CSE3081-02 알고리즘 설계와 분석

[숙제 3] 보고서

20161663 허재성

1. 실행 결과 및 분석

실행한 컴퓨터의 환경

OS : Windows 10

CPU : Intel® Core(TM) i3-5005U CPU @ 2.00GHz

RAM : 8.00GB

Compiler : Visual Studio 19 Release Mode/x64 Platform

실행 결과

파일 이름	작동 여부	MST weight	수행 시간(초)	$k_{scanned}$
HW3_com-amazon.ungraph.txt	YES	2,729,670,156	4.365(0.027)	925,853
HW3_com-dblp.ungraph.txt	YES	2,747,895,457	4.699(0.030)	1,049,826
HW3_com-lj.ungraph.txt	YES	28,308,045,762	195.708(1.030)	34,681,152
HW3_com-youtube.ungraph.txt	YES	14,578,691,475	14.11(0.08)	2,987,623

그래프의 size가 가장 큰 HW3_com-lj.ungraph.txt의 경우 결과를 얻기 위해서 VS에 디폴트로 설정된 스택, 힙의 예약 크기를 536870912로 증가시켰다.

수행 시간을 과정 별로 나누어 보면 다음과 같다. 시간 단위는 초이다.

	amazon.ungraph	dblp.ungraph	lj.ungraph	youtube.ungraph
num of vertices (V)	334,863	317,080	3,997,962	1,134,890
num of edges (E)	925,872	1,049,866	34,681,189	2,987,624
make-graph	0.001	0.001	0.053	0.004
construct-graph	2.772	3.044	101.552	8.264
make-component	0.381	0.329	9.571	1.576
make-heap	0.027	0.030	1.030	0.086
Kruskal	0.938	1.044	62.828	3.490
delete-graph	0.244	0.248	20.673	0.695

total	4.365	4.699	195.708	14.116

각 과정의 시간 복잡도에 대한 설명은 아래 **2. 자료 구조(Data Structure), 알고리즘(Algorithm)** 항목에 코드와 함께 자세히 설명되어 있다.

1. make-graph는 파일로부터 V, E, max-weight를 입력 받은 후 그래프 객체를 생성하는 과정이다. 시간 복잡도는 O(V)이다.

2. construct-graph는 파일로부터 간선의 정보를 입력 받아 그래프의 인접 리스트에 간선을 추가하는 과정이다. 하나의 간선을 삽입하는데 시간 복잡도는 O(1)이므로 전체 시간 복잡도는 O(E)이다.

3. make-component는 DFS로 그래프의 연결 요소를 찾고 연결 요소의 개수만큼 연결 요소 객체를 생성하는 과정이다. 이 때 시간 복잡도에 가장 큰 영향을 주는 과정은 DFS로 DFS의 시간 복잡도는 인접 리스트를 사용했으므로 O(V + E)가 된다.

4. make-heap 과정에서는 Kruskal Algorithm을 이용해 MST를 찾기 위해 간선들의 weight 값을 key로 하는 최소 힙을 구성해야 한다. 최소 힙을 구성하기 위한 시간 복잡도는 O(E)가 된다.

5. Kruskal 과정에서는 간선의 최소 힙과 정점의 disjoint sets를 바탕으로 MST를 찾는다. 이 때 시간 복잡도는 최악의 경우 O(ElogV)가 된다. Kruskal Algorithm의 전체 과정 중 make-heap 과정의 시간은 제외되었다.

6. delete-graph 과정에서는 그래프에서 동적 할당한 메모리를 모두 해제한다. 인접 리스트를 해제하는 데 가장 많은 시간이 소모되는데 시간 복잡도는 O(V + E)이다.

4개의 그래프 모두 construct-graph 과정에서 가장 많은 시간이 소모되었다. 이론적인 시간 복잡도는 O(E)에 불과하지만 new를 이용한 동적 메모리 할당의 실제 시간 소모가 크므로 오랜 시간이 소모된다.

construct-graph 과정의 시간(단위는 초)을 비교해보면 다음과 같다.

graph	Num of edges (E)	E_2/E_1	construct-graph (t)	t_2/t_1
amazon	925,872	-	2.772737	-
dblp	1,049,866	1.133921	3.044731	1.098096
youtube	2,987,624	2.845719	8.264012	2.714201
lj	34,681,189	11.608284	101.552170	12.288482

어떤 두 그래프의 간선의 개수를 각각 E_1, E_2라 하고 construct-graph 과정에서 소요 시간을 t_1, t_2라 했을 때, E_1, E_2의 비와 t_1, t_2의 비가 거의 일치함을 알 수 있다. 따라서 construct-graph 과정의 소요 시간이 시간 복잡도가 O(ElogV)인 Kruskal Algorithm보다 더 오래 걸리더라도 시간 복잡도 자체는 O(E)에 이루어지는 것을 알 수 있다.

make-component 과정(DFS)의 시간(단위는 초)을 비교해보면 다음과 같다.

graph	V	E	X = V + E	X_2/X_1	DFS (t)	t_2/t_1
amazon	334,863	925,872	1,260,735	-	0.381141	-
dblp	317,080	1,049,866	1,367,263	1.084496	0.329847	0.865420
youtube	1,134,890	2,987,624	4,122,514	3.015158	1.576061	4.778158
lj	3,997,962	34,681,189	38,679,151	9.382418	9.571396	6.072986

이론적인 시간 복잡도와 실제 시간에 차이가 있다. 실제로는 단순히 V +E의 합이 아니라 모든 간선을 두 번씩 확인해야 하므로 차이가 있다. 또한 같은 그래프에 대해서도 시간 결과가 일정히 나오지 않고 조금씩 차이가 있어 각 그래프의 시간 결과가 시간 복잡도를 정확히 반영하지 않을 수 있다. make-component 과정에는 DFS의 시간 비중이 가장 높지만 그 외에도 다른 연산들이 포함되기 때문에 시간 복잡도가 실제 시간을 정확히 반영하기 어렵다. 대략적인 시간 복잡도는 O(V +E)이다.

