HW5 보고서 숙제

전공: 컴퓨터공학과 학년: 4학년 학번: 20212021 이름: 원대호

1. 컴퓨터 실험 환경

[1] OS 에디션 : Windows 11 Home

[2] 버전 : 23H2

[3] OS 빌드 : 22631.4169

[4] CPU : 12th Gen Intel(R) Core(TM) i7-1260P

[5] RAM : 16.0GB

[6] Compiler: Visual Studio 2022 Release Mode / x64 Platform

```
C:\Users\sed26\source\repos\Project1\x64\Release\Project1.exe(프로세스 204328개)이(가) 종료되었습니다(코드: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

2. 실험 목표

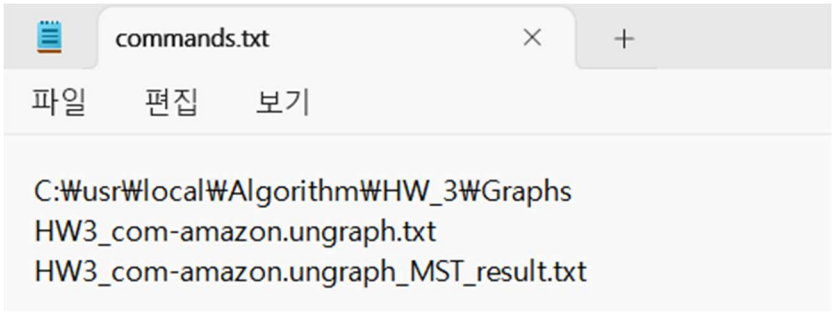
[1] greedy에 대한 이해도를 높인다.

[2] kruscal의 minimum spanning tree algorithm을 구현한다.

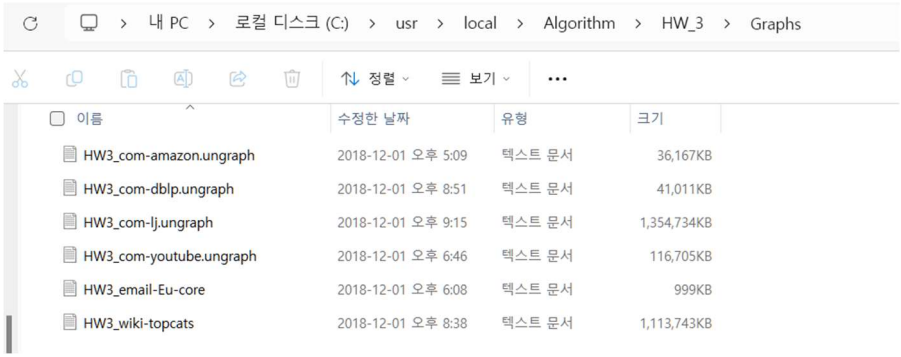
[3] 자신이 구현한 각 방법의 이론적인 시간 복잡도와 실제 수행 시간과의 관계를 분석한다.

3. 실험 세팅

[1] visual studio의 solution directory에 존재하는 commands.txt

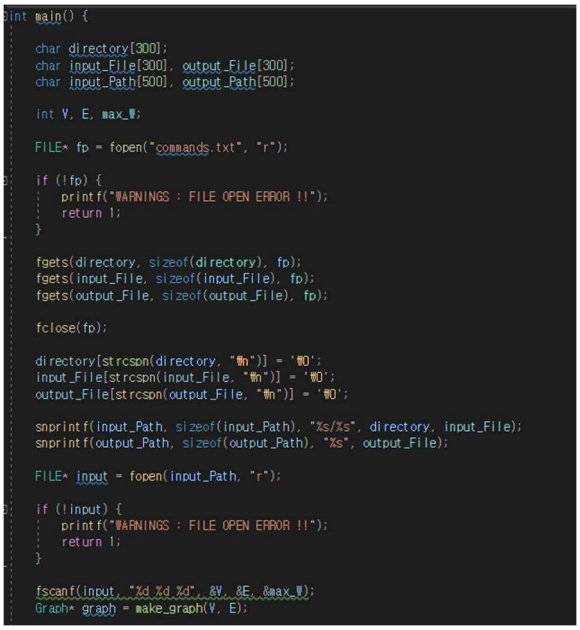


[2] C:₩usr₩local₩Algorithm₩HW_3₩Graphs



4. 코드 구현

[1] main 함수



