알고리즘 설계와 분석(CSE3081) HW4

Minimum Spanning Tree 알고리즘의 구현

담당 교수: 임인성

이름: 남기동

학번: 20180032

1. 결과 및 분석

파일 이름	작동 여부	MST weight	수행 시간(초)	K*scanned*
HW3_email-Eu-core	YES	3110161	0.007	25571
HW3_com-amazon.ungraph	YES	2729637388	0.505	925863
HW3_com-dblp.ungraph	YES	2747862690	0.601	1049824
HW3_com-youtube.ungraph	YES	14578658707	2.093	2987623
HW3_wiki-topcats	YES	5351148376	25.661	28416106
HW3_com-lj.ungraph	YES	28308012994	31.261	34681189

위의 표는 주어진 파일 6개를 토대로 Kruskal Algorithm을 구현한 코드를 사용하였을 때 작동하는지의 여부와 구해진 MST weight, 수행 시간, 그리고 MST를 구현하기까지 스캔한 edge의 개수를 나타내는 Kscanned로 구성되어 있는 결과표이다. 결과적으로 주어진 6개의 자료 모두에 대하여 코드가 작동하였음을 확인할 수 있다. 그리고 이러한 결과로부터 Kruskal Algorithm의 시간 복잡도라고 배운 O(E log V)가 실질적으로 반영되고 있는지를 파악하기 위해 아래의 표를 추가적으로 만들었다.

파일 이름	Vertex 개수	Edge 개수	E log2 V	수행 시간(초)
HW3_email-Eu-core	1,005	25,571	255,017	0.007
HW3_com-amazon.ungraph	334,863	925,872	16,992,713	0.505
HW3_com-dblp.ungraph	317,080	1,049,866	19,185,671	0.601
HW3_com-youtube.ungraph	1,134,890	2,987,624	60,093,367	2.093
HW3_wiki-topcats	1,791,489	28,511,807	592,267,213	25.661
HW3_com-lj.ungraph	3,997,692	34,681,189	760,582,751	31.261

우선, 첫 번째의 파일에는 다른 파일에 비해 vertex의 개수, edge의 개수도 현저히 작으며 수행 시간도 현저히 작기 때문에 비교에서 제외하였다. 따라서 비교는 2번째 파일부터 6번째 파일까지로 진행하였으며 위의 표와 같은 결과값을 얻었다.

2번째를 기준으로 3, 4, 5, 6번째 파일의 E log V를 비교하여 증가 비율을 구하고 그것이 수행 시간의 증가 비율과 동일한지 비교하는 방식으로 O(E log V)라는 시간 복잡도가 실질적으로 반영되었는지를 확인하고자 하였다.

수행 시간의 증가 비율을 구하면 다음과 같다.

1배, 1.19배, 4.14배, 50.81배, 61.9배

다음으로 E log2 V의 증가 비율을 구하면 다음과 같다.

1배, 1.12배, 3.5배, 34.85배, 44.75배

이처럼 수행 시간의 증가 비율과 E log2 V의 증가 비율이 대체적으로 유사함을 확인할 수 있다. 물론 약간의 차이는 발생하며 그 차이는 파일이 포함하고 있는 vertex와 edge의 크기가 증가함에 따라 커지는 것처럼 보인다. 이는 기존의 Kruskal Algorithm의 경우와 달리 구현한 코드에 추가적인 연산이 있기 때문이라 예상된다. 추가된 연산으로는 여러 개의 connected graph가 있을 수도 있기 때문에 subset을 통해 union-find의 연산을 두 번 수행한다는 점과 weight를 더하는 과정에서도 그렇게 만들어진 connected graph를 모두 살펴보는 과정이 있다. 이러한 부분 때문에 파일의 크기가 커질수록 해당 부분들을 계산하고 처리하는데 소요되는 시간이 조금씩 늘어나기 때문에 실질적인 수행 시간의 증가 비율이 이론적인 E log2 V의 증가 비율보다 큰 것이라 생각해볼 수 있다.