Kruskal Algorithm의 과정 중 make-heap 과정의 시간(단위는 초)을 비교해보면 다음과 같다.

graph	Num of edges (E)	E_2/E_1	make-heap (t)	t_2/t_1
amazon	925,872	-	0.027293	-
dblp	1,049,866	1.133921	0.030060	1.101381
youtube	2,987,624	2.845719	0.086690	2.883898
lj	34,681,189	11.608284	1.030117	11.882766

어떤 두 그래프의 간선의 개수를 각각 E_1, E_2라 하고 make-heap 과정에서 소요 시간을 t_1, t_2라 했을 때, E_1, E_2의 비와 t_1, t_2의 비가 거의 일치함을 알 수 있다. 따라서 make-heap 과정이 O(E)에 이루어지는 것을 알 수 있다.

Kruskal Algorithm의 과정 중 make-heap 과정을 제외한 시간(단위는 초)을 비교해보면 다음과 같다. 로그의 밑은 10으로 통일했다.

graph	V	E	X = ElogV	X_2/X_1	Kruskal (t)	t_2/t_1
amazon	334,863	925,872	5,115,319	-	0.938670	-
dblp	317,080	1,049,866	5,775,490	1.129057	1.044692	1.112949
youtube	1,134,890	2,987,624	18,089,925	3.132188	3.490238	3.340925
lj	3,997,962	34,681,189	228,959,614	12.656747	62.827858	18.001024

이론적인 시간 복잡도 O(ElogV)와 실제 시간이 어느 정도 일치한다. youtube 그래프와 lj 그래프의 시간 차이를 비교했을 때, lj의 시간 소요가 시간 복잡도로 예측한 것보다 더 크게 나타나는 것을 알 수 있다. 실제로 Kruskal Algorithm의 과정에서 최소 힙에서 간선을 가져오는 과정에 O(ElogE)의 시간이 소요되는데 O(ElogE)는 시간 복잡도의 관점에서 O(ElogV)와 같기 때문에 무시

되었다. 이를 고려하면 Kruskal Algorithm이 시간 복잡도 O(ElogV)에 동작한다고 할 수 있다.

그래프 객체 graph를 동적 할당하고 해제하는 과정을 제외한 모든 과정의 시간 복잡도를 합하면 (포함해도 결과는 같다.) 다음과 같다.

O(E) + O(V + E) + O(E) + O(ElogV) = O(ElogV)

따라서 그래프를 구성하고 connected component를 모두 찾고 Kruskal Algorithm으로 MST를 찾는 전체 과정의 시간 복잡도는 O(ElogV)이다.

2. 자료 구조(Data Structure), 알고리즘(Algorithm)

Kruskal Algorithm을 구현하기 위해 필요한 Disjoint Set을 위한 클래스를 정의한다. Set은 Disjoint Set의 원소가 하나인 개별 집합을 구현한 클래스이다.

```
class Set { // 정점 번호를 포함한 disjoint-set의 개별 집합
public:
        int vname; // 정점 번호 == set 번호
        int rank;
        Set* parent; //부모 노드를 가리키는 포인터

        // Set 생성자
        Set();
        Set(int _v);
};
```

멤버 변수

1. vname은 원소가 하나인 개별 집합의 원소로 개별 집합에 저장될 그래프의 정점(vertex)의 번호를 의미하며 이 자체로 개별 집합의 번호가 된다. 그래프에서 정점의 개수는 최대 2^{24}개로 int형으로 표현 가능하다.

2. rank는 Disjoint Set에서 각 집합을 조상으로 하는 집합의 깊이 중 최대 깊이이다. 트리에서 서브 트리의 depth(루트 노드 1 기준) 중 최댓값이 된다.

3. parent는 각 집합의 부모 집합을 가리키는 포인터이다. 집합 x의 parent가 집합 y를 가리키면 집합 x, y는 y로 합쳐진다.

```
Set::Set() {}
Set::Set(int _v) : vname(_v), rank(0), parent(NULL) {}
```

생성자

vname을 입력받은 정점의 번호로 초기화한다. 처음에는 각 정점이 원소가 하나인 개별 집합이므로 rank를 0으로 초기화한다. 이 상태에서는 부모 집합이 없으므로 NULL로 설정한다.

Set을 이용해 Disjoint Set을 구현한 DisjointSet 클래스이다.

```
class DisjointSet { // Union-Find를 위한 disjoint-set
public:
	int numOfSets; // 개별 집합 개수 = 정점 개수(초기)
	Set* disjointSets;

	// DisjointSet 생성자, 멤버 함수, 소멸자
	DisjointSet(int _num);
	int find(int v);
	void unionByRank(int x, int y);
	~DisjointSet();
};
```

멤버 변수

1. numOfSets는 Disjoint Sets의 개별 집합 개수이다. 처음에는 그래프의 정점의 개수와 같다.

2. disjointSets는 원소가 하나인 개별 집합 Set 객체들을 저장할 배열을 동적할당하기 위한 포인터 변수이다.

```
DisjointSet::DisjointSet(int _num) : numOfSets(_num) {
	disjointSets = new Set[numOfSets];
	for (int i = 0; i < numOfSets; i++)
		disjointSets[i] = Set(i);
}
```

생성자 DisjointSet은 numSet을 전달받은 인자 _num으로 초기화한 후 결정된 numOfSets 크기만큼 Set 배열 disjointSets를 동적 할당한다. disjointSets 배열을 할당한 후 각 배열의 원소로 Set 객체를 생성한다.

```
int DisjointSet::find(int v) {
	if (!disjointSets[v].parent) return v;
	else return find(disjointSets[v].parent->vname);
}

void DisjointSet::unionByRank(int x, int y) {
	int rootX = find(x);
	int rootY = find(y);

	if (rootX == rootY) return;

	if (disjointSets[rootX].rank < disjointSets[rootY].rank) {
		disjointSets[rootX].parent = &disjointSets[rootY];
	}
	else {
		if (disjointSets[rootX].rank == disjointSets[rootY].rank)
			disjointSets[rootX].rank++;
		disjointSets[rootY].parent = &disjointSets[rootX];
	}
	numOfSets--; // 서로 다른 두 집합이 합쳐짐. 집합의 개수는 1개 감소
}
```

멤버 변수로 Disjoint Set의 핵심 기능인 find와 union을 구현한 멤버 변수 find와 unionByRank가

있다.

