알고리즘 설계와 분석 HW3 결과보고서

2020.12.26 (토) 8PM / 20181202 김수미

1. 노트북 실험 환경

OS : Windows 10 Home (맥북 Parallels)

CPU : Intel(R) Core(TM) i5-7Y54 CPU @ 1.20GHz

RAM : 4.00GB

Compiler : Microsoft Visual Studio Community 2017 15.9.9 / X64 Release

2. 실험 결과

파일 이름	작동여부	MST Weight	수행 시간(초)	K scanned
HW3_com-dblp.ungraph.txt	YES	2,747,895,457	10.2(4.0)	1,049,834
HW3_com-amazon.ungraph.txt	YES	2,729,670,156	8.9(3.4)	925,855
HW3_com-youtube.ungraph.txt	YES	14,578,691,475	29.1(11.1)	2,987,623
HW3_com-lj.ungraph.txt	YES	28,308,045,762	1036.5(365.0)	34,681,165

수행 시간은 '전체 수행 시간(Min heap 구축 시간)'으로 나타낸 것이다. 전체 수행 시간은 $O(|E|\log|V|)$의 시간복잡도를 따르며, Min heap 구축 시간은 $O(|E|)$의 시간복잡도를 따른다.

위 표에서 수행 시간만을 남기고 그래프 관련 정보를 추가하면 아래와 같다.

파일 이름	Vertex 개수	Edge 개수($|E|$)	수행 시간(초)	$\|E\|\log_2\|V\|$
HW3_com-dblp.ungraph.txt	317,080	1,049,866	10.2(4.0)	18,897,588
HW3_com-amazon.ungraph.txt	334,863	925,872	8.9(3.4)	16,665,696
HW3_com-youtube.ungraph.txt	1,134,890	2,987,624	29.1(11.1)	59,752,480
HW3_com-lj.ungraph.txt	3,997,962	34,681,189	1036.5(365.0)	762,986,158

먼저 괄호 안의 Min heap 구축 시간을 보았을 때, 위에서부터 차례대로 4, 3.4, 11.1, 365의 소요시간을 가지는데, 맨 처음 4초를 기준으로 각각 1배, 0.85배, 2.78배, 91.25배 로 나타낼 수 있다. 또한 $|E|$의 값을 살펴보면 위에서부터 1049866, 925872, 2987624, 34681189 임을 확인할 수 있다. 이 역시 맨 처음 1049866를 기준으로 각각 1배, 0.88배, 2.84배, 33배 로 나타낼 수 있는데, 마지막 값을 제외하면(데이터 크기가 커지면서 실험 환경에 따라 소요 시간이 정비례 하지 않는 듯 하다) 모두 Min heap의 구축 시간이 $O(|E|)$의 시간복잡도에 비례한다는 것을 보여준다.

그 다음 전체 수행 시간의 경우, 위에서부터 차례대로 10.2, 8.9, 29.1, 1036.5의 소요시간을 가지는데, 맨 처음 10.2초를 기준으로 각각 1배, 0.87배, 2.85배, 101.61배 로 나타낼 수 있다. 또한

$|E|\log_2|V|$의 값을 살펴보면 위에서부터 18897588, 16665696, 59752480, 762986158 의 값을 가지는데, 이 역시 맨 처음 18897588을 기준으로 1배, 0.88배, 3.16배, 40.37배 로 나타낼 수 있다. 마지막 값을 제외하면(데이터 크기가 커지면서 실험 환경에 따라 소요 시간이 정비례 하지 않는 듯 하다) 모두 전체 수행 시간이 $O(|E|\log|V|)$의 시간복잡도에 유사하게 비례한다는 것을 보여준다.

3. 코드 구현 설명 : Graph Representation & Disjoint Set

해당 코드는 주어진 그래프 정보를 가지고, Kruskal 알고리즘을 사용하여 그래프의 Minimum Spanning Tree의 총 weight 값을 구하는 것을 목표로 한다. 이 때 Disjoint-sets 자료구조와 Min-heap 자료구조를 사용하여 효율을 높일 것이다. 아래는 코드 구현에 대한 세부적인 설명이다.

```
1   #include <stdlib.h>
2   #include <stdio.h>
3   #include <time.h>
4    char input_file[128]; char output_file[128];
5    int n_vertices, n_edges, MAX_WEIGHT;
6    long long total_weight = 0, count = 0;
7    int v_count = 0; int cntCmp_arr[1000];
8    clock_t start1, start2, end1, end2;
```