2. 구현

```
typedef struct Edge* edge_pointer;
typedef struct Edge {
    int from;
    int to;
    int weight;
}edge;
typedef struct Subset {
    int parent;
    int rank;
    int root;
}subset;
typedef struct Group {
    int root;              //해당 group의 root
    int comp_cnt;          //해당 group에 속한 component의 개수
    long long weight_total; //해당 group의 MST를 구성하는 weight의 총합
}Group;

int n_vertex;           //ASCII 파일에서 읽어온, vertex의 개수
int n_edge;             //ASCII 파일에서 읽어온, edge의 개수
int MAX_WEIGHT;         //ASCII 파일에서 읽어온, edge의 weight가 가장 가질 수 있는 가장 큰 값
edge_pointer Edge_Arr;  //Edge의 정보를 저장하는 배열
Group* group;           //connected graph 단위로 저장하는 배열
int group_cnt = 0;      //connected graph의 개수
int k_scanned = 0;      //MST를 생성할 때까지 읽은 edge의 개수
clock_t start, end;     //시간을 측정할 때 사용하는 변수

int find(subset subsets[], int i);              //해당 원소의 parent를 찾아서 반환하는 함수
void Union(subset subsets[], int x, int y);     //두 원소가 속한 집합을 병합(merge)하는 함수
void KruskalMST();                              //disjoint set의 find, union을 바탕으로 구현한 MST를 구하는 Kruskal 알고리즘
void buildMinHeap(edge_pointer arr, int n);     //Minheap을 만드는 함수
void heapify(edge_pointer arr, int n, int i);   //heap의 성질을 유지하도록 하는 함수
edge extractMin(edge_pointer arr, int* n);      //제일 작은 root node를 min heap에서 꺼내는 함수
```

우선 이번 과제로 Kruskal 알고리즘을 사용하여 MST를 계산하는 코드에 사용되는 변수와 함수에 대한 정보이다. Struct는 총 3가지가 사용되었는데 Edge는 파일에서 입력받는 간선의 정보를 저장하기 위한 구조체이며 subset은 각 vertex를 기준으로 union-find를 사용할 때 필요한 정보를 갖고 있는 구조체이다. 수업시간에 다루었던 기존 예제 코드와 다르게 root가 사용되었는데 이는 기존의 예제 코드는 1개의 connected graph를 기준으로 하는 반면 이번 과제에서 구현한 코드는 여러 개의 connected graph를 가질 수도 있기 때문이다. 따라서 subset이 1개의 root에 연결되지 않을 수도 있기 때문에 각각의 vertex가 갖게 되는 root를 저장하기 위해 root라는 변수를 subset 구조체에 추가했다. Group은 connected graph를 의미하며 앞서 이야기했듯 여러 개의 connected graph가 있을 수도 있기 때문에 만든 구조체이고 Group* group이라는 변수를 통해 connected graph의 개수만큼 크기를 할당해서 사용할 예정이다.

N_vertex, n_edge, MAX_WEIGHT는 파일의 첫 번째 줄에서 입력받는 edge들의 기본 정보이고 edge_pointer Edge_Arr는 그 edge들의 정보를 저장하는 변수이며 group_cnt는 connected graph의 개수, k_scanned는 MST를 생성할 때까지 읽은 edge의 개수를 의미한다.

사용하는 함수로는 총 6개가 있다. Find, Union은 예제 코드를 참조하였으며 Kruskal 알고리즘을 구현하는데 핵심으로 사용되는 코드이며 find는 해당 vertex의 root를 찾는 함수이며 Union은 서로 다른 root를 갖는 두 vertex를 merge하는 함수이다. buildMinheap은 heapify함술를 사용하여 Minheap을 만드는 함수이며 heapify는 재귀적으로 사용되어 heap의 성질을 유지하도록 만드는 함수이고 extractMin은 Minheap의 root인 제일 작은 노드를 꺼내는 함수이다. 마지막으로 KruskalMST 함수는 위의 5개 함수를 사용하여 MST를 계산하는 Kruskal 알고리즘을 구현한 코드이다.

```
fscanf(fp, "%d %d %d", &n_vertex, &n_edge, &MAX_WEIGHT); //파일의 첫 줄에서 vertex의 개수, edge의 개수, 최대 weight를 입력 받는다
Edge_Arr = (edge_pointer)malloc(sizeof(edge) * n_edge);
for (int i = 0; i < n_edge; i++) { //파일의 둘째 줄부터 edge의 개수만큼 edge의 정보를 입력 받는다
    fscanf(fp, "%d %d %d", &Edge_Arr[i].from, &Edge_Arr[i].to, &Edge_Arr[i].weight);
    if (Edge_Arr[i].weight > MAX_WEIGHT || Edge_Arr[i].from > n_vertex || Edge_Arr[i].to > n_vertex) {
        fprintf(stderr, "DATA FROM FILE HAS ERROR₩n");
        exit(0);
    }
}
fclose(fp);

start = clock();
KruskalMST();
end = clock();
double total_time = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("Time spent: %f seconds₩n", total_time);
```

