알고리즘

- 10. Prim, Dijkstra, Kurskal, Bellman-Ford -

제 출 일	2018.12.17
학 과	컴퓨터공학과
학 번	201402391
이 름	이 병 만

※ 과제 목표 및 해결 방법

- Prim, Dijstra, Kruskal, Bellman-Ford 구현

※ 주요 부분 코드 설명 (알고리즘 부분 코드 캡쳐)

1. Prim

```
public class Graph {
    private static final Graph NIL = null;
    private Graph pie = NIL;
    private int key;
    private int index;

    public Graph() {}

    public Graph(int key) {
        this.setkey(key);
    }

    public Graph(Graph pie, int key, int index) {
        this.setPie(NIL);
        this.setkey(key);
        this.setIndex(index);
    }
```

➔ 그래프가 가지는 속성들을 세팅해주는 클래스이다. key, pie, index를 가지고 그래프 생성자를 위한 함수를 선언하였다.

```
public class prim {
    private static final Graph NIL = null;
    int[][] graph;
    Graph u;
    Graph v;

    public prim() {}

    public void prim(int[][] graph) {
        this.graph = graph;
    }

    public void MST_Prim(Graph[] g, int s_vertex) {
        /* initialize */
        for (int i=0; i<g.length; i++) {
            g[i] = new Graph(NIL, Integer.MAX_VALUE, i);
        }
        g[s_vertex] = new Graph(0);

        /* create Queue*/
        ArrayList<Graph> que = new ArrayList<Graph>();
        for (int i=0; i<g.length; i++) {
            que.add(g[i]);
        }

        while (!que.isEmpty()) {
            u = minHeap.remove(minHeap.heapSort(que));

            for (int i=0; i<g.length; i++) {
                if (graph[u.getIndex()][i] != 0) {
                    v = g[i];
                    if (que.contains(v) && graph[u.getIndex()][v.getIndex()] < v.getkey()) {
                        v.setPie(u);
                        v.setkey(graph[u.getIndex()][v.getIndex()]);
                    }
                }
            }
        }
        /* save file */
        Prim_print("Output_Prim.txt", g);
    }
```

➔ Prim은 키 값이 가장 작은 정점을 선택 후 연결된 정점의 키 값보다 가중치가 작은 것의 키 값과 부모를 갱신해준다. 먼저 읽어온 파일에서 생성한 2차원 배열을 graph에 저장한다.
g 배열을 초기화한다. 이 때 시작 정점도 인자로 받았기 때문에 시작 정점은 키의 값만 초기화해준다.
정점을 담는 배열을 생성한다. 배열이 비어있는지 확인하고 minHeap의 방식을 사용해서 가중치가 가장 작은 정점(w)을 배열에서 꺼낸다. 꺼낸 정점(w)과 인접한 정점(v)을 찾는다. 찾은 정점(v)이 배열에 존재하고, graph에서 저장된 값보다 인접한 정점의 키의 값이 크면 v의 값을 갱신해준다.
배열이 비어있을 때까지 반복하고 출력 함수를 호출한다.

```java
public void Prim_print(String path, Graph[] g) {
    try {
        String savePath = main.class.getResource("").getPath(); // Get a Absolute path
        File file = new File(savePath + path);
        FileWriter fw = new FileWriter(file);
        BufferedWriter bufWriter = new BufferedWriter(fw);

        for (int i=0; i<g.length; i++) {
            if(g[i].getPie()== null) {
                System.out.println(i+" "+i+" "+g[i].getkey());
                continue;
            }
            System.out.println(i+" "+g[i].getPie().getIndex()+" "+g[i].getkey());
            bufWriter.write(i+" "+g[i].getPie().getIndex()+" "+g[i].getkey());
            bufWriter.newLine();
        }

        bufWriter.close();
    } catch(Exception e) {
        System.out.println(e);
    }
}
```

➔ Prim을 출력을 위한 함수이다. 출력한 것을 txt에 써줄 파일을 열어서 해당 인덱스와 연결된 정점과 해당 정점의 키의 값을 써준다. 모든 정점에 대한 파일에 써준 뒤 파일을 닫는다.

```java
public class main {
    public static void main(String[] args) throws IOException {
        int list[][] = File_read.fileRead("graph_sample_prim_kruskal.txt");
        prim prim = new prim();
        Graph[] g = new Graph[list.length];
        prim.prim(list);
        prim.MST_Prim(g, 0);
    }
}
```

➔ 파일을 읽고 2차원 배열을 반환받고, 시작 정점을 지정해준다, 지정한 정점으로부터 탐색을 시작한다.

※ 실행 결과

```
0       8       0       9       11      0       0
8       0       10      0       0       0       0
0       10      0       5       0       0       0
9       0       5       0       13      0       12
11      0       0       13      0       8       0
0       0       0       0       8       0       7
0       0       0       12      0       7       0
0 0 0
1 0 8
2 3 5
3 0 9
4 0 11
5 4 8
6 5 7
```

2. Dijkstra

```
public class Graph {
    private static final Graph NIL = null;
    private Graph pie = NIL;
    private int distance;
    private int index;

    public Graph() {}

    public Graph(int distance, int index) {
        this.setDistance(distance);
        this.setIndex(index);
    }

    public Graph(Graph pie, int distance, int index) {
        this.setPie(NIL);
        this.setDistance(distance);
        this.setIndex(index);
    }
```

➔ 그래프가 가지는 속성들을 세팅해주는 클래스이다. distance, pie, index를 가지고 그래프 생성자를 위한 함수를 선언하였다.

```
public class dijkstra {
    private static final Graph NIL = null;
    int[][] graph;
    Graph u;
    Graph v;
    int w;

    public dijkstra() {}

    public void dijkstra(int[][] graph) {
        this.graph = graph;
    }

    public void Dijkstra(Graph[] g, int s_vertex) {
        ArrayList<Graph> S = new ArrayList<Graph>();

        /* initialize */
        for (int i=0; i<g.length; i++) {
            g[i] = new Graph(NIL, Integer.MAX_VALUE, i);
        }
        g[s_vertex] = new Graph(0, s_vertex);

        /* create Queue*/
        ArrayList<Graph> que = new ArrayList<Graph>();
        for (int i=0; i<g.length; i++) {
            que.add(g[i]);
        }

        while (!que.isEmpty()) {
            u = minHeap.remove(minHeap.heapSort(que));
            S.add(u);

            for (int i=0; i<g.length; i++) {
                if (graph[u.getIndex()][i] != 0) {
                    v = g[i];
                    relax(u, v, graph[u.getIndex()][i]);

                }
            }
        }
        /* save file */
        Dijkstar_print("Output_Dijkstar.txt", g);
    }
```

➔ Dijkstra는 minHeap을 통해 거리가 가장 짧은 정점을 추찰해서 해당 정점을 방문했다고 표시후 그 정점과 연결된 간선으로 거리를 갱신하는 것이다.
먼저 S라는 배열을 생성한다. 이 배열은 que에서 꺼낸 정점을 보관하는 배열이라고 보면 된다.
g라는 정점을 담는 배열을 초기화한다. Prim과 동일하게 인자로 받은 시작 정점을 0으로 초기화해준다.
que 배열을 생성하고 g배열에 저장한 정점들을 하나씩 넣어준다. que가 비어있을 때까지 minHeap을 사용해서 거리가 짧은 정점(u)을 꺼내고, S배열에 넣는다. 꺼낸 정점과 인접한 정점(v)을 찾는다. 추출한 정점과 인접한 정점과 두 정점과의 거리를 relax함수의 인자로 넣는다. 다음과 같은 반복을 하고 que가 비어지면 print함수를 호출한다.

```java
public void relax(Graph u, Graph v, int w) {
    if (v.getDistance() > u.getDistance() + w) {
        v.setDistance(u.getDistance() + w);
        v.setPie(u);
    }
}
```

➔ 인자로 받은 두 개의 정점과 거리로 v의 거리가 u의 거리와 w를 더한 값이 크면 거리가 짧은 값으로 갱신해주는 함수이다.

```java
public void Dijkstar_print(String path, Graph[] g) {
    try {
        String savePath = main.class.getResource("").getPath(); // Get a Absolute path
        File file = new File(savePath + path);
        FileWriter fw = new FileWriter(file);
        BufferedWriter bufWriter = new BufferedWriter(fw);
        String str = "";
        int j=0;
        for (int i=0; i<g.length; i++) {
            j=i;
            while (g[j].getPie() != null) {
                str += "<-" + g[j].getPie().getIndex();
                j = g[j].getPie().getIndex();
            }
            System.out.println(i + str + " " + g[i].getDistance());
            bufWriter.write(i + str + " " + g[i].getDistance());
            bufWriter.newLine();
            str = "";
        }

        bufWriter.close();
    } catch(Exception e) {
        System.out.println(e);
    }
}
```

➔ 출력을 위한 함수이다. 출력한 것을 txt에 써줄 파일을 열어서 모든 정점에 대해서 써준 뒤 파일을 닫는다 .

```java
public class main {

    public static void main(String[] args) throws IOException {
        int list[][] = File_read.fileRead("graph_sample_dijkstra.txt");
        dijkstra Dijkstra = new dijkstra();
        Graph[] g = new Graph[list.length];

        Dijkstra.dijkstra(list);
        Dijkstra.Dijkstra(g, 1);
    }
}
```

➔ 파일을 읽고 2차원 배열을 반환받고, 초기화 후 Dijkstra에 시작 정점을 지정해주어 실행한다.