1. find는 입력으로 전달받은 번호의 원소가 속한 집합 번호를 반환한다. 원소가 하나인 개별 집합일 경우 개별 집합 번호가 반환되며 개별 집합이 아닐 경우 부모 원소로 이동해 트리의 루트의 번호를 반환한다. 해당 번호가 집합의 번호이다. 루트 번호를 찾기 위해 재귀적으로 구현되었다. 두 집합을 합하는 union 기능이 Union-By-Rank 방법으로 구현되었다면 두 집합을 합했을 때 트리의 깊이는 집합에 속한 정점의 개수가 V개일 때, O(logV)가 된다. 따라서 find 함수는 O(logV)의 시간 복잡도를 가진다.

2. unionByRank는 두 개별 집합을 합하는 함수로 Union-By-Rank 방법에 의해 구현되었다. 두 개별 집합이 합쳐질 때 루트 노드의 rank가 작은 집합이 큰 집합으로 합쳐진다. 이때 합쳐진 집합의 rank는 이전의 큰 rank 값으로 유지된다. 만약 두 집합의 루트의 rank가 같을 경우 합쳐진 집합의 rank는 이전 rank보다 1이 증가한다. find 함수를 이용해 각 개별 집합의 루트를 찾는다. find 함수로 합칠 두 집합의 루트를 찾는데 O(logV)의 시간이 소모된다. 그 외의 연산은 모두 O(1)에 이루어지므로 unionByRank 함수의 시간 복잡도는 O(logV)이다.

```
DisjointSet::~DisjointSet() {
        delete[] disjointSets;
        disjointSets = NULL;
}
```

소멸자 ~DisjointSet()으로 동적 할당된 disjointSets 배열을 해제한다.

그래프에서 간선(edge)의 정보를 저장하는 클래스 Edge이다.

```
class Edge {
public:
        long long name; // ename = (v1, v2)    // name 0 ~ |E| - 1
        int v1;
        int v2;
        long long weight;
        int flag; // 해당 edge가 mst에 속할 경우 어떤 component의 mst의 edge인지 표시
                  // mst edge에 포함되지 않을 경우 NONE(-1)

        // Edge 생성자
        Edge();
        Edge(long long _n, int _v1, int _v2, long long _w);
};
```

멤버 변수

1. name은 간선의 번호이다. 간선이 |E|개 존재할 경우 간선 번호는 0~|E| - 1 까지 존재한다. 간선의 개수가 최대 2^{32}까지 가능하므로 int 형(2^{31}-1이 최대)으로 모두 표현할 수 없다. 따라서 long long 자료형(2^{63} – 1이 최대)로 표현한다.

2. v1, v2는 간선의 두 정점의 번호를 의미한다. 입력에 따라 다르지만 통상적으로 v1 < v2이다. 그래프에 루프(loop)가 존재하지 않으므로 v1과 v2가 같을 수는 없다.

3. weight는 간선의 비용을 나타내는 변수이다. max_weight가 최대 2^{32}이므로 위와 같은 이유로 long long 자료형으로 표현한다.

4. flag는 각 간선이 그래프의 연결 요소(connected component)의 MST(최소 비용 신장트리)에 속할 때, 어떤 연결 요소의 MST에 속하는 지를 해당 연결 요소의 번호로 나타낸다. 만약 MST에 포함되지 않을 경우 NONE(매크로로 -1로 정의)이다.

```
Edge::Edge() {}
Edge::Edge(long long _n, int _v1, int _v2, long long _w)
        : name(_n), v1(_v1), v2(_v2), weight(_w), flag(NONE) {}
```

생성자로 전달 받은 인자에 맞게 name, v1, v2, weight를 설정해주고 flag를 NONE으로 초기화해준다. 파일로 그래프의 정보를 읽어 간선 정보를 하나씩 읽을 때마다 Edge 객체가 생성된다.

그래프를 표현하는 인접 리스트(Adjacency List)를 구현하기 위해 연결 리스트의 노드를 정의해야 한다. 이를 위한 EdgeNode 클래스이다.

```
class EdgeNode {
public:
        long long edgeName;     // edge 이름을 저장
        EdgeNode* next;

        // EdgeNode 생성자
        EdgeNode(long long _en);
};
```

EdgeNode에는 멤버 변수로 그래프의 간선 정보가 저장되어 있다.

1. edgeName은 그래프의 간선의 번호이다. Edge 클래스의 name과 같다.

2. EdgeNode 포인터 next는 EdgeNode 객체가 다른 EdgeNode 객체를 가리킬 수 있도록 하는 포인터이다. 이를 이용해 인접 리스트를 구현할 수 있다.

```
EdgeNode::EdgeNode(long long _en) : edgeName(_en), next(NULL) {}
```

생성자로 간선의 번호를 인자로 받으면 edgeName을 초기화하고 노드의 next 포인터를 NULL로 초기화한다.

그래프의 경우, 모든 정점이 연결되어 있지 않을 수 있다. 이 경우 각 정점이 어떤 연결 요소에 속하는지 알기 위해서는 DFS 또는 BFS의 과정이 필요하다. 이번 과제에서는 DFS로 연결 요소를 찾아낸다.

