[알고리즘 설계와 분석]

2023 학년도 1 학기 – HW4

학번	20190785
이름	박수빈
과목코드	CSE3081
분반	01
담당 교수님	임인성 교수님

1. 실험 환경

OS: Windows 11 Home (65-bit)

CPU: Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz

RAM: 8.00GB

Compiler: Visual Studio 2022 (x64 Release Mode)

2. 구현 방법

disjoint set을 이용하여 Kruskal's algorithm을 구현하기 위해 사용한 자료구조는 다음과 같다.

```
typedef struct {
    int src, dest;
    int weight;
}Edge;
Edge* graph;

typedef struct {
    int parent;
    int64_t weightSum;
}Subset;
Subset* disjoint_sets;

typedef struct {
    int root;
    int64_t weightSum;
    int mst_vertices;
}Result;
```

input file로부터 입력 받은 edge정보들을 Edge 구조체에 저장하였다. 그리고 이 Edge들로 구성된 그래프는 undirected graph이므로 edge list로 구현하였다. 그래서 Edge* graph로 선언해주었다. 그리고 disjoint set을 이용하기 위해 disjoint set의 subset들을 parent와 weightSum을 멤버 변수로 갖는 구조체로 구현하였다. parent에는 자신의 부모에 해당하는 vertex의 번호를 저장하거나, 부모가 없는 경우에는 (-1)*(본인을 root로 삼는 subset의 개수)를 저장하였다. 마지막으로, 결과를 저장하기 위해 Result 구조체를 만들어 사용하였다. MST를 만들기 위해 사용된 disjoint set의 root인 vertex의 번호를 'root'에, MST를 이루는 정점의 개수를 'mst_vertices'에, MST를 구성하는 모든 간선의 weight의 합을 'weightSum'에 저장하는 형태이다.

이러한 자료구조들을 사용하여 Kruskal's algorithm을 구현하는 함수들은 아래와 같다.

```
Edge* create_graph(int n_vertices, int n_edges);
Subset* init_disjoint(Subset* disjoint_sets);
void insert_minheap(Edge* graph, Edge edge, int* n_heap);
Edge delete_minheap(Edge* graph, int* n_heap);

int Find(Subset* disjoint_sets, int i);
void Union(Subset* disjoint_sets, int i, int j, int64_t edge_weight);
int kruskal_mst(Edge* graph);
Result* calculate(Edge* graph, Subset* disjoint_sets, Result* result);
```

1) Edge* create_graph(int n_vertices, int n_edges);

```
Edge* create_graph(int n_vertices, int n_edges){
    graph = (Edge*)malloc(sizeof(Edge) * (n_edges+1));
    if (!graph) {
        fprintf(stderr, "Memory allocation for graph has been failed.₩n");
        exit(1);
    }

    graph[0].src = n_vertices;
    graph[0].dest = n_edges;

    return graph;
}
```

그래프를 초기에 생성하는 데 사용되는 함수이다. 참고로, 본 과제에서는 heap sort를 이용하여 간선들을 weight이 감소하지 않는 순서로 정렬할 것이기 때문에 graph를 min heap으로 구현하였다. 그렇기 때문에 0번째가 아닌 1번째 원소부터 Edge 정보들을 입력받는다. 0번째 원소를 비워둘 수도 있지만, 임의로 scr 자리에는 n_vertices(총 정점의 개수)와 dest 자리에는 n_edges(총 간선의 개수)를 저장하였다.

2) Subset* init_disjoint(Subset* disjoint_sets);

```
Subset* init_disjoint(Subset* disjoint_sets) {
    disjoint_sets = (Subset*)malloc(sizeof(Subset) * n_vertices);
    if (!disjoint_sets) {
        fprintf(stderr, "Memory allocation for disjoint sets has been failed.₩n");
        exit(1);
    }

    for (int i = 0; i < n_vertices; i++) {
        disjoint_sets[i].parent = -1;
        disjoint_sets[i].weightSum = 0;
    }
    return disjoint_sets;
}
```

disjoint set을 이루는 subset들을 초반에 초기화시켜줄 때 사용하는 함수이다. 앞서 서술했듯이, subset의 parent에는 자신의 부모에 해당하는 vertex의 번호를 저장하거나, 부모가 없는 경우에는 (-1)*(본인을 root로 삼는 subset의 개수)를 저장한다. 여기서 -1이 곱해지는 이유는 위의 함수 코드를 보면 알 수 있다. 모든 subset들의 parent 값이 -1로 초기화된 상태에서 시작하기 때문이다. 더 자세한 설명은 이후에 Union 함수에서 할 예정이다.