위의 코드는 main함수에서 연결된 File pointer를 활용하여 파일을 읽는 것과 KruskalMST 함수를 호출하는 것, 그리고 해당 알고리즘의 시간을 측정하는 부분이다. 우선 파일의 첫 줄에서 vertex의 개수, edge의 개수, 최대 weight를 입력받는다. edge 개수만큼 Edge_Arr를 동적할당 한 다음, edge의 정보를 Edge_Arr에 입력받는다. KruskalMST함수를 호출하기 전에 start = clock()을 통해 시간을 측정하고 KruskalMST함수가 리턴된 후 end에 시간을 측정하여 이를 뺀 값을 걸린 시간으로 계산하여 출력한다.

```
int find(subset subsets[], int i) { //해당 원소의 parent를 찾아서 반환하는 함수
    if (subsets[i].parent != i) //parent가 자기 자신이 아닌 경우 parent를 찾을 때까지 recursion 수행
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}
void Union(subset subsets[], int x, int y) { //두 원소가 속한 집합을 병합(merge)하는 함수
    int xroot = find(subsets, x); //x가 속한 집합의 root를 찾고
    int yroot = find(subsets, y); //y가 속한 집합의 root를 찾아

    //rank가 큰 쪽으로 merge(rank가 작은 쪽의 root의 parent를 큰 쪽의 root로 설정)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    //rank가 동일하면 한 쪽으로 merge하고 한쪽의 rank를 1 증가
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}
```

Union-Find는 예제 코드와 다르지 않다. Find에서는 주어진 i번째 vertex의 root를 recursive하게 찾는 함수이며 Union은 입력 받은 두 개의 vertex의 root를 find함수를 통해 계산한 이후 rank가 큰 쪽으로 merge하는 함수이다. Rank가 동일한 경우에는 한 쪽으로 merge하고 rank를 1 증가시킨다.

```
void buildMinHeap(edge_pointer arr, int n) { // 배열 기반으로 min heap 구성
    for (int i = n / 2 - 1; i >= 0; i--) { // 마지막 내부 노드부터 역순으로 heapify 수행
        heapify(arr, n, i);
    }
}
void heapify(edge_pointer arr, int n, int i) {
    int smallest = i;  // 현재 노드를 가장 작은 값으로 설정
    int left = 2 * i + 1;  // 왼쪽 자식 노드 인덱스
    int right = 2 * i + 2;  // 오른쪽 자식 노드 인덱스
    int tmp_weight, tmp_to, tmp_from;

    // 왼쪽 자식이 루트보다 작은 경우
    if (left < n && arr[left].weight < arr[smallest].weight) smallest = left;

    // 오른쪽 자식이 루트보다 작은 경우
    if (right < n && arr[right].weight < arr[smallest].weight) smallest = right;

    // 현재 루트와 가장 작은 값을 교환
    if (smallest != i) {
        tmp_weight = arr[i].weight; tmp_from = arr[i].from; tmp_to = arr[i].to;
        arr[i].weight = arr[smallest].weight; arr[i].from = arr[smallest].from; arr[i].to = arr[smallest].to;
        arr[smallest].weight = tmp_weight; arr[smallest].from = tmp_from; arr[smallest].to = tmp_to;

        // 변경된 자식 노드에 대해 재귀적으로 heapify 호출
        heapify(arr, n, smallest);
    }
}
edge extractMin(edge_pointer arr, int* n) { // min heap에서 최소값을 꺼내는 함수
    if (*n <= 0) {
        fprintf(stderr, "HEAP IS Empty!₩n");
        exit(0);  // 힙이 비어있을 경우 -1 반환 또는 예외 처리
    }
    edge min_edge = arr[0];  // 최소값은 루트 노드에 위치
    arr[0] = arr[*n - 1];  // 힙의 마지막 노드를 루트로 이동
    (*n)--;  // 힙의 크기를 1 감소

    heapify(arr, *n, 0);  // 힙 속성 유지를 위해 heapify 호출

    return min_edge;
```

