

실습보고서

2020. 12. 01 (화)

[알고리즘 00분반]_13주차
201501513_김성현

조건사항

운영체제 : MAC OS

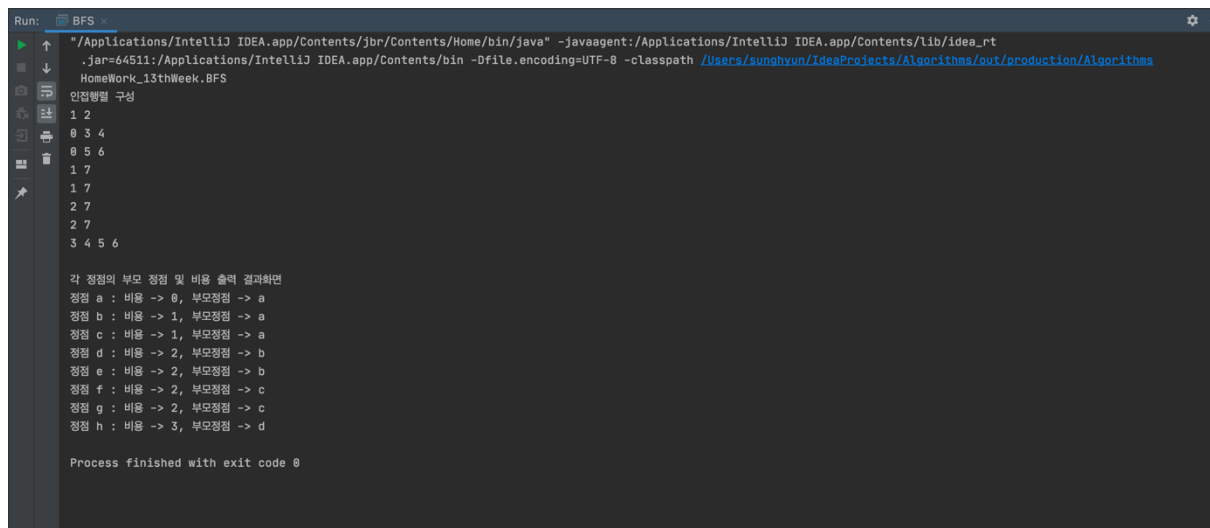
IDE : IntelliJ

라이브러리 : io(입출력), util(코드설계), nio(파일읽기)

Error 사항) 파일을 불러올 때 상대경로로 "./"를 주어서 같은 패키지 내가 아닌 같은 디렉토리에
서 파일을 읽습니다.

1. Adjacent list를 이용하여 BFS를 구현하라

<실행 결과>



```
Run: BFS
"/Applications/IntelliJ IDEA.app/Contents/jbr/Contents/Home/bin/java" -javaagent:/Applications/IntelliJ IDEA.app/Contents/lib/idea_rt
.jar=64511:/Applications/IntelliJ IDEA.app/Contents/bin -Dfile.encoding=UTF-8 -classpath /Users/sunghyun/IdeaProjects/Algorithms/out/production/Algorithms
HomeWork_13thWeek.BFS
인접행렬 구성
1 2
0 3 4
0 5 6
1 7
1 7
2 7
2 7
3 4 5 6

각 정점의 부모 정점 및 비용 출력 결과화면
정점 a : 비용 -> 0, 부모정점 -> a
정점 b : 비용 -> 1, 부모정점 -> a
정점 c : 비용 -> 1, 부모정점 -> a
정점 d : 비용 -> 2, 부모정점 -> b
정점 e : 비용 -> 2, 부모정점 -> b
정점 f : 비용 -> 2, 부모정점 -> c
정점 g : 비용 -> 2, 부모정점 -> c
정점 h : 비용 -> 3, 부모정점 -> d

Process finished with exit code 0
```

<코드 설명>

```

int node_number = Integer.parseInt(str[0]); //점 개수
graph[] graphs = new graph[node_number];
path = new int[node_number][];

for(int i = 1; i < str.length; i++){
    String[] split = str[i].split( regex: "\\");
    ArrayList<Integer> list = new ArrayList<>();
    for(int j = 0; j < split.length; j++){
        int edge = Integer.parseInt(split[j]);
        if(edge == 1){
            list.add(j);
        }
    }

    path[i-1] = new int[list.size()];
    for(int k = 0; k < list.size(); k++){
        path[i-1][k] = list.get(k);
    }
}

System.out.println("인접행렬 구성");
print(path);

