 방문하지 않은 vertex의 id값 즉 행, 열 값을 큐에 저장해준다. 그 후 큐에 저장된 vertex를 하나씩 뽑아와서 방문하지 않은 vertex인 경우(WHITE색) 해당 vertex를 이전 인접vertex와 연결해주고(parent) 해당 다른 큐의 값을(vertex) 반복문으로 또 꺼내오기 전에 큐에서 읽어온 이 vertex값을 visit매개변수로 하여 visit함수를 재귀호출 해준다. 이렇게 하면 깊이우선탐색이 가능하다. 마지막으로는 탐색을 완료한 vertex 색을 BLACK으로 바꿔주고 빠져 나온 시간(f)를 저장해주면 된다.

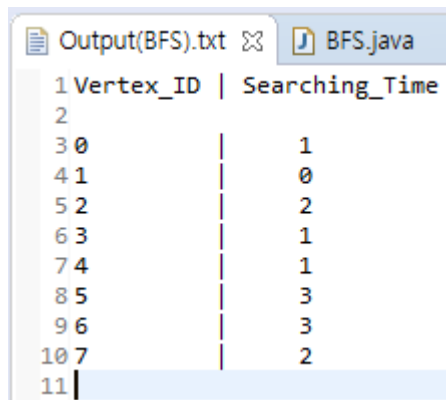
결과(시간 복잡도 포함)

콘솔 출력결과 화면

```
BFS의 시작 vertex를 입력하시오 (정해진 vertex크기보다 작게입력하시오)
1
id값:0 탐색횟수: 1
id값:1 탐색횟수: 0
id값:2 탐색횟수: 2
id값:3 탐색횟수: 1
id값:4 탐색횟수: 1
id값:5 탐색횟수: 3
id값:6 탐색횟수: 3
id값:7 탐색횟수: 2
=====
DFS 결과( DFS는 0부터 탐색 고정)
id값:0 도착시간: 1 빠져나온 시간: 16
id값:1 도착시간: 2 빠져나온 시간: 15
id값:2 도착시간: 8 빠져나온 시간: 11
id값:3 도착시간: 3 빠져나온 시간: 14
id값:4 도착시간: 5 빠져나온 시간: 6
id값:5 도착시간: 7 빠져나온 시간: 12
id값:6 도착시간: 9 빠져나온 시간: 10
id값:7 도착시간: 4 빠져나온 시간: 13
=====
```

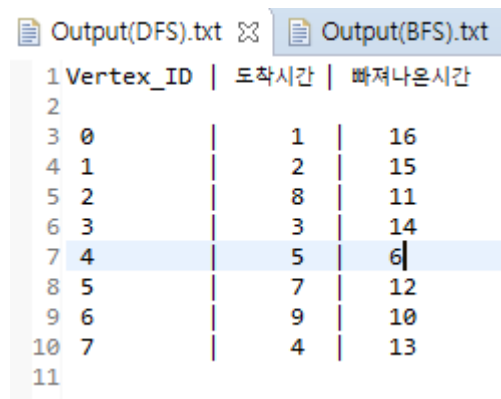
1을 입력 받아서 1인 vertex를 시작정점으로 했을때의 결과이다. (DFS는 시작vertex 0으로 고정)

BFS 출력파일 결과화면



| 1 Vertex_ID | Searching_Time |
|-------------|----------------|
| 2 | |
| 3 0 | 1 |
| 4 1 | 0 |
| 5 2 | 2 |
| 6 3 | 1 |
| 7 4 | 1 |
| 8 5 | 3 |
| 9 6 | 3 |
| 10 7 | 2 |
| 11 | |

DFS 출력파일 결과화면



| 1 Vertex_ID | 도착시간 | 빠져나온시간 |
|-------------|------|--------|
| 2 | | |
| 3 0 | 1 | 16 |
| 4 1 | 2 | 15 |
| 5 2 | 8 | 11 |
| 6 3 | 3 | 14 |
| 7 4 | 5 | 6 |
| 8 5 | 7 | 12 |
| 9 6 | 9 | 10 |
| 10 7 | 4 | 13 |
| 11 | | |

⇒ ppt와 시간이 똑같진 않지만 그림을 그려 확인해본 결과 인접한 vertex들 중에서 우선순위가 탐색만 다를 뿐이지 BFS, DFS 모두 잘 구현됨을 볼 수 있었다.

V가 정점의 수라고 한다면, V개의 정점을 가지는 그래프를 인접 행렬로 표현하기 위해서는 간선의 수에 무관하게 항상 n^2 의 메모리 공간이 필요하다. 이에 따라 인접 행렬은 간선이 많이 존재하는 밀집 그래프를 표현하는 경우에 적합하다. 적은 숫자의 간선만을 가지는 희소 그래프의 경우에는 메모리의 낭비가 크므로 적합하지 않다.

인접 행렬을 이용하면 두 정점을 연결하는 간선의 존재 여부를 $O(1)$ 시간 안에 즉시 알 수 있다는 장점이 있다. 또한 정점의 차수는 인접 행렬의 행이나 열을 조사하면 알 수 있으므로 $O(v)$ 의 연산에 의해 알 수 있다. 반면에 그래프에 존재하는 모든 간선의 수를 알아내려면 인접 행렬 전체를 조사해야 하므로 V^2 번의 조사가 필요하게 되어 $O(V^2)$ 의 시간이 요구된다.