위의 코드는 Minheap과 관련된 3개의 함수 코드이다. buildMinHeap은 minheap에 있는 노드의 개수를 기준으로 마지막 노드부터 역순으로 heapify를 반복적으로 호출하여 recursively minheap의 성질을 유지시키는 함수이다. Heapify는 minheap에 있는 노드의 개수(n)와 현재 노드(i)를 입력 받아서 현재 노드의 child 중 더 작은 값이 있다면 그 child를 smallest로 설정하고 현재 노드와 변경한 후 heapify를 호출한다. 변경되지 않은 부분에 대해서는 heapify를 호출하지 않는다. extractMin은 제일 작은 루트 노드를 꺼내고 heap의 마지막 노드를 루트 자리에 넣고 heapify를 호출하여 다시 heap이 minheap의 성질을 가지도록 하는 함수이다.

마지막으로 KruskalMST의 코드를 살펴보려 한다. KruskalMST는 총 3개의 부분으로 구성되어있다.

```
group = (Group*)malloc(sizeof(Group) * n_vertex);
subset* subsets = (subset*)malloc(n_vertex * sizeof(subset)); //Vertex 개수만큼 set의 공간을 할당(single node를 만들기 위해)
for (int i = 0; i < n_vertex; i++) { //V subset을 single node로 만든다
    subsets[i].parent = i; //single node는 자기 스스로가 자신의 parent이다
    subsets[i].rank = 0; //rank는 root로부터 들어간 depth를 의미하므로 root node, single node의 rank는 0이다
}

//Union-Find로 connected graph를 만든다
for (int i = 0; i < n_edge; i++) {
    edge next_edge = Edge_Arr[i];
    int x = find(subsets, next_edge.from); //next_edge의 출발 vertex를 포함한 집합의 root를 x로 받아옴
    int y = find(subsets, next_edge.to);   //next_edge의 도착 vertex를 포함한 집합의 root를 y로 받아옴
    if (x != y)
        Union(subsets, x, y);
}

//모든 vertex를 모두 검색하면서 connected graph가 몇 개 있는지, 해당 graph의 집합을 나타내는 root는 무엇인지 갱신
for (int i = 0; i < n_vertex; i++) {
    if (i == find(subsets, i)) {
        group[group_cnt].root = i;          //해당 connected group의 root를 저장
        group[group_cnt].weight_total = 0;  //초기화
        group[group_cnt].comp_cnt = 0;      //초기화
        group_cnt++;                        //connected group의 개수를 1 증가
    }
}
```

첫 번째 부분은 vertex의 개수만큼 group과 subsets을 동적 할당하고 subset을 각 vertex에 대한 single node로 만들어 union-find를 할 준비를 한다. 이 때 각 single node의 parent는 자기 자신이며 rank는 0으로 설정된다. 이후에는 모든 edge를 대상으로 union-find를 사용하여 connected graph를 만든다. 각 edge에 대해 두 vertex의 root를 find함수로 찾고 다른 경우에만 union을 사용하는 방식이다. 다음으로는 모든 vertex를 검색하면서 connected graph가 몇 개 있는지, 해당 graph의 집합을 나타내는 root는 무엇인지 갱신한다.

```
//connected graph(group)의 component 개수를 계산하고 subset의 root를 계산하여 저장해놓는다
for (int i = 0; i < n_vertex; i++) {
    temp = find(subsets, i);
    for (int j = 0; j < group_cnt; j++) {
        if (temp == group[j].root) { //connected graph(group)들 중 root가 같은 집합을 찾으면
            group[j].comp_cnt++;     //해당 graph의 connected component의 수를 증가시킨다
            subsets[i].root = group[j].root;
            break;
        }
    }
}

int num_edge = n_edge;
int in_edge = 0;
buildMinHeap(Edge_Arr, n_edge);

//subsets을 root배고 초기화 시켜서 다시 find-union을 쓸 수 있는 상태로 만든다
for (int i = 0; i < n_vertex; i++) {
    subsets[i].parent = i;
    subsets[i].rank = 0;
}
```