DFS를 재귀함수로 구현할 경우 그래프의 정점의 개수가 많아지면 DFS의 깊이가 깊어져서 호출

스택에서 이용 가능한 스택 메모리를 초과해서 사용하게 되어 Stack Overflow가 발생할 수 있다. 이를 방지하기 위해 DFS를 재귀 함수가 아닌 반복문과 스택 자료구조를 이용하여 구현한다. 이를 위해 스택 자료구조를 구현해야 한다.

스택은 연결 리스트를 이용하여 구현한다. 다음은 스택의 노드인 StackNode 클래스이다.

```
class StackNode { // DFS를 위한 스택의 노드
public:
        int vertexName;// vertex 이름을 저장
        StackNode* next;

        // StackNode 생성자
        StackNode(int _vn);
};
```

1. vertexName은 스택에 저장된 그래프의 정점의 번호가 저장되어 있다.

2. StackNode 포인터 next는 다른 StackNode 객체를 가리킬 수 있는 포인터로 이를 이용해 연결 리스트로 스택을 구현할 수 있다.

```
StackNode::StackNode(int _vn) : vertexName(_vn), next(NULL) {}
```

생성자로 전달받은 인자로 vertexName을 초기화하고 next도 NULL로 초기화한다.

StackNode를 이용하여 Stack 자료구조를 구현한다.

```
class Stack {   // DFS를 위한 Stack
public:
        StackNode* top;

        // Stack 생성자, 멤버 함수, 소멸자
        Stack();
        bool empty();
        void push(int vertex);
        int peek();
        int pop();
        ~Stack();
};
```

멤버 변수인 StackNode 포인터 top은 스택에 가장 최근에 추가된 노드를 가리킨다. top에서 스택으로의 삽입과 스택에서의 삭제가 이루어진다.

```
Stack::Stack() : top(NULL) {}
```

생성자로 top을 NULL로 초기화한다.

```
bool Stack::empty() {
        return !top;
}

void Stack::push(int vertex) {
        StackNode* node = new StackNode(vertex);
        node->next = top;
        top = node;
}
```

```
int Stack::peek() {
        if (empty()) return NONE;
        return top->vertexName;
}

int Stack::pop() {
        if (empty()) return NONE;
        StackNode* delNode = top;
        int ret = delNode->vertexName;
        top = delNode->next;
        delNode->next = NULL;
        delete delNode;
        return ret;
}
```

멤버 함수로 스택의 주요 기능들을 구현한다.

1. empty 함수는 top을 확인해 스택이 비어 있는지 확인한다.

2. push 함수는 입력받은 vertex를 원소로 가지는 새로운 StackNode 객체를 동적 할당 후 해당 노드를 스택의 top에 추가한다.

3. peek은 스택에 가장 최근에 추가된 노드의 원소를 반환한다. 이 때 해당 노드를 스택에서 삭제하지 않는다.

4. pop은 스택에 가장 최근에 추가된 노드의 원소를 반환한다. peek과 다르게 해당 노드의 연결을 끊어 스택에서 제거하고 delete로 해제한다.

스택의 주요 기능들은 모두 O(1)의 시간 복잡도를 가진다.

```
Stack::~Stack() {
        while (!empty())
                pop();
}
```

소멸자로 스택에 남아 있는 노드가 있을 경우 스택이 빌 때까지 pop 연산을 수행한다.

그래프의 연결 요소들을 표현하기 위해 Component 클래스를 정의한다.

```
class Component {
public:
        int Cname;              // component 이름(0 ~ component 개수 - 1)
        int CVnum;              // component에 포함된 vertex 개수
        long long CEnum;        // component의 MST에 포함된 edge 개수
        long long kscanned;     // MST 찾는 과정에서 처리한 edge 개수
        long long mstCost;      // component의 MST cost

        // Component 생성자
        Component();
        Component(int _n);
};
```

멤버 변수

1. Cname은 연결 요소의 변호이다. 연결 요소가 |C|개 존재할 경우 번호는 0 ~ |C| - 1이 존재한다.

2. CVnum은 연결 요소에 포함된 정점의 개수이다.

3. CEnum은 각 연결 요소의 MST에 포함된 간선의 개수이다. MST를 찾는 과정에서 MST에 속하는 간선을 발견할 때마다 CEnum이 1씩 증가하고 최종적으로 CVnum – 1개가 된다.

4. kscanned는 MST를 찾는 과정에서 처리한 간선의 개수이다. weight 기준으로 비내림차순으로 간선을 선택했을 때 해당 간선이 MST에 추가되거나, 추가하면 cycle을 생성해 추가되지 않는 경우 모두 다 해당 간선을 처리한 것으로 생각한다. 따라서 모든 Component 객체의 kscanned 합은 최소 |V| - |C| (|C|개의 연결 요소가 모두 MST를 이루었을 때 MST로 이루어진 Forest의 간선의 개수)이고 최대 |E|가 된다.

5. mstCost는 해당 연결 요소의 MST의 간선의 weight의 합이다.

```
Component::Component() {}
Component::Component(int _n) : Cname(_n), CVnum(0), CEnum(0), kscanned(0), mstCost(0)
{}
```

생성자로 인자로 입력받은 연결 요소 번호로 Cname을 초기화한다. 나머지 변수들은 일단 0으로 초기화 후 추후에 갱신한다.