```

먼저 입력받은 파일을 String 배열로 만들어 파라미터로 받은 solution메소드에서 점의 개수와 각 간선들의 위치를 파악한 다음 adjacent list 형태로 저장하기 위해 ArrayList를 사용하여 각 정점에서 연결된 정점들의 정보들을 담아주었습니다.

```

for(int i = 0; i < node_number; i++){
    graphs[i] = new graph(i, color: "White");
}

graphs[0].color = "Gray";
graphs[0].depth = 0;
graphs[0].parents = 0;

BFS(graphs);

```

그리고 초기값을 설정하기 위해 graph 배열에서 0부터 정점들의 개수만큼 "White"로 설정해주고 초기 root vertex인 0번째 graph의 초기값을 재설정해준 다음 BFS를 통해 그래프 탐색을 실행하였습니다.

```

public void BFS(graph[] graphs){
    Queue<graph> que = new LinkedList<>();
    que.offer(graphs[0]);
    while(!que.isEmpty()){
        graph g = que.poll();
        int vertex = g.vertex;
        for(int i = 0; i < path[vertex].length; i++){
            int edge = path[vertex][i];
            graph h = graphs[edge];
            if(h.color.equals("White")) {
                h.parents = g.vertex;
                h.depth = g.depth + 1;
                h.color = "Gray";

                que.offer(h);
            }
        }
        g.color = "Black";
    }

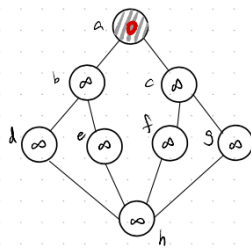
    System.out.println("\n각 정점의 부모 정점 및 비용 출력 결과화면");
    for(int i = 0; i < graphs.length; i++){
        System.out.println(graphs[i]);
    }
}

```

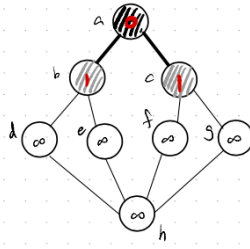
BFS 메소드에서는 Queue를 선언해주고 parameter로 받은 root vertex를 Queue에 초기 저장해주고 미리 설정해둔 Adjacent list에서 출발 정점에서 연결된 정점들을 모두 읽어 값을 재설정해주고 white - gray - black 순으로 변화를 나타냈습니다. 그리고 깊이는 que에서 deque한 값에서 +1을 해주어 탐색 순서에 구분을 주었고 부모 역시 deque한 정점의 번호를 저장해두었습니다.

< 탐색 경로 작성 >

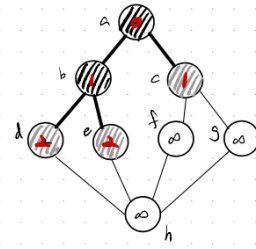
BFS



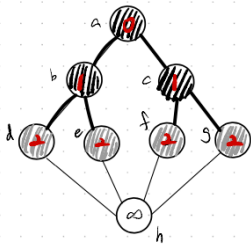
Q a
0



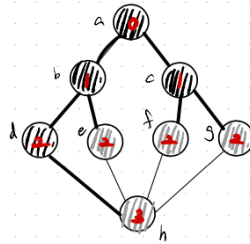
Q b c
1 1



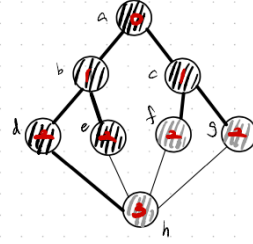
Q c d e
1 2 2



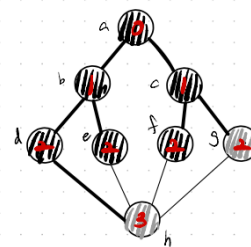
Q d e f g
2 2 2 2



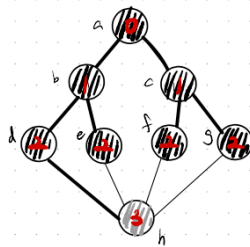
Q e f g h
2 2 2 3



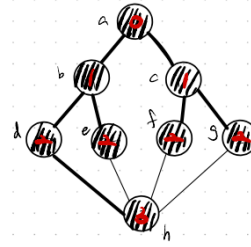
Q f g h
2 2 3



Q g h
2 3



Q h
3



Q ∅

오른쪽방향으로 실행순서에 따라 변화를 나타내었습니다.