```
int main() {

    char directory[300];
    char input_File[300], output_File[300];
    char input_Path[500], output_Path[500];

    int V, E, max_W;

    FILE* fp = fopen("commands.txt", "r");

    if (!fp) {
        printf("WARNINGS : FILE OPEN ERROR !!");
        return 1;
    }

    fgets(directory, sizeof(directory), fp);
    fgets(input_File, sizeof(input_File), fp);
    fgets(output_File, sizeof(output_File), fp);

    fclose(fp);

    directory[strcspn(directory, "₩n")] = '₩0';
    input_File[strcspn(input_File, "₩n")] = '₩0';
    output_File[strcspn(output_File, "₩n")] = '₩0';

    snprintf(input_Path, sizeof(input_Path), "%s/%s", directory, input_File);
    snprintf(output_Path, sizeof(output_Path), "%s", output_File);

    FILE* input = fopen(input_Path, "r");

    if (!input) {
        printf("WARNINGS : FILE OPEN ERROR !!");
        return 1;
    }

    fscanf(input, "%d %d %d", &V, &E, &max_W);
    Graph* graph = make_graph(V, E);
```

```
for (int j = 0; j < E; j++) {
    fscanf(input, "%d %d %d", &graph->edges[j].u, &graph->edges[j].v, &graph->edges[j].w);
}

fclose(input);
clock_t s_time, e_time;
double exc_time;
s_time = clock();

Find_MST(output_Path, graph);

e_time = clock();
exc_time = ((double)(e_time - s_time)) / CLOCKS_PER_SEC;
printf("수행시간(초) : %.4f₩n", exc_time);

free(graph->edges);
free(graph);

return 0;
```

Main 함수는 minimum spanning tree를 계산하기 위한 그래프 data를 읽고 결과를 출력하는 역할을 한다. 먼저, commands.txt파일에서 그래프 입력 경로와 출력 파일 경로를 읽고, 해당 graph 파일에서 vertex의 개수, edge의 개수, max_weight를 저장한다. 이후 findMST함수를 호출해 MST를 계산하고, 이를 출력 파일에 저장한다. 또한 실제 수행 시간을 측정해 화면에 출력한다. 모든 작업이 완료된 후, 할당된 메모리를 해제해 메모리 누수를 방지한다.

[2] compare 함수

```
int compare(const void* a, const void* b) {

    const Result* result1 = (const Result*)a;
    const Result* result2 = (const Result*)b;

    if (result1->size != result2->size) {

        return result1->size - result2->size;
    }

    return (result1->weight > result2->weight) - (result1->weight < result2->weight);
}
```

결과 파일을 출력할 때 number of vertices로 오름차순으로 정렬한다. 이후 number of vertices가 동일한 경우 total weight로 정렬한다. 이를 구현하기 위해 compare함수를 구축했다.

[3] make_graph 함수

```
Graph* make_graph(int V, int E) {

    Graph* G = (Graph*)malloc(sizeof(Graph));

    G->V = V;
    G->E = E;

    G->edges = (Edge*)malloc(E * sizeof(Edge));

    return G;
}
```

make_graph 함수는 graph 구조체를 동적으로 생성하고 초기화하는 역할을 한다. 주어진 vertex와 edge의 개수를 설정한 후, edge 정보를 저장할 동적 배열을 메모리에 할당한다.

[3] Union과 Find 함수