앞서 정의한 자료 구조들을 이용하여 Graph 클래스를 정의한다.

```
class Graph {
public:
	int Vnum;		//	vertex 개수
	int Cnum;		//	component 개수
	long long Enum;		//	edge 개수
	long long Wmax;		//	최대 weight
	EdgeNode** adjList;	//	Adjacent list
	EdgeNode** next;	//	vertex의 edge 중 다음에 확인할 edge 번호 담긴
Node 저장 배열, DFS에 이용
	Edge* edgeArray;	//	edge 저장 배열
	int* components;	//	각 정점이 어떤 component에 속하는 지 표시
	Component* comp;	//	connected-components
	long long* heap;	//	edge 저장 heap
	long long heapNum;	//	heap에 저장된 원소 개수
	Stack* st;		//	dfs를 위한 스택
	DisjointSet* ds;	//	vertices로 이루어진 disjoint-sets

	//	Graph 생성자, 멤버 함수, 소멸자
	Graph(int _V, long long _E, long long _W);
	void insertEdge(long long ename, int v1, int v2, long long weight);
	void dfs(int v);
	void findComponents();
```

```
        void makeComponents();
        void makeDisjointSets();
        bool heapEmpty();
        void adjustHeap(long long i);
        void makeHeap();
        long long heapDelete();
        void kruskal();
        ~Graph();
};
```

멤버 변수

1. Vnum, Enum은 그래프에 포함된 정점과 간선의 개수이다.

2. Cnum은 그래프의 연결 요소의 개수이다.

3. Wmax는 그래프의 간선이 가질 수 있는 최대 weight이다.

4. EdgeNode 이중 포인터 adjList는 인접 리스트를 위한 EdgeNode* 포인터 배열을 동적 할당하기 위한 이중 포인터이다.

5. EdgeNode 이중 포인터 next는 DFS 시 특정 정점의 간선 중 다음에 탐색할 간선의 정보를 저장하기 위한 EdgeNode* 포인터 배열을 동적 할당하기 위한 이중 포인터이다.

6. Edge 포인터 edgeArray는 그래프의 간선을 나타내는 Edge 객체들을 저장할 배열을 동적할당하기 위한 포인터이다.

7. int 포인터 components는 그래프의 각 정점이 속한 연결 요소 번호를 저장하는 배열을 동적 할당하기 위한 포인터이다.

8. Component 포인터 comp는 찾아낸 연결 요소 개수만큼 Component 객체를 저장할 배열을 동적 할당하기 위한 포인터이다.

9. long long 포인터 heap은 edge의 weight를 key로 하는 최소 힙을 구현하기 위한 포인터이다. edge의 번호를 저장할 배열을 동적 할당한다.

10. heapNum은 heap에 저장된 edge 번호의 개수이다. 초기에는 |E|와 같다.

11. Stack 포인터 st는 DFS를 위한 스택을 동적 할당하여 사용한다.

12. DisjointSet 포인터 ds는 정점들로 이루어진 개별 집합을 위한 포인터 변수로 DisjointSet 객체를 동적 할당하여 사용한다.

```
Graph::Graph(int _V, long long _E, long long _W)
        : Vnum(_V), Cnum(0), Enum(_E), Wmax(_W), comp(NULL), heapNum(Enum), ds(NULL)
{
        adjList = new EdgeNode * [Vnum];
        next = new EdgeNode * [Vnum];
        for (int i = 0; i < Vnum; i++)
                adjList[i] = NULL;
        edgeArray = new Edge[Enum];
        components = new int[Vnum];
```

```
        for (int i = 0; i < Vnum; i++)
                components[i] = NONE;
        heap = new long long[heapNum + 1];
        st = new Stack();
}
```

Graph 객체의 생성자이다. 파일을 통해 그래프의 정점의 개수, 간선의 개수, 간선의 최대 비용을 알아낸 후 인자로 전달 받으면 Vnum, Enum, Wmax를 각각 초기화한다. Cnum은 0으로 초기화하고, comp, ds는 NULL로 초기화한다. heapNum은 알아낸 간선의 개수로 초기화한다.

인접 리스트를 구현하기 위해 EdgeNode 포인터 배열을 동적 할당 후 adjList가 가리킨다. 마찬가지로 DFS에 사용하기 위해 EdgeNode 포인터 배열을 동적 할당 후 next가 가리킨다. adjList의 각 원소를 NULL로 초기화한다.

알아낸 간선 개수만큼 Edge 배열을 동적 할당하여 edgeArray가 가리킨다. 또한 알아낸 정점 개수만큼 int 배열을 동적 할당하여 components가 가리킨다. 초기에는 각 정점이 어떤 연결 요소에 속하는 지 알지 못하므로 배열 components의 모든 원소를 NONE으로 설정한다. 힙을 위해 Enum + 1 크기의 배열을 동적 할당하고 heap이 가리킨다. 마지막으로 Stack을 동적 할당한다.

정점의 개수가 V일 때, 시간 복잡도는 O(V)이다.

```
void Graph::insertEdge(long long ename, int v1, int v2, long long weight) {
        Edge e(ename, v1, v2, weight);
        EdgeNode* node1 = new EdgeNode(ename);
        EdgeNode* node2 = new EdgeNode(ename);

        edgeArray[ename] = e;
        heap[ename + 1] = ename;        //      heap에 일단 순서대로 저장
        node1->next = adjList[v1];
        node2->next = adjList[v2];
        adjList[v1] = node1;
        adjList[v2] = node2;
}
```

생성자로 그래프를 생성 후 그래프에 간선을 추가하는 함수이다. 간선 번호와 간선의 두 정점 번호, 간선의 비용을 전달받는다. 전달 받은 인자들을 이용하여 Edge 객체를 만들고 EdgeNode를 동적 할당하여 인접 리스트 adjList에 추가한다. Edge 객체 e를 만들고 간선의 번호에 따라 edgeArray의 알맞은 위치에 e를 삽입한다. 또한 heap에 알맞은 위치(edgeArray에 저장된 인덱스 + 1)에 e의 번호를 삽입한다. 일단 heap 배열에 간선이 입력된 순서대로 저장 후 후에 makeHeap 함수에 의해 최소 힙이 만들어진다.