3) void insert_minheap(Edge* graph, Edge edge, int* n_heap);

```
void insert_minheap(Edge* graph, Edge edge, int* n_heap) {
    if ((*n_heap) == n_edges) {
        fprintf(stderr, "The min heap is full.₩n");
        exit(1);
    }
    (*n_heap)++;

    int i = (*n_heap);
    while (1) {
        if (i == 1) break; //If number of elements is 1, there's nothing to compare
        //'i' keep going up while it find right place to insert the data
        if (edge.weight >= graph[i / 2].weight) break;
        graph[i] = graph[i / 2];
        i /= 2;
    }
    //Finally, i find right place and insert the data into the place
    graph[i] = edge;
    return;
}
```

앞서 언급했듯이, 본 과제에서는 heap sort를 이용해 graph에서 간선의 weight이 증가하지 않는 순서로 정렬한다. 따라서 입력 파일로부터 간선의 정보를 입력 받을 때, 위의 insert_minheap함수를 통해 graph에 삽입하는 방식으로 입력 받는다. 인자로 전달받은 'edge'를 graph라는 max heap에 삽입하는 함수이다. min heap의 가장 마지막 자리에서 시작하여 edge의 weight 값보다 작은 값을 마주칠 때까지 계속해서 heap의 root 쪽으로 올라가는 식으로 자리를 찾는다. 그렇게 부모 자리로 계속 올라가는 과정에서 부모에 있던 노드를 자식 자리로 옮겨놓는 작업도 동시에 진행한다. 마침내 적합한 자리를 찾으면 그 곳에 edge를 삽입하고 함수를 종료한다.

4) Edge delete_minheap(Edge* graph, int* n_heap);

```
Edge delete_minheap(Edge* graph, int* n_heap) {
    if ((*n_heap) == 0) {
        fprintf(stderr, "The heap is empty.\n");
        exit(1);
    }

    //Save first element in 'first'
    Edge first = graph[1];

    //Save the last element in 'last'
    Edge last = graph[(*n_heap)];
    (*n_heap)--;

    //Starting point of parent and child
    int parent = 1, child = 2;

    while (1) {
        if (child > (*n_heap)) break;

        //Check which one is smaller among left child and right child
        if (child <= (*n_heap) - 1) {
            if (graph[child + 1].weight < graph[child].weight) ++child;
        }

        //Find the place where 'last' should be inputed
        if (last.weight <= graph[child].weight) break;

        //Move the 'child' up to 'parent'
        graph[parent] = graph[child];

        //Move to down in the heap
        parent = child;
        child = child * 2;
    }

    //Input the 'last' into the place where we had found in previous process
    graph[parent] = last;

    //Return the element what we have removed from the heap
    return first;
}
```

min heap에서 가장 첫 번째 원소를 뽑아낸 다음, 남은 원소들을 다시 min heap의 특성을 유지하도록 정렬해 주는 함수이다. 가장 첫 번째 원소를 뽑아서 first에 저장해 놓은 다음 나중에 함수가 끝나면 이를 반환하도록 한다. 첫 번째 원소를 뽑은 다음에는, heap의 가장 마지막 원소를 가장 첫 번째 원소의 자리로 집어넣은 다음, 자식들과 계속 비교하며 자식들보다 본인이 커지기 직전까지 내려가서 본인의 자리를 찾는 형식으로 min heap의 특성을 유지한다. 이와 같은 delete_minheap 함수는 insert_minheap 함수와 같이 쓰이면 heap sort라는 정렬 방법으로 이처럼 쓰일 수 있는데, 이 정렬 방법은 원소를 추가하거나 삭제할 때 모두 O(nlogn)이라는 시간 복잡도를 지닌다. disjoint set을 만들 때와 같이 원소의 추가와 삭제 모두가 빈번하게 일어나는 경우 heap sort가 매우 적절하다. (왜냐하면 다른 정렬 방법들은 추가 시 시간 복잡도가 매우 낮으면 삭제 시 시간 복잡도가 매우 높은 식으로, 한 쪽으로 치우친 경우가 많기 때문이다.)