```
int Find(Subset* set, int x) {

    if (x != set[x].parent) {
        set[x].parent = Find(set, set[x].parent);
    }

    return set[x].parent;
}

void Union(Subset* set, int a, int b, int64_t w) {

    int root_A = Find(set, a);
    int root_B = Find(set, b);

    if (set[root_A].rank > set[root_B].rank) {
        set[root_B].parent = root_A;
        set[root_A].weight += set[root_B].weight + w;
        set[root_A].size += set[root_B].size;
    }
    else if (set[root_A].rank < set[root_B].rank) {
        set[root_A].parent = root_B;
        set[root_B].weight += set[root_A].weight + w;
        set[root_B].size += set[root_A].size;
    }
    else {
        set[root_B].parent = root_A;
        set[root_A].rank++;
        set[root_A].weight += set[root_B].weight + w;
        set[root_A].size += set[root_B].size;
    }
}
```

이 코드는 Union-Find 알고리즘을 구현한 것으로, 두 node가 같은 집합에 속해 있는지 확인하고, 서로 다른 집합을 합치는 역할을 한다. Find 함수는 path compression을 사용해 특정 node의 root를 찾고, 탐색 route를 최적화한다. Union 함수는 두 node의 root를 비교해 rank가 높은 쪽을 parents로 설정하거나, Rank가 같다면 한쪽의 Rank를 증가시켜 균형을 맞춘다. 또한, 두 집합을 합칠 때 Weight와 Size를 갱신해 관리한다. 이러한 구조는 Kruskal algorithm에서 cycle을 방지하고 MST를 구성하는 데 이용된다.

[4] insert_edge와 delete_edge함수

```
void insert_edge(Min_heap* minheap, Edge edge) {

    minheap->size++;

    int temp = minheap->size - 1;

    while (temp > 0 && edge.w < minheap->heap[(temp - 1) / 2].w) {

        minheap->heap[temp] = minheap->heap[(temp - 1) / 2];
        temp = (temp - 1) / 2;

    }
    minheap->heap[temp] = edge;
}
```

```
Edge delete_edge(Min_heap* minheap) {

    Edge MIN = minheap->heap[0];
    minheap->heap[0] = minheap->heap[minheap->size - 1];

    minheap->size--;
    Heap_Sort(minheap, 0);

    return MIN;
}
```

insert_edge 함수는 새로운 간선을 heap의 마지막 위치에 추가한 뒤, min heap을 유지하기 위해 부모

노드와 가중치를 비교하며 위로 이동시킨다. 반면, delete_edge 함수는 heap의 최상단에 있는 최소 가중치 간선을 제거한 뒤, 마지막 간선을 root로 이동시키고 Heap_Sort를 호출해 heap을 재정렬한다.

[5] make_MINHEAP과 Heap_sort함수

```
Min_heap* make_MINHEAP(int max) {

    Min_heap* min_heap = (Min_heap*)malloc(sizeof(Min_heap));
    min_heap->maxSize = max;
    min_heap->size = 0;
    min_heap->heap = (Edge*)malloc(max * sizeof(Edge));

    return min_heap;
}


void Heap_Sort(Min_heap* minheap, int index) {

    int R = 2 * index + 2;
    int L = 2 * index + 1;
    int SMALL = index;

    if (L < minheap->size && minheap->heap[L].w < minheap->heap[SMALL].w) {
        SMALL = L;
    }

    if (R < minheap->size && minheap->heap[R].w < minheap->heap[SMALL].w) {
        SMALL = R;
    }