2. Adjacent list를 이용하여 DFS를 구현하라

<실행 결과>

```
Run: DFS
"/Applications/IntelliJ IDEA.app/Contents/jbr/Contents/Home/bin/java" -javaagent:/Applications/IntelliJ IDEA.app/Contents/lib/idea_rt
.jar=64179:/Applications/IntelliJ IDEA.app/Contents/bin -Dfile.encoding=UTF-8 -classpath /Users/sunghyun/IdeaProjects/Algorithms/out/production/Algorithms
HomeWork_13thWeek.DFS
인접행렬 구성
1 2
0 3 4
0 5 6
1 7
1 7
2 7
2 7
3 4 5 6

각 정점의 부모 정점 및 발견된 시간, 탐색완료 시간 출력 결과
정점 0 : 최초 발견 시간 -> 1, 탐색완료 시간 -> 16, 부모정점 -> a
정점 1 : 최초 발견 시간 -> 2, 탐색완료 시간 -> 15, 부모정점 -> a
정점 2 : 최초 발견 시간 -> 8, 탐색완료 시간 -> 11, 부모정점 -> f
정점 3 : 최초 발견 시간 -> 3, 탐색완료 시간 -> 14, 부모정점 -> b
정점 4 : 최초 발견 시간 -> 5, 탐색완료 시간 -> 6, 부모정점 -> h
정점 5 : 최초 발견 시간 -> 7, 탐색완료 시간 -> 12, 부모정점 -> h
정점 6 : 최초 발견 시간 -> 9, 탐색완료 시간 -> 10, 부모정점 -> c
정점 7 : 최초 발견 시간 -> 4, 탐색완료 시간 -> 13, 부모정점 -> d

Process finished with exit code 0
|
```

<코드 설명>

```
int vertex_number = Integer.parseInt(str[0]);

vertex[] vertices = new vertex[vertex_number];
for(int i = 0; i < vertex_number; i++){
    vertices[i] = new vertex(i, color: "White");
}

path = new int[vertex_number][];

for(int i = 1; i < str.length; i++){
    String[] split = str[i].split( regex: " ");
    ArrayList<Integer> list = new ArrayList<>();
    for(int j = 0; j < split.length; j++){
        int edge = Integer.parseInt(split[j]);
        if(edge == 1){
            list.add(j);
        }
    }

    path[i-1] = new int[list.size()];
    for(int k = 0; k < list.size(); k++){
        path[i-1][k] = list.get(k);
    }
}
```

초기 설정에서는 BFS와 마찬가지로 data13.txt에서 읽은 문자열 배열을 파라미터로 받아와 정점의 개수와 각 간선들을 ArrayList를 생성해 Adjacent list를 생성하여 path란 String 2차원 배열에 재저장을 해주었습니다.

```

time = 0;
System.out.println("인접행렬 구성");
print(path);
System.out.println("\n각 정점의 부모 정점 및 발견된 시간, 탐색완료 시간 출력 결과");
for(int i = 0; i < vertices.length; i++){
    if(vertices[i].color.equals("White")){
        DFS(vertices, vertices[i]);
    }
}

for(int i = 0; i < vertices.length; i++){
    System.out.println(vertices[i]);
}