다음으로는 connected graph의 component 개수를 계산하고 subset의 root를 계산하여 저장해놓는다. 왜냐하면 이후 subsets의 root 빼고 초기화시켜서 다시 find-union을 쓸 수 있는 상태로 만들 것이기 때문이다. 이런 작업이 추가된 이유는 이번 과제에 주어진 graph가 1개의 connected graph라는 보장이 없기 때문이다. Subset이 이미 union-find로 연결되어 있기 때문에 이 subset을 사용해서는 MST를 계산할 수 없다. 왜냐하면 Kruskal algorithm의 원리는 union-find를 통해 떨어진 vertex들을 연결시키면서 weight를 더해가기 때문이다. 이후에는 buildMinheap을 사용하여 minheap을 만들고 subset을 초기화시킨다.

```
//모든 Edge를 검색하면서
while (num_edge > 0) {
    edge next_edge = extractMin(Edge_Arr, &num_edge);
    k_scanned++; //heap의 원소가 하나씩 빠질 때 마다 더해줘서

    //edge의 출발 vertex와 도착 vertex의 집합이 같은지 find를 통해 검사한다
    int x = find(subsets, next_edge.from); //next_edge의 출발 vertex를 포함한 집합의 root를 x로 받아옴
    int y = find(subsets, next_edge.to);   //next_edge의 도착 vertex를 포함한 집합의 root를 y로 받아옴

    //만약 해당 edge(next_edge)가 cycle을 초래하지 않으면 포함한다
    //x != y : edge의 출발 vertex와 도착 vertex가 같은 집합이 아니었다는 뜻 --> edge를 포함해도 cycle이 발생하지 않는다
    //subsets[next_edge.from].root == subsets[next_edge.to].root : 둘이 같은 집합, 즉 같은 connected graph의 원소일 때
    if (x != y && subsets[x].root == subsets[y].root) {
        in_edge++;
        if (in_edge == n_vertex - 1)
            break;
        temp = subsets[x].root; //connected graph의 root를 temp에 저장하고
        for (int j = 0; j < group_cnt; j++) { //connected graph의 목록을 찾아보면서
            if (temp == group[j].root) { // 해당하는 connected graph을 찾으면
                group[j].weight_total += next_edge.weight;   //해당 graph의 total weight를 증가시킨다
                break;
            }
        }
        Union(subsets, x, y);
    }
}
printf("k_scanned: %d   in_edge: %d    n_vertex: %d\n", k_scanned, in_edge, n_vertex);
```

Kruskal MST의 핵심이자 마지막 부분으로 num_edge에 저장된 minheap의 edge 수가 0이 되기 전까지 while loop을 반복한다. 다른 종료 조건으로는 in_edge가 가리키는 vertex의 수가 connected graph의 vertex-1이 되었을 때이다. K_scanned는 heap의 원소가 하나씩 빠질 때마다 계산된 edge가 1개씩 증가되기 때문에 while loop마다 1씩 증가시켜주고 minheap에서 빠진 edge에 대해 vertex의 root를 find로 찾고 둘이 다르다면, 즉 아직 연결되지 않은 그룹이라면 union을 통해 연결한다. 이 때 subsets[x].root == subsets[y].root라는 조건이 하나 더 추가되는데 이는 여러 connected graph가 있다면 두 vertex가 같은 connected graph에 속한 것이 아닐 수 있기 때문에 미리 계산해둔 subsets의 root 정보를 이용한 것이다. 그리고 union하기 전에 여러 connected graph의 정보를 저장한 group 구조체를 검사해서 해당 group의 weight_total에 해당 edge의 weight를 더해주는 과정을 거친다.

find함수를 통해 subset의 tree를 탐색하는 것이 worst case의 경우 logV에 해당하고 해당 부분이 포함된 곳의 while loop이 worst case에 모든 edge를 탐색하기 때문에 시간복잡도는 O(E logV)꼴이 된다. 기존의 수업시간에 다룬 예제 코드에서는 qsort를 통해 edge를 weight를 기준으로 오름차순으로 정렬할 때 O(E logE)의 시간이 걸렸지만 이번 과제에서는 Minheap을 구현하고 제일 작은 원소를 빼는 방식을 사용하고 있다. 이러한 방식도 qsort와 동일하게 O(E logE)의 시간복잡도를 갖는데 K scanned가 edge보다 작은 것을 보며 알 수 있듯, 이러한 방식은 모든 edge를 정렬하지 않고도 검색하다가 vertex의 개수만큼 추출을 하면 더 이상 탐색을 하지 않아도 된다는 장점이 있다.

3. 환경

OS: Windows 10 Education

CPU: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz

RAM: 16.0GB

Compiler: Visual Studio 2022 Release Mode