    if (SMALL != index) {
        Edge temp = minheap->heap[index];
        minheap->heap[index] = minheap->heap[SMALL];
        minheap->heap[SMALL] = temp;
        Heap_Sort(minheap, SMALL);
    }
}
```

이 코드는 Min Heap을 생성 및 유지하는 두 가지 주요 기능을 구현한다. make_MINHEAP 함수는 최대 크기를 인자로 받아 heap 구조체를 동적으로 생성하고, 간선을 저장할 배열을 초기화한다. Heap_Sort 함수는 주어진 index를 기준으로 자식 node와 비교하며 min-heap property를 유지하도록 재귀적으로 정렬한다. 왼쪽 자식과 오른쪽 자식 중 더 작은 가중치를 가진 node와 현재 node를 교환한 후, 재귀적으로 Heap_Sort를 호출한다. 이를 통해 삽입, 삭제 연산 후에도 heap의 정렬 상태가 유지된다.

[6] Find_MST함수

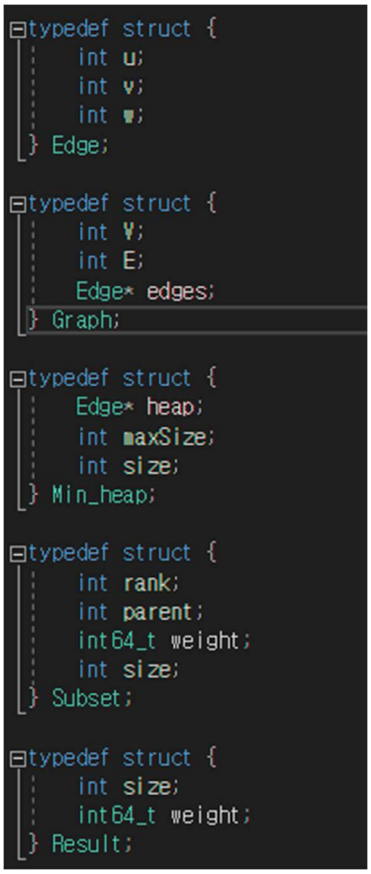
```
void Find_MST(const char* output_Path, Graph* graph) {

    int V = graph->V;
    int k_scanned = 0;

    Min_heap* minheap = make_MINHEAP(graph->E);

    for (int x = 0; x < graph->E; x++) {

        insert_edge(minheap, graph->edges[x]);
    }

    Subset* set = (Subset*)malloc(V * sizeof(Subset));
    for (int i = 0; i < V; i++) {
        set[i].parent = i;
        set[i].rank = 0;
        set[i].weight = 0;
        set[i].size = 1;
    }

    while (minheap->size > 0) {
        Edge nextEdge = delete_edge(minheap);
        k_scanned += 1;

        int a = Find(set, nextEdge.u);
        int b = Find(set, nextEdge.v);

        if (a != b) {
            Union(set, a, b, nextEdge.w);
        }
    }


    Result* R = (Result*)malloc(V * sizeof(Result));

    int cnt = 0;
```

```
for (int x = 0; x < V; x++) {
    if (set[x].parent == x) {
        R[cnt++] = (Result){ set[x].size, set[x].weight };
    }
}
printf("kscanned : %d\n", k_scanned);
// 결과 파일을 출력할 때 number of vertices로 오름차순으로 정렬한 후,
// 같은 것들은 total weight로 정렬하기 위한 함수이다.
qsort(R, cnt, sizeof(Result), compare);

FILE* output = fopen(output_Path, "w");
if (output) {

    fprintf(output, "%d\n", cnt);

    for (int i = 0; i < cnt; i++) {
        fprintf(output, "%d %lld\n", R[i].size, R[i].weight);
    }
    fclose(output);
}

free(R);
free(set);
free(minheap->heap);
free(minheap);
```

이 함수는 Kruskal algorithm을 주어진 graph의 Minimum Spanning Tree를 계산하고 결과를 file에 저장한다. 먼저, 입력된 edge들을 저장하기 위해 Min Heap을 생성하고 모든 간선을 삽입한다. 각 정점은 Union-Find 구조로 초기화되어, Path Compression과 Union by Rank로 효율적으로 관리된다. 이후 while loop를 통해 Min Heap에서 weight가 가장 작은 edge을 하나씩 꺼내고, 두 정점이 다른 집합에 속해 있으면 Union 연산을 수행해 MST를 구성한다. 이 process에서 k_scanned가 기록된다. MST가 완성되면 각 연결 component의 size와 weight를 Result 배열에 저장한다. 이 배열은 정점 수와 가중치를

기준으로 정렬돼 출력 file에 기록된다. 수행된 간선 스캔 수와 결과는 출력 파일에 저장되며, 마지막으로 동적 memory를 해제해 memory 누수를 방지한다.

[7] typedef struct



Edge 구조체는 두 정점과 해당 간선의 weight를 저장한다. Graph 구조체는 정점과 간선의 개수 및 간선 list을 관리한다. Min_heap 구조체는 간선을 저장하는 최소 힙을 표현하며, 힙 배열, 최대 size, 현재 size를 포함한다. Subset 구조체는 Union-Find 연산을 위해 각 정점의 rank, parent, weight, size를 저장한다. 마지막으로 Result 구조체는 각 연결 component의 정점 수와 총 가중치를 저장해 최종 결과를 나타낸다.

5. 실험

[a] 시간 복잡도에 따른 코드 구현