insertEdge 함수를 한번 호출할 경우 시간 복잡도는 O(1)이다. 그래프의 간선의 개수를 E라 하면 총 E번 호출되므로 O(E)의 시간 복잡도를 가진다.

```
void Graph::dfs(int v) {        //      connected component를 찾기 위한 DFS
        EdgeNode* w = NULL;
        components[v] = Cnum;   //      v를 방문
        st->push(v);
```

```
        while (!st->empty()) {
                int u = st->peek();
                int prev = u;
                w = next[u];

                for (; ; ) {
                        if (!w) {
    //  현재 vertex의 간선 중 방문할 정점이 있는 간선이 없을 경우
                                st->pop();
                                break;
                        }

                        int vnext = edgeArray[w->edgeName].v1 == prev ?
                        edgeArray[w->edgeName].v2 : edgeArray[w->edgeName].v1;
        //      현재 edge의 또 다른 vertex
                        if (components[vnext] == Cnum) {
                                w = w->next;
                                continue; // 이미 dfs로 탐색한 경우, 다른 edge로 넘어감
                        }
                        else {
                                components[vnext] = Cnum;
                                st->push(vnext);
                                next[prev] = w->next;
                                w = adjList[vnext];
                                prev = vnext;
                        }
                }
        }
        Cnum++; //      다음 connected component 번호 & component 개수 1 증가
}
```

각 정점이 속한 연결 요소를 찾기 위한 dfs 함수이다. 시작 정점을 탐색하여 속한 연결 요소를 결정 후 해당 정점을 스택에 삽입한다. 스택이 빌 때까지 스택의 가장 최근에 삽입된 정점 u를 확인한다. 만약 정점 u와 간선으로 연결된 정점 중 더 이상 탐색할 정점이 없을 경우 스택에서 u를 제거한다. 만약 u의 어떤 간선으로 연결된 정점이 이미 탐색한 정점일 경우 다음 간선으로 넘어간다. 그렇지 않을 경우 해당 정점을 탐색하여 속한 연결 요소를 결정 후 스택에 삽입한다. 이 때 현재 정점 u의 다음에 탐색할 간선을 바로 찾을 수 있게 next 배열과 prev 변수를 이용한다. 스택이 비고 연결된 모든 정점을 찾은 후 Cnum의 개수를 1 증가시킨다.

그래프의 정점의 개수가 V, 간선의 개수가 E일 때, 인접 리스트로 그래프를 구현했을 때 DFS의 시간 복잡도는 O(V + E)이다.

```
void Graph::findComponents() {          //      각 vertex가 속한 컴포넌트를 결정함
        for (int i = 0; i < Vnum; i++)
                next[i] = adjList[i];   //      next 배열 초기화

        for (int i = 0; i < Vnum; i++) {
                if (components[i] == NONE) // 아직 어느 component에 속하는지 알지 못함
                        dfs(i);
        }
}
```

그래프의 모든 정점에 대하여 연결 요소를 결정해주는 함수이다. 이를 위해 최악의 경우 모든 정점에 대하여 dfs 함수를 호출한다.(모든 정점이 서로 다른 연결 요소일 경우) dfs를 위해 next 배열을 초기화해준 뒤에 모든 정점의 components를 확인한다. 만약 NONE일 경우 해당 정점이 아직 어떤 연결 요소에 포함되었는지 결정되지 않았으므로 dfs 함수를 호출한다. NONE이 아닐 경우 이미 해당 정점은 연결 요소가 결정되었으므로 넘어간다. 따라서 dfs 함수는 연결 요소의 개수만큼 호출된다. 연결 요소가 C개 존재할 때, 각 연결 요소별로 정점의 개수와 간선의 개수가 따로 존재한다. 이들을 모두 합하면 각각 전체 그래프의 정점의 개수 V와 간선의 개수 E가 된다. 따라서 전체 시간 복잡도는 O(V + E)가 된다.

```
void Graph::makeComponents() {
	comp = new Component[Cnum];
	for (int i = 0; i < Cnum; i++)
		comp[i] = Component(i);
	for (int i = 0; i < Vnum; i++) {
		comp[components[i]].CVnum++;
	}
}
```

findCompoents 함수에서 각 정점이 속한 연결 요소를 알아낸 것을 바탕으로 comp 배열을 동적 할당 후 각 연결 요소에 속한 정점 수만큼 연결 요소의 정점 개수를 증가시킨다. 그래프의 연결 요소 개수가 C일 때, 1 <= C <= V이므로 시간 복잡도는 O(V)이다.

```
void Graph::makeDisjointSets() {
	ds = new DisjointSet(Vnum);	//	disjoint sets 객체 만듬(초기 집합 개수 = |V|)
	for (int i = 0; i < ds->numOfSets; i++)
		ds->disjointSets[i] = Set(i);
}
```

정점의 개수만큼 원소가 한 개인 개별 집합을 가지는 DisjointSet 객체를 동적 할당한다. ds의 disjointSets 배열에 각 정점 번호를 가지는 Set 객체를 생성해 저장한다.

```
bool Graph::heapEmpty() {
	return !(heapNum >= 1);
}
```

힙이 비었는지 확인한다.

```
void Graph::adjustHeap(long long i) {
	long long child;
	long long rootKey = edgeArray[heap[i]].weight;
	long long temp = heap[i];
	child = 2 * i;

	while (child <= heapNum) {
	  if ((child < heapNum) &&
	    edgeArray[heap[child]].weight >= edgeArray[heap[child + 1]].weight)
		child++;
	  if (rootKey <= edgeArray[heap[child]].weight) break;
	  else {
		heap[child / 2] = heap[child];
		child *= 2;
	  }
	}
	heap[child / 2] = temp;
```

```
}
```

heap에 저장된 원소들 중 i번째 원소를 루트로 하는 서브 이진 트리를 최소 힙으로 만드는 함수이다. i번째 원소를 최대 2개의 자식 원소와 비교하여 이 중에서 key 값인 힙의 원소인 번호에 해당하는 간선의 weight가 더 작은 자식이 있을 경우 해당 자식 원소가 트리의 위로 올라오게 된다. 원래 i번째 원소가 알맞은 위치에 도착할 때까지 반복한다. 루트의 깊이를 1, 이진 트리의 최대 깊이를 k라 할 때 원소 i의 깊이를 h라 할 때, adjust의 시간 복잡도는 O(k-h)이다.

```
void Graph::makeHeap() {       //      min-heap 만듬
        for (long long i = heapNum / 2; i > 0; i--)
                adjustHeap(i);
}
```

adjust 함수를 호출하여 heap에 저장된 원소들을 조정하여 최소 힙으로 만든다. 자식 원소가 존재하는 원소부터 루트까지 adjustHeap 함수를 호출한다.

adjust 함수를 이용한 전체 시간 비용을 I라 하면 I는 다음과 같다.

$$I \leq \sum_{i=1}^{k} (k-i)2^{i-1}$$

이 때 k는 트리가 full binary tree일 때의 깊이이다. 트리의 원소의 개수가 N이면 k는 $\log_2(N+1)$이고 우변의 값은 $2^k - k - 1$이므로 N로 나타내면 $N - \log_2(N+1)$이 된다.