5) int Find(Subset* disjoint_sets, int i);

```
int Find(Subset* disjoint_sets, int i) {
    while (disjoint_sets[i].parent >=0 ) {
        i = disjoint_sets[i].parent;
    }
    return i;
}
```

이 Find 함수를 구현하는 데는 path compression, recursion 등 여러 방법이 있지만, 본 과제에서는 iteration을 사용하여 구현하였다. parent의 값이 음수가 나올 때까지, 즉 자신이 root node인 원소를 찾을 때까지 while loop을 반복하는 식으로 i 원소의 조상을 찾는 함수이다.

6) void Union(Subset* disjoint_sets, int i, int j, int64_t edge_weight);

```
void Union(Subset* disjoint_sets, int i, int j, int64_t edge_weight) {
    //find the root node of i and j
    int i_root = Find(disjoint_sets, i);
    int j_root = Find(disjoint_sets, j);

    int tmp = disjoint_sets[i_root].parent + disjoint_sets[j_root].parent;

    if (disjoint_sets[i_root].parent > disjoint_sets[j_root].parent) {
        disjoint_sets[i_root].parent = j_root;
        disjoint_sets[j_root].parent = tmp;
        disjoint_sets[j_root].weightSum += disjoint_sets[i_root].weightSum + edge_weight;
    }
    else if (disjoint_sets[i_root].parent < disjoint_sets[j_root].parent) {
        disjoint_sets[j_root].parent = i_root;
        disjoint_sets[i_root].parent = tmp;
        disjoint_sets[i_root].weightSum += disjoint_sets[j_root].weightSum + edge_weight;
    }
    else {
        if (i_root < j_root) {
            disjoint_sets[j_root].parent = i_root;
            disjoint_sets[i_root].parent = tmp;
            disjoint_sets[i_root].weightSum += disjoint_sets[j_root].weightSum + edge_weight;
        }
        else {
            disjoint_sets[i_root].parent = j_root;
            disjoint_sets[j_root].parent = tmp;
            disjoint_sets[j_root].weightSum += disjoint_sets[i_root].weightSum + edge_weight;
        }
    }
    return;
}
```

i를 포함하는 subset와 j를 포함하는 subset를 합치는 역할을 하는 함수이다. 우선 i와 j의 root node를 찾은 다음, 이 두 root node 중 parent의 값이 작은 subset에 큰 subset를 포함시키는 방식으로 set을 합친다. parent의 값이 작은 subset에 큰 subset를 합치는 이유는, 그 반대로 하면 나중에 Find 함수로 어떤 원소의 root 노드를 찾을 때 한참 올라가야 하는, 즉 while loop을 많이 돌려야 해서 시간 복잡도가 많이 증가하는 문제가 발생할 수 있기 때문이다. parent의 값이 작다는 것은 음수라면 그만큼 그 set에 포함되어 있는 subset이 많다는 뜻이고, subset이 작은 쪽이 큰 쪽으로 합쳐지는 것이 앞서 말했듯이 시간 복잡도 측면에서 더 유리하다. 그리고 이렇게 합쳐지는 과정에서 parent 값만 갱신해주는 것이 아니라 weightSum값도 갱신해줘야 한다. 현재 그 subset에 포함되어 있는 간선들의 weight의 합이 얼마인지 계산을 해줘야 하기 때문이다.