※ 실행 결과 및 분석

```
0 0 1 0 0 5 0
0 0 0 0 0 0 0
0 0 0 0 0 0 8
0 0 0 12 5 0 0
0 0 0 0 0 0 7
0<-1 8
1 0
2<-3<-1 10
3<-1 9
4<-2<-3<-1 12
5<-1 11
6<-5<-1 19
7<-4<-2<-3<-1 16
```

3. Kruskal

```
public class kruskal {
    private int parent[]; // weightedUnion과 collapsingFind를 위한 배열
    private int size; // 정점의 수
    private int minCost = 0; // 최소 비용
    private Edge[] edges;

    public kruskal() {
        try {
            String path = main.class.getResource("").getPath(); // Get a Absolute path
            File read_file = new File(path + "graph_sample_prim_kruskal.txt");

            // Create inputStream
            FileReader filereader = new FileReader(read_file);
            BufferedReader bufReader = new BufferedReader(filereader);
            String line = "";
            String[] vertex_weigth = null;

            line = bufReader.readLine();
            int total_vertex = line.charAt(1)-48;
            int edge_size = File_read.fileSize("graph_sample_prim_kruskal.txt");
            size = total_vertex;
            parent = new int[size];
            Arrays.fill(parent, -1); // parent 배열 -1로 초기화(정점을 독립 원소로 만듦)

            // 파일에서 읽은 데이터를 바탕으로 E 초기화
            edges = new Edge[edge_size]; // edges 배열 초기화

            int i=0;
            while((line = bufReader.readLine()) != null){ // 간선과 가중치의 나열
                vertex_weigth = line.split(" "); // 간선과 가중치를 배열로 나눠준다
                edges[i] = new Edge(Integer.parseInt(vertex_weigth[0]), Integer.parseInt(vertex_weigth[1]), Integer.parseInt(vertex_weigth[2]));
                // add함수를 호출해서 간선과 가중치 저장
                i++;
            }

            bufReader.close();
        } catch(IOException  e) {
            e.printStackTrace();
        }
    }
```

➔ 파일을 읽어와서 값을 초기화 시켜주는 부분이다. 먼저 파일을 읽는다. 정점의 개수만큼 parent배열의 크기를 정하고, 엣지의 개수를 세어 엣지를 저장할 배열 edges 배열을 초기화한다. 엣지가 들어있는 파일을 읽으면서 edges 배열에 연결된 정점과 그 가중치를 저장한다.

```
/* i가 속한 집합과 j가 속한 집합을 합집합으로 만든다 */
public void weightedUnion(int i, int j) {
    int I_root = collapsingFind(i);// i와 j의 루트를 찾는다. collapsingFind 사용
    int J_root = collapsingFind(j);

    if(parent[I_root] >= parent[J_root]) { // i의 루트크기가 더 작으면
        parent[J_root] = parent[I_root] + parent[J_root]; //j의 루트에 j루트의 크기 + i루트의 크기를 더한 값을 넣어준다.
        parent[I_root] = j; // j의 값을 i루트에 연결
    }
    if(parent[I_root] < parent[J_root]) { // j의 루트크기가 더 작으면
        parent[I_root] = parent[I_root] + parent[J_root]; //i의 루트에 j루트의 크기 + i루트의 크기를 더한 값을 넣어준다.
        parent[J_root] = i; // i의 값을 j루트에 연결
    }
}

/* i가 속한 집합의 root를 찾는다 */
public int collapsingFind(int i) {
    int root; // 루트 변수 선언
    for(root = i; parent[root] >= 0; root = parent[root]); // 루트를 찾는다.
    for(int j = i; j != root; j = parent[j]) {
        parent[j] = root; // parent 배열에 root값 저장
    }
    return root;
}
```

➔ weightedUnion 함수는 I가 속한 집합과 j가 속한 집합을 합집합으로 만드는 함수이다. collapsingFind()를 호출하여 I와 j의 루트를 찾는다. 찾은 루트로 조건문을 실행을 한다. if(parent[I_root] >= parent[J_root])이면 즉, j가 속한 집합의 크기가 크다는 것을 의미하므로 I의 루트는 j의 자식으로 들어온다. j의 루트값은 I와 j의 원소의 크기를 더한 값이 된다. 반대로 j의 루트는 i의 자식으로 들어온다. i의 루트값은 I와 j의 원소의 크기를 더한 값이 된다.
collapsingFind 함수는 I가 속한 집합의 Root를 찾는 함수이다. for문에서 I의 값을 root변수에 넣고, root = parent[root]을 이용해서 부모노드를 따라가면서 루트를 찾는다. 루트를 찾으면 for문을 종료하고 다음 for문을 실행을 한다. I부터 루트까지 연결된 노드들을 다 root의 값으로 저장을 해준다. 이 코드를 통해서 노드들을 평평하게 생성해준다.

```
/* Kruskal 알고리즘 구현 */
public void Kruskal() {
    ArrayList<Edge> A = new ArrayList<Edge>();
    Arrays.sort(edges, new Comparator<Edge>() {
        @Override
        public int compare(Edge a, Edge b) {
            return a.weight - b.weight;
        }
    });

    for(int i = 0; i<edges.length; i++) {
        int I_Root = collapsingFind(edges[i].v); // 루트를 찾는다
        int J_Root = collapsingFind(edges[i].w); // 루트를 찾는다
        if(I_Root == J_Root && !edges[i].selected) {  // 루트가 같으고 선택되었다면
            continue;
        } else {
            weightedUnion(edges[i].v, edges[i].w);
            edges[i].selected = true; // 선택되었다고 마킹
            minCost = minCost + edges[i].weight; // 최소비용 업데이트
            A.add(edges[i]);
        }
    }

    Kruskal_print("Output_Kruskal.txt", A);
}
```

➔ 정렬을 사용해서 가중치의 배열을 오름차순으로 정렬을 하였다.
collapsingFind함수로 v와 w정점의 루트를 찾고 비교를 해서 루트의 값이 다르면 weightedUnion함수를 호출을 해주어 간선을 추가해주었다.
간선을 추가해준 뒤 selected의 값을 ture로 마킹을 해주었고, 기존에 가지고 있던 최소비용의 값과 새로 추가된 가중치를 합해서 업데이트를 해주었다. (여기서 루트의 값을 비교를 해준 이유는 같은 집합에서 간선을 추가하게 될 경우 사이클이 형성된다는 이유 때문에 루트의 값을 비교해주었다.)

```
class Edge {
    int v, w; // 정점
    int weight; // 가중치
    boolean selected; // 선택된 정점을 마킹하는 변수

    public Edge(int v, int w, int weight) { // E를 초기화 해주는 생성자
        this.v = v;
        this.w = w;
        this.weight = weight;
        selected = false;
    }
}
```

➔ Edge Class는 입력파일로부터 그래프의 정보를 얻어 E를 초기화 해주기 위한 클래스이다. 두 개의 정점과 가중치, 선택된 정점을 마킹하는 selected를 변수선언을 하였다. 생성자를 만들어서 정점과 가중치를 입력을 받아서 Edge라는 배열을 초기화해주었다. this를 사용해서 입력받은 값을 그대로 저장해주었고, selected는 아직 방문하지 않았기 때문에 false로 초기화를 해주었다.

```java
public void Kruskal_print(String path, ArrayList<Edge> A) {
    try {
        String savePath = main.class.getResource("").getPath(); // Get a Absolute path
        File file = new File(savePath + path);
        FileWriter fw = new FileWriter(file);
        BufferedWriter bufWriter = new BufferedWriter(fw);

        for (int i=0; i<A.size(); i++) {
            System.out.println(A.get(i).v + " " + A.get(i).w  + " " + A.get(i).weight);
            bufWriter.write(A.get(i).v + " " + A.get(i).w  + " " + A.get(i).weight);
            bufWriter.newLine();
        }

        bufWriter.close();
    } catch(Exception e) {
        System.out.println(e);
    }
}
```

➔ 출력을 위한 함수이다. 출력한 것을 txt에 써줄 파일을 열어서 모든 정점에 대해서 써준 뒤 파일을 닫는다 .

```java
public class main {
    public static void main(String[] args) throws IOException {
        kruskal mst = new kruskal();
        mst.Kruskal();
    }
}
```

※ 실행 결과 및 분석

```
2 3 5
5 6 7
0 1 8
4 5 8
0 3 9
0 4 11
```

4. Bellma-Ford

```java
public class bellman_ford {
    private Edge[] edges;

    public bellman_ford() {
        try {
            String path = main.class.getResource("").getPath(); // Get a Absolute path
            File read_file = new File(path + "graph_sample_bellman.txt");

            // Create inputStream
            FileReader filereader = new FileReader(read_file);
            BufferedReader bufReader = new BufferedReader(filereader);

            int counter = File_read.fileSize("graph_sample_bellman.txt");
            String line = "";
            String[] vertex_weigth = null;

            line = bufReader.readLine();
            int total_vertex = Integer.parseInt(line);

            int i=0;
            edges = new Edge[counter];
            while ((line = bufReader.readLine()) != null) {
                vertex_weigth = line.split(" ");
                edges[i] = new Edge(Integer.parseInt(vertex_weigth[0]), Integer.parseInt(vertex_weigth[1]), Integer.parseInt(vertex_weigth[2]));
                i++;
            }

            final int N = total_vertex;

            bellmanford bellmanfords = new bellmanford();
            bellmanfords.bellmanford(edges, 1, N);
            bufReader.close();
        }
        catch(IOException e) {
            System.out.println(e);
        }
    }
}
```

➔ 파일을 읽어와서 값을 초기화 시켜주는 부분이다. 먼저 파일을 읽는다. 정점의 개수를 저장하고, 엣지의 개수를 세어 엣지를 저장할 배열 edges 배열을 초기화한다. 엣지가 들어있는 파일을 읽으면서 edges 배열에 연결된 정점과 그 가중치를 저장한다.

```java
class Edge {
    int v, w; // 정점
    int weight; // 가중치
    boolean selected; // 선택된 정점을 마킹하는 변수

    public Edge(int v, int w, int weight) { // E를 초기화 해주는 생성자
        this.v = v;
        this.w = w;
        this.weight = weight;
        selected = false;
    }
}
```

➔ Edge Class는 입력파일로부터 그래프의 정보를 얻어 E를 초기화 해주기 위한 클래스이다. 두 개의 정점과 가중치, 선택된 정점을 마킹하는 selected를 변수선언을 하였다. 생성자를 만들어서 정점과 가중치를 입력을 받아서 Edge라는 배열을 초기화해주었다. this를 사용해서 입력받은 값을 그대로 저장해주었고, selected는 아직 방문하지 않았기 때문에 false로 초기화를 해주었다.

```java
public void bellmanFord(Edge[] edges, int source, int N) {
    int E = edges.length;
    int distance[] = new int[N];
    int parent[] = new int[N];

    Arrays.fill(distance, Integer.MAX_VALUE);
    distance[source] = 0;

    Arrays.fill(parent, -1);

    int k = N;

    while (--k > 0) {
        for (int j = 0; j < E; j++) {
            int u = edges[j].v;
            int v = edges[j].w;
            int w = edges[j].weight;

            if (distance[u] != Integer.MAX_VALUE && (distance[u] + w < distance[v])) {
                distance[v] = distance[u] + w;
                parent[v] = u;
            }
        }
    }

    for (int i = 0; i < E; i++) {
        int u = edges[i].v;
        int v = edges[i].w;
        int w = edges[i].weight;

        if (distance[u] != Integer.MAX_VALUE && (distance[u] + w < distance[v])) {
            System.out.println("Negative Weight Cycle Found!!");
            return;
        }
    }
```

➔ 엣지의 길이를 저장하고 거리를 저장할 배열과 parent 배열을 반든다. 초기화 작업을 하고 k가 음수일 때까지 반복문을 실행하면서 엣지의 개수만큼 반복문을 또 실행한다. 엣지의 각 정점과 가중치를 저장하고 비교하여 갱신한다. 사이클이 생기면 안되므로 아래 반복문으로 처리했다.

```java
try {
    String savePath = main.class.getResource("").getPath(); // Get a Absolute path
    File file = new File(savePath + "Output_Bellman_Ford.txt");
    FileWriter fw = new FileWriter(file);
    BufferedWriter bufWriter = new BufferedWriter(fw);
    String str = "";

    for (int i = 0; i < N; i++) {
        str = printPath(parent, i);
        System.out.println(i+ str + " " + distance[i]);
        bufWriter.write(str + distance[i]);
        bufWriter.newLine();
    }


    bufWriter.close();
} catch(Exception e) {
    System.out.println(e);
}
```

➔ 파일에다 저장하는 것으로 i에 해당하는 경로를 반환받아서 파일에 한 줄씩 저장한다.

```
public String printPath(int parent[], int v) {
    String str = "";
    v = parent[v];
    while (v >= 0) {
        str += " <- " + v;
        v = parent[v];
    }
    return str;
}
```

➔ 인자로 배열과 해당 정점을 입력받고 v가 0보다 작을 때까지 반복하여 경로를 저장한 뒤, 음수가 되면 반환한다.

※ 실행 결과 및 분석