```
void Find_MST(const char* output_Path, Graph* graph) {

    int V = graph->V;
    int k_scanned = 0;

    Min_heap* minheap = make_MINHEAP(graph->E);

    for (int x = 0; x < graph->E; x++) {

        insert_edge(minheap, graph->edges[x]);
    }

    Subset* set = (Subset*)malloc(V * sizeof(Subset));
    for (int i = 0; i < V; i++) {
        set[i].parent = i;
        set[i].rank = 0;
        set[i].weight = 0;
        set[i].size = 1;
    }

    while (minheap->size > 0) {
        Edge nextEdge = delete_edge(minheap);
        k_scanned += 1;

        int a = Find(set, nextEdge.u);
        int b = Find(set, nextEdge.v);

        if (a != b) {
            Union(set, a, b, nextEdge.w);
        }
    }


    Result* R = (Result*)malloc(V * sizeof(Result));

    int cnt = 0;
```

```
for (int x = 0; x < V; x++) {
    if (set[x].parent == x) {
        R[cnt++] = (Result){ set[x].size, set[x].weight };
    }
}
printf("kscanned : %d₩n", k_scanned);
// 결과 파일을 출력할 때 number of vertices로 오름차순으로 정렬한 후,
// 같은 것들은 total weight로 정렬하기 위한 함수이다.
qsort(R, cnt, sizeof(Result), compare);

FILE* output = fopen(output_Path, "w");
if (output) {

    fprintf(output, "%d₩n", cnt);

    for (int i = 0; i < cnt; i++) {
        fprintf(output, "%d %lld₩n", R[i].size, R[i].weight);
    }
    fclose(output);
}

free(R);
free(set);
free(minheap->heap);
free(minheap);
```

Find_MST는 kruscal algorithm을 기반으로 mst를 구현한 함수이다. 이 함수는 주어진 그래프의 간선을 min heap에 삽입해 sorting된 상태로 유지하며, find와 union 연산을 통해 mst를 구성한다. 먼저, 모든 간선은 min heap에 삽입하는 insert_edge는 O(|E|log|E|)의 시간 복잡도를 가진다. 그러나 최악의 경우 간선의 수 |E|는 |V|^2에 비례하기 때문에 근사적으로 O(|E|log|V|)로 표현된다. 이후 while loop에서는 min heap에서 최소 weight 간선을 하나씩 꺼내는 delete_edge 연산이 수행되며, 이 연산은 O(log|E|)의 time complexity를 가진다. 그러나 |E|는 |V|^2에 비례하므로 O(log|V|)로 근사화된다. 각 간선에 대해 find와 union연산이 수행되며, 이는 path compression 및 union by rank 최적화로 인해 O(log|V|)의 time complexity를 가진다. 전체적으로 while loop는 O(|E|log|V|)의 time complexity를 가지며, 이는 MST를 구성하는 핵심연산이다. 한편, subset 초기화와 결과 배열을 처리하는 반복문은 가각 O(|v|)의 시간 복잡도를 가지지만, 이는 전체 time complexity에 큰 영향을 미치지 않는다.

[b] 실험결과

```
fclose(input);
clock_t s_time, e_time;
double exc_time;
s_time = clock();

Find_MST(output_Path, graph);

e_time = clock();
exc_time = ((double)(e_time - s_time)) / CLOCKS_PER_SEC;
printf("수행시간(초) : %.4f₩n", exc_time);
```

[출력 예시]

```
kscanned : 2987624
수행시간(초) : 1.5330

C:\Users\sed26\source\repos\Project1\x64\Release\Project1.exe(프로세스 18350
8개)이(가) 종료되었습니다(코드: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