7) int kruskal_mst(Edge* graph);

```
int kruskal_mst(Edge* graph) {
    Edge obj;
    int k_scanned = 0; //탐색한 간선의 개수
    int edge_count = 0; //MST로 선택된 간선의 개수

    int x, y;
    while (edge_count<n_vertices-1 && n_heap>0) {

        k_scanned++;

        obj = delete_minheap(graph, &n_heap);

        x = Find(disjoint_sets, obj.src);
        y = Find(disjoint_sets, obj.dest);

        if (x != y) {
            Union(disjoint_sets, x, y, obj.weight);
            edge_count++;
        }

    }
    return k_scanned;
}
```

앞서 작성했던 함수들을 이용하여 본격적으로 Kruskal's algorithm을 구현하는 함수이다. MST를 구성하는 간선으로 선택된 간선의 개수를 변수 edge_count에 저장한다. 그리고 이 edge_count가 총 정점의 개수(n_vertices)에서 1을 뺀 것보다 작고, heap(graph)의 개수가 0보다 크다면 while loop을 계속 실행한다. while loop에서는 우선 min heap에서 첫 번째 원소를 뽑는다. 그리고 그 원소가 MST에 포함되는지 확인하는 과정을 disjoint set operation을 통해 실행한다. 간선의 양 끝점이 각각 같은 subset에 포함된다면, 이는 해당 간선이 MST에 추가되면 cycle을 형성한다는 뜻이므로 포함시키지 않는다. 오로지 이 양 끝점이 다른 subset에 속해있는 경우에만 Union 함수를 통해 같은 subset으로 합쳐준다. 그런 다음 edge_count를 1만큼 증가시켜준다. 이러한 while loop이 실행될 때마다 변수 k_scanned의 값은 1씩 증가하고, while loop이 끝나면 함수가 끝나면서 이 k_scanned를 반환한다.

8) Result* calculate(Edge* graph, Subset* disjoint_sets, Result* result);

```
Result* calculate(Edge* graph, Subset* disjoint_sets, Result* result) {

    int i,j;
    int idx = 0;

    //각 component의 root를 찾은 다음, 그 root가 가지고 있는 정보인 weightSum을 저장한다.
    for (i = 0; i < n_vertices; i++) {
        if (disjoint_sets[i].parent < 0) {
            result[idx].root = i;
            result[idx].weightSum = disjoint_sets[i].weightSum;
            result[idx].mst_vertices = (-1)*disjoint_sets[i].parent;
            idx++;
        }
    }

    return result;
}
```

이 함수를 호출하기 전에는 앞서 소개했던 kruskal_mst 함수를 실행하고 난 뒤, 아래의 과정을 먼저 거쳐야 한다. disjoint_sets을 처음부터 끝까지 살피며 parent의 값이 음수인 것이 몇 개인지 세야 한다. parent의 값이 음수라는 것은 해당 노드가 MST를 이루는 disjoint set의 root 노드라는 뜻이고, 이 root 노드가 2개 이상이라는 것은 해당 graph가 connected component를 갖는다는 뜻이기 때문에 이를 세기 위해 이 값을 n_component에 저장해주는 것이다. 이렇게 n_component 값을 알아낸 뒤에는, Result형 1차원 배열인 result를 n_component 크기만큼 동적 할당해준다. 이 result 배열은 결과를 계산하기 위한 배열이다.

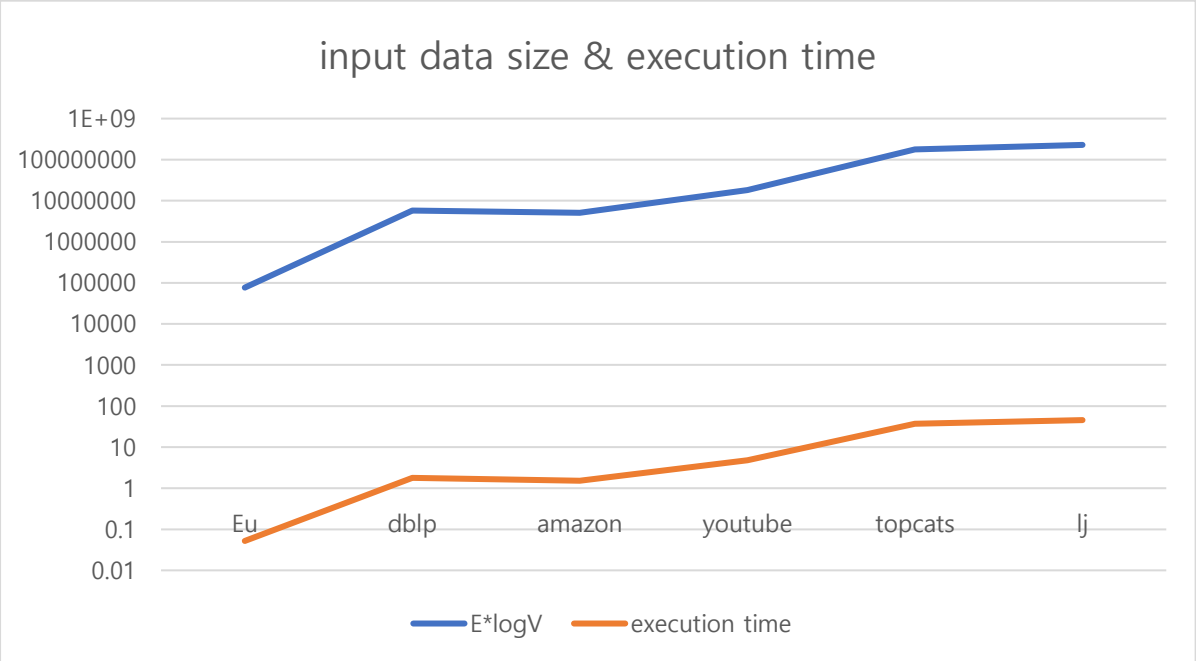
```
for (int i = 0; i < n_vertices; i++) {
    if (disjoint_sets[i].parent < 0) n_component++;
}
Result* result = (Result*)malloc(sizeof(Result) * n_component);
```

이러한 과정을 거친 뒤에, calculate 함수를 호출하여 result 배열을 채우는 것이다. disjoint_sets을 순회하면서 parent의 값이 0보다 작은, 즉 MST의 root 노드에 해당하는 것이 발견된다면 그 노드의 번호를 root에 저장해주고, weightSum값도 저장해준다. 그리고 앞서 언급했듯이, parent에 절댓값을 취한 값이 해당 MST를 구성하는 정점의 개수이므로 parent의 값에 (-1)을 곱한 값을 mst_vertices에 저장해준다. 이로써 result에는 disjoint set을 이용한 Kruskal's algorithm을 통해 찾아낸 MST의 정보들이 담기게 되는 것이다.

3. 실험 결과

파일 이름	작동 여부	MST weight	수행 시간 (초)	Kscanned
HW3_email-Eu-core.txt	YES	3110161	0.0520	25571
HW3_com-dblp.ungraph.txt	YES	2747895457	1.7920	1049834
HW3_com-amazon.ungraph.txt	YES	2729670156	1.5080	925855
HW3_com-youtube.ungraph.txt	YES	14578691475	3.1540	2987623
HW3_wiki-topcats.txt	YES	5351181035	36.8840	28415895
HW3_com-lj.ungraph.txt	YES	28308045762	45.5230	34681165

4. 시간 복잡도



본 과제에서 구현한 프로그램의 시간 복잡도는 O(E*logV)이다. (E는 간선의 개수, V는 정점의 개수) 그 이유는 우선 간선들을 min heap에 삽입하는 과정은 O(logE)만큼의 시간이 걸리는데, 이를 E번 행하므로 총 O(ElogE)만큼의 시간이 걸린다. 이후 kruskal_mst 함수에서 (V-l)회 (엄밀히 말하자면 k_scanned회) min heap에서 뽑아내는 작업(O(logE))을 하고, 두 번의 Find 함수(O(logV))를 실행하고, 어떤 경우에는 Union 함수(O(1))도 실행한다. 따라서 총 O(ElogV)만큼의 시간 복잡도를 지니

게 된다. 그리고 위의 그래프에서도 확인할 수 있듯이, 본 실험에서 측정한 실제 프로그램 실행 시간이 이러한 시간 복잡도를 반영한다고 결론지을 수 있다. (참고로, 스케일 조정을 위해 y축을 로그 눈금 간격으로 설정하였습니다.)