따라서 힙의 원소의 개수가 N개일 때, makeHeap 함수의 시간 복잡도는 O(N)이다. 힙에 그래프의 간선이 저장되므로 간선의 개수는 E개일 때, makeHeap의 시간 복잡도는 O(E)이다.

```
long long Graph::heapDelete() {
  if (heapEmpty()) {
        return NONE;
  }
  else if (heapNum == 1) {
        heapNum = 0;
        return heap[1];
  }
  else {
        long long ret = heap[1];
        long long temp = heap[heapNum--];
        long long parent = 1, child = 2;

        while (child <= heapNum) {
          if ((child < heapNum) &&
            (edgeArray[heap[child]].weight >= edgeArray[heap[child + 1]].weight))
              child++;
          if (edgeArray[temp].weight <= edgeArray[heap[child]].weight) break;
          else {
              heap[parent] = heap[child];
              parent = child;
              child = child * 2;
          }
        }

        heap[parent] = temp;
```

```
        return ret;
    }
}
```

최소 힙에서 원소를 삭제하는 함수이다. ret에 최소 힙의 루트(heap[1])에 저장된 간선 번호를 저장한다. 이 후 힙의 마지막 원소를 루트에 위치 시킨 후 자식 노드들과 비교하여 알맞은 위치를 찾는다. 알맞은 위치를 찾는 방법은 adjust와 유사하다. 힙의 조정이 끝난 후 ret를 반환한다.

heapDelete 함수의 시간 복잡도는 O(V)이다.

```
void Graph::kruskal() {
        CHECK_TIME_START;
        makeHeap();     //      O(|E|)
        CHECK_TIME_END(make_heap_duration);

        CHECK_TIME_START;
        makeDisjointSets();     //      O(|V|)

        int end_flag;
        int vf, vr;
        long long e, w;

        while (true) {
                end_flag = 0;
                for (int i = 0; i < Cnum; i++)
                        if (comp[i].CEnum == comp[i].CVnum - (long long)1)
end_flag++;     //      어떤 component의 mst가 완성될 경우 end_flag 1 증가
                if (end_flag == Cnum) break;    //      모든 component의 mst가 완성될
경우

                e = heapDelete();       //      O(log|E|)
                vf = edgeArray[e].v1;
                vr = edgeArray[e].v2;

                if (components[vf] != components[vr]) {
                        return;
                }
                int compNum = components[vf];

                comp[compNum].kscanned++;       //      처리한 edge 수 증가
                if (ds->find(vf) != ds->find(vr)) {     //      edge의 두 정점이 서로
같은 집합에 속하지 않는 경우
                        edgeArray[e].flag = compNum;    //      해당 component의 mst의
edge임
                        comp[compNum].CEnum++; //      component에 간선 추가
                        comp[compNum].mstCost += edgeArray[e].weight; //      mst
cost에 추가
                        ds->unionByRank(vf, vr);        //      두 집합을 합침
                }
        }
        CHECK_TIME_END(kruskal_duration);
}
```

위의 함수들을 이용하여 Kruskal Algorithm을 구현한 함수이다. 그래프가 만들어지고 모든 정점들

이 각자 속한 연결 요소를 찾아 Component 객체들이 생성된 뒤에 호출되어야 한다. 호출되면 makeHeap 함수를 호출하여 간선의 weight를 key로 하여 최소 힙을 생성한다. 힙 heap에 저장된 것은 간선의 번호로 번호로 edgeArray에서 해당 간선을 찾아 weight를 확인한다. 최소 힙을 구축한 뒤에 makeDisjointSets 함수를 호출하여 정점의 개수만큼의 개별 집합을 생성한다. 이후 그래프의 모든 연결 요소의 MST가 생성될 때까지 최소 힙 heap에서 간선 번호를 제거 후 가져온다. 가져온 번호의 간선을 확인해 두 정점 vf, vr을 알아낸 뒤 find 연산을 통해 vf와 vr이 같은 집합에 속하는지 확인 후 같은 집합에 속하면 간선을 추가 시 cycle이 생성되므로 추가하지 않는다. 서로 다른 집합에 속하면 간선을 추가할 수 있으므로 간선을 추가한다. 이 때 해당하는 연결 요소의 간선 개수를 1 증가시키고 mstCost에 추가한 간선 비용을 더해준다. unionByRank 함수를 이용해 두 정점이 속한 집합을 합쳐 하나의 집합으로 만든다. 간선의 추가 여부에 관계 없이 최소 힙에서 간선을 제거할 때마다 해당 간선은 처리된 것이므로 해당 연결 요소의 kscanned를 1 증가시킨다. 매번 반복문의 시작마다 그래프의 모든 연결 요소의 MST가 완성되었는지 확인 후 완성되었으면 반복을 종료한다.

Kruskal Algorithm의 시간 복잡도는 간선 번호를 저장한 최소 힙을 구성하는데 O(E), disjoint sets을 구성하는데 O(V), 최소 힙에서 weight가 최소인 간선을 제거하여 가져오는 비용이 O(logE)이므로 간선을 가져오는 총 비용 $O(k_{scanned}logE)$, 힙에서 간선을 가져온 후 간선의 두 정점이 같은 집합에 속하는 지 확인 후 합치는 과정이 O(logV)이므로 총 비용 $O(k_{scanned}logV)$이다. 따라서 이들을 모두 합하면 $O(E + V + k_{scanned}logE + k_{scanned}logV)$이다. 이 때

$$0 \leq E \leq \frac{V(V-1)}{2}$$

이므로 $O(logE) <= O(logV^2) = O(2logV) = O(logV)$이다. 또한 $V - 1 <= k_{scanned} <= E$ 이므로

Kruskal Algorithm의 시간 복잡도는 O(ElogV)가 된다.

```
Graph::~Graph() {
	for (int i = 0; i < Vnum; i++) {
		while (adjList[i]) {
			EdgeNode* del = adjList[i];
			adjList[i] = del->next;
			delete del;
		}
	}
	delete[] adjList;
	delete[] next;
	delete[] edgeArray;
	delete[] components;
	delete[] heap;
	delete st;
}
```

프로그램이 종료되기 전 동적할당된 Graph 객체를 해제하는 소멸자이다. 인접 리스트 adjList의 모든 노드를 소멸 후 동적 할당된 모든 배열 adjList, next, edgeArray, components, heap과 스택 st를 해제한다. 시간 복잡도는 O(V + E)이다.

```
#include <iostream>
```

```
#include <fstream>
#include <string>
#include <cstdio>
#include <Windows.h>

#define CHECK_TIME_START QueryPerformanceFrequency ((_LARGE_INTEGER*)&freq);
QueryPerformanceCounter((_LARGE_INTEGER*)&start)
#define CHECK_TIME_END(a) QueryPerformanceCounter((_LARGE_INTEGER*)&end);
a=(float)((float) (end - start)/freq)
_int64 start, freq, end;
#define NONE    -1

float total_duration = 0;
float make_graph_duration = 0;
float construct_graph_duration = 0;
float component_duration = 0;
float kruskal_duration = 0;    //      make_heap 제외
float make_heap_duration = 0;
float delete_duration = 0;

class Set { … };
class DisjointSet { … };
class Edge { … };
class EdgeNode { … };
class StackNode { … };
class Stack { … };
class Component { … };
class Graph { … };

int main(void) {
        std::string cmd_file = "commands.txt";
        std::string input_file, output_file;
        std::ifstream ifs(cmd_file);
        if (ifs.fail()) {
                std::cout << "Command File Open Error!" << std::endl;
                return -1;
        }
        ifs >> input_file >> output_file;
        ifs.close();

        ifs.open(input_file);
        std::ofstream ofs(output_file);
        if (ifs.fail()) {
                std::cout << "Input File Open Error!" << std::endl;
                return -1;
        }

        Graph* graph = NULL;
        int Vnum;
        long long Enum, Wmax;

        ifs >> Vnum >> Enum >> Wmax;
        std::cout << "\n" << input_file << "\n";
        std::cout << "vertex 개수 : " << Vnum << "\n" << "edge 개수 : " << Enum <<
"\n" << "max-weight : " << Wmax << "\n\n";

        CHECK_TIME_START;
        graph = new Graph(Vnum, Enum, Wmax);
        CHECK_TIME_END(make_graph_duration);
```

```
        CHECK_TIME_START;
        for (long long i = 0; i < graph->Enum; i++) {
                int v1, v2;
                long long weight;
                ifs >> v1 >> v2 >> weight;
                graph->insertEdge(i, v1, v2, weight);
        }
        CHECK_TIME_END(construct_graph_duration);

        CHECK_TIME_START;
        graph->findComponents();
        graph->makeComponents();
        CHECK_TIME_END(component_duration);
        graph->kruskal();
        ofs << graph->Cnum << "\n";
        for (int i = 0; i < graph->Cnum; i++)
                ofs << graph->comp[i].CVnum << " " << graph->comp[i].mstCost <<
"\n";
        for (int i = 0; i < graph->Cnum; i++) {
                std::cout << "component " << graph->comp[i].Cname << " 정보" << "\n";
                std::cout << "Vertex 개수 : " << graph->comp[i].CVnum << "\n";
                std::cout << "MST Edge 개수 : " << graph->comp[i].CEnum << "\n";
                std::cout << "MST cost : " << graph->comp[i].mstCost << "\n";
                std::cout << "kscanned : " << graph->comp[i].kscanned << "\n\n";
        }

        CHECK_TIME_START;
        delete graph;
        CHECK_TIME_END(delete_duration);

        total_duration = make_graph_duration + construct_graph_duration +
component_duration
                + make_heap_duration + kruskal_duration + delete_duration;
        printf("make-graph duration : %f(s)\n", make_graph_duration);
        printf("construct-graph duration : %f(s)\n", construct_graph_duration);
        printf("component duration : %f(s)\n", component_duration);
        printf("make-heap duration : %f(s)\n", make_heap_duration);
        printf("kruskal_duration : %f(s)\n", kruskal_duration);
        printf("delete-graph duration : %f(s)\n", delete_duration);
        printf("total_duration : %f(s)\n", total_duration);

        ifs.close();
        ofs.close();

        return 0;
}
```

main 함수에서 command.txt 파일을 통해 그래프가 저장된 파일을 열어 그래프의 정점의 개수, 간선의 개수, 간선의 최대 비용을 입력 받아 이를 이용해 Graph 객체를 동적 할당한다. 그래프 객체를 생성 후 그래프 파일의 간선 개수만큼 간선 정보를 읽어 insertEdge 함수로 그래프에 간선을 추가한다. 간선을 추가하여 그래프가 완성되면 findComponents 함수를 호출해 DFS로 연결 요소를 모두 찾고, makeComponents 함수로 연결 요소를 만든다. 이 후 kruskal 함수를 호출하여 Kruskal Algorithm으로 그래프의 각 연결 요소의 MST를 찾아낸다. MST를 찾은 후, 동적 할당한 그래프 객체를 해제한다.