```

인접행렬을 구성한 다음 발견된 시간과 완료된 시간을 나타내기 위해 전역변수로 time이라는 변수를 주어 모든 정점들을 확인하기 위해 반복문을 정점 개수만큼 실행시킨 다음 연관된 정점의 색이 White인 경우에만 DFS를 통해 탐색 되도록 하였습니다.

```

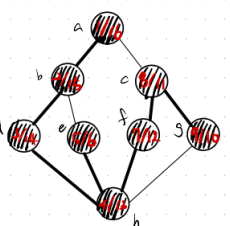
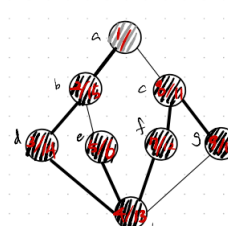
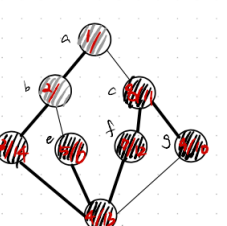
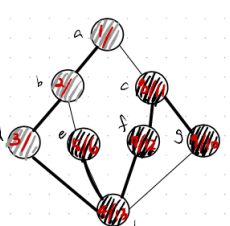
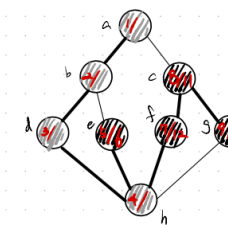
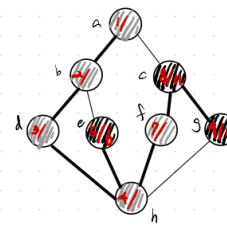
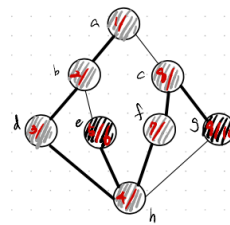
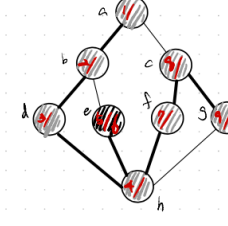
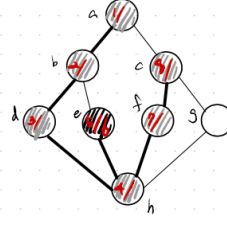
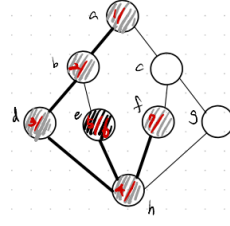
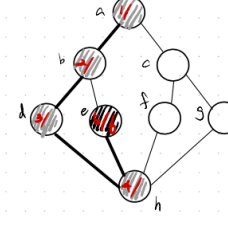
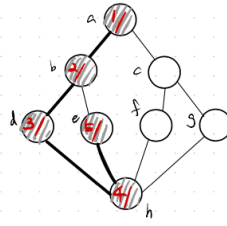
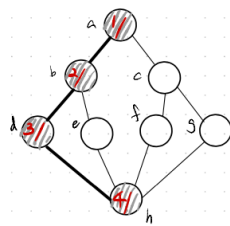
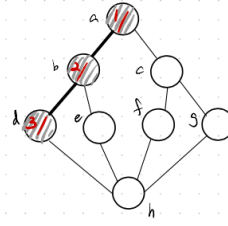
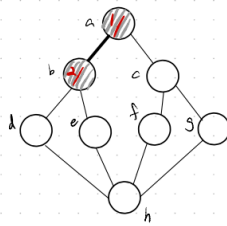
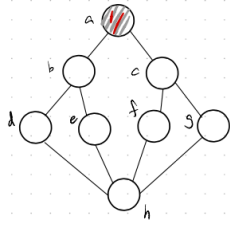
public void DFS(vertex[] vertices, vertex u){
    time = time+1;
    u.findTime = time;
    u.color = "Gray";
    for(int i = 0; i < path[u.vertex].length; i++){
        int v = path[u.vertex][i];
        if(vertices[v].color.equals("White")){
            vertices[v].parents = u.vertex;
            DFS(vertices, vertices[v]);
        }
    }
    u.color = "Black";
    time = time+1;
    u.outTime = time;
}

```

그리고 DFS 메소드에서는 전역변수인 time을 받을 때 증가시켜 탐색시간으로 저장하고 정점의 색을 Gray로 바꾼 다음 인접한 정점에 대해 White인 경우에만 재귀적으로 DFS가 다시 실행되도록 하였습니다. 그리고 이 모든 반복문이 끝나게 되면 완료 시간을 기록하기 위해 탈출시간에서 +1을 더해 완료시간으로 저장되도록 하였습니다.

< 탐색 경로 작성 >

DFS



3. 그래프의 모든 정점(vertex)의 집합 V 를 겹치지 않는 두 개의 집합 V_1 과 V_2 로 나누고, 그래프의 모든 모서리(edge)가 V_1 의 한 정점과 V_2 의 한 정점을 연결하면(다시 말해, V_1 에 속해있는 정점 사이를 연결하는 모서리도 없고 V_2 에 속해있는 정점 사이를 연결하는 모서리도 없다.) 이런 그래프를 이분그래프라 한다. 그래프가 이분 그래프인지 알아내는 효율적인 알고리즘을 설계하라.

```

BipartiteGraph(G, s)
checkBipartite = true
For each vertex  $u \in G.V - \{s\}$ 
    u.belong = null
    u.isVisit = false
s.belong = A
s.isVisit = true
for 1 to number of G.V
    if not checkBipartite
        break
    else
        Q = 0
        ENQUEUE(Q, s)
        While Q  $\neq$  0 and checkBipartite
            U = DEQUEUE(Q)
            For each  $v \in G.Adj[u]$ 
                If not isVisit
                    v.isVisit = true
                    if u.belong equal "A"
                        v.belong = "B"
                    else
                        v.belong = "A"
                    ENQUEUE(Q, v)
                Else if u.belong equal v.belong
                    checkBipartite = false
                    break
if checkBipartite
    Bipartite Graph
Else
    Is not Bipartite Graph

```