[1] 첫번째 시행

파일 이름	작동 여부	MST weight	수행 시간(초)	Kscanned
HW3_com-amazon.ungraph	YES	2729670156	0.3190	925855
HW3_com-dblp.ungraph	YES	2747895457	0.3320	1049834
HW3_com-lj.ungraph	YES	28308045762	14.8840	34681165
HW3_com-youtube.ungraph	YES	14578691475	1.0120	2987623

[2] 두번째 시행

파일 이름	작동 여부	MST weight	수행 시간(초)	Kscanned
HW3_com-amazon.ungraph	YES	2729670156	0.3700	925855
HW3_com-dblp.ungraph	YES	2747895457	0.3540	1049834
HW3_com-lj.ungraph	YES	28308045762	17.9020	34681165
HW3_com-youtube.ungraph	YES	14578691475	0.9990	2987623

[3] 세번째 시행

파일 이름	작동 여부	MST weight	수행 시간(초)	Kscanned
HW3_com-amazon.ungraph	YES	2729670156	0.2750	925855
HW3_com-dblp.ungraph	YES	2747895457	0.3030	1049834
HW3_com-lj.ungraph	YES	28308045762	15.0530	34681165
HW3_com-youtube.ungraph	YES	14578691475	1.0070	2987623

[최종 결과]

파일 이름	작동 여부	MST weight	수행 시간(초)	Kscanned
HW3_com-amazon.ungraph	YES	2,729,670,156	0.321	925855
HW3_com-dblp.ungraph	YES	2,747,895,457	0.329	1049834
HW3_com-lj.ungraph	YES	28,308,045,762	15.946	34681165
HW3_com-youtube.ungraph	YES	14,578,691,475	1.006	2987623

[c] 결과 분석

[1] MST weight 오름차순 결과

파일 이름	작동 여부	MST weight	수행 시간(초)	Kscanned
HW3_com-amazon.ungraph	YES	2,729,670,156	0.321	925,855
HW3_com-dblp.ungraph	YES	2,747,895,457	0.329	1,049,834
HW3_com-youtube.ungraph	YES	14,578,691,475	1.006	2,987,623
HW3_com-lj.ungraph	YES	28,308,045,762	15.946	34,681,165

[2] |E|log|V| 결과

파일 이름	Kscanned	Vertex	|E|log|V|	수행 시간
HW3_com-amazon.ungraph	925,855	334,863	5,115,225	0.321
HW3_com-dblp.ungraph	1,049,834	317,080	5,775,314	0.329
HW3_com-youtube.ungraph	2,987,623	1,134,890	18,089,919	1.006
HW3_com-lj.ungraph	34,681,165	3,997,962	228,959,455	15.946

[3] |E|log|V| / 수행시간 결과

파일 이름	|E|log|V| / 수행 시간	수행 시간
HW3_com-amazon.ungraph	15,935,280	0.321
HW3_com-dblp.ungraph	17,554,145	0.329
HW3_com-youtube.ungraph	17,982,026	1.006
HW3_com-lj.ungraph	14,358,425	15.946

Time complextiy에 대한 실험적 결과를 검증하기 위해 수행 시간과 |V| , |E| , |E|log|V| 값을 비교해 분석했다. 이론적으로 |E|log|V| 값이 커질수록 수행 시간은 비례적으로 증가한다. 실험 결과, Amazon과 dblp ungraph에서는 이론적 time complexity와 실제 수행 시간이 비례 관계를 잘 유지했다. Youtube ungraph는 약간의 비례 관계 약화가 있었지만, 이는 graph의 구조적 특성으로 인한 결과로 해석할 수 있다. 그러나 lj ungraph에서는 비례 관계가 명확하다고 볼 수는 없다. 이는 grpah의 구조적 불균형, memory 접근 비효율성, 또는 환경적 요인 때문일 가능성이 높다. 그럼에도 불구하고 전반적으로 kruscal algroithm은 O(|E|log|V|)의 time complexity를 만족하며, 이론적 분석이 대부분의 실험 result로 검증됐음을 확인할 수 있었다. 추가적으로 algorithm의 최적화, memory 접근 방식과 같은 요소들을 세부적으로 분석함으로써 정확성을 더욱 높일 수 있을 것이다.