먼저 코드 작성에 필요한 헤더파일들을 가져오고, 전역 변수들을 선언해준다. 수행 시간 측정을 위해 〈time.h〉 헤더파일을 사용해 주었으며, total_weight 변수는 MST의 총 weight 값의 합을 저장해야 하므로 가장 큰 범위값을 가지는 long long 타입으로 선언해 주었다.

```
10    // min HEAP 구성요소 구조체
11  struct Element {
12      int key;
13      int index;
14  };
```

Min heap의 구성요소가 되는 구조체를 선언해 주었다. Key 필드에는 그래프의 edge들의 weight 값이 저장되며, index에는 heap의 구성요소 고유 번호가 저장된다. 해당 고유번호는 edge 정보에 접근하는 데에 사용된다.

```
16    // min HEAP insert 함수
17  void insert_min_heap(struct Element *element, int new_element, int *n, int index) {
18      int i = ++(*n);
19      while ((i != 1) && (new_element) < element[i / 2].key) {
20          element[i].key = element[i / 2].key;
21          element[i].index = element[i / 2].index;
22          i /= 2; }
23      element[i].key = new_element;
24      element[i].index = index;
25  }
```

위 함수는 Min heap에 값을 집어 넣어 Min heap을 구축하는 함수이다. 입력 받는 txt 파일 내부에 저장되어있는 모든 edge를 한번씩 저장하며($O(|E|)$), 새롭게 입력 받은 edge가 Min heap 내부에서

본인의 자리를 찾아가는데 걸리는 시간은 O(logn) 이므로, 총 소요되는 시간은 이론적으로 $O(|E| + \log n) = O(|E|)$ 이다.

```
27      // min HEAP delete 함수
28      int delete_min_heap(struct Element *element, int *n) {
29          int parent, child;
30          struct Element item, temp;
31          item = element[1];
32          temp = element[(*n)--];
33          parent = 1; child = 2;
34          while (child <= *n) {
35              if ((child < *n) && (element[child].key > element[child + 1].key)) child++;
36              if (temp.key <= element[child].key) break;
37              element[parent] = element[child];
38              parent = child;
39              child *= 2;
40          }
41          element[parent] = temp;
42          return item.index;
43      }
```

위 함수는 Min heap에서 최솟값을 하나씩 pop(delete)하는 함수이다. 매번 힙이 가지고 있는 값 중에서 가장 작은 값을 반환하며, 값을 반환한 다음에는 내부 힙 구조를 다시 Min heap 형태로 맞춰주는 adjust 과정을 거친다.

```
45      // weighted edge를 나타내기 위한 구조체
46      struct Edge {
47          int src, dest;
48          int weight;
49      };
50
51      // 각 Connected Components들을 하나의 그래프로 본다
52      struct Graph {
53          int V, E;
54          struct Edge* edge;
55      };
```

위 두 구조체는 그래프 표현을 위해 생성한 것들이다. Edge 구조체는 그래프의 weighted edge를 나타내기 위한 것이며, edge를 구성하는 두개의 vertex번호와 weight 값을 필드로 가진다.

Graph 구조체는 입력 파일이 나타내는 전체 그래프 정보를 저장하는 구조체이다. 총 vertex 개수와 총 edge 개수를 저장하는 V, E 필드 그리고 모든 edge들의 정보를 저장하는 Edge* edge 배열 필드로 이루어져 있다. 배열 필드는 총 edge 개수만큼의 공간을 가지게 되며, 모든 edge들의 구성 vertex 및 weight 정보가 저장될 것이다.

```
57      // 그래프를 생성하는 함수
58      struct Graph* createGraph(int V, int E) {
59          struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
60          graph->V = V;
61          graph->E = E;
62          graph->edge = (struct Edge*)malloc(graph->E * sizeof(struct Edge));
63          return graph;
64      }
```

그래프를 생성하는 함수이다. 입력 텍스트 파일로부터 그래프 정보를 받아 저장한 다음 Graph* graph 포인터를 반환한다. 이처럼 그래프는 모든 edge정보를 배열로 저장하는 형태로 구현했다. 이 형태는 입력 받는 텍스트 파일의 양식을 고려했을 때, adjacency list 혹은 adjacency matrix로 표현하는 것 보다 효율적으로 그래프를 관리할 수 있게 해준다.

```
66      // Union-Find의 subset 표현을 위한 구조체
67      struct subset {
68          int parent;
69          int rank;
70      };
71
72      // Find 연산을 수행하는 함수
73      int find(struct subset subsets[], int i) {
74          if (subsets[i].parent != i)
75              subsets[i].parent = find(subsets, subsets[i].parent);
76          return subsets[i].parent;
77      }
78
79      // Union 연산을 수행하는 함수
80      void Union(struct subset subsets[], int x, int y) {
81          int xroot = find(subsets, x);
82          int yroot = find(subsets, y);
83          if (subsets[xroot].rank < subsets[yroot].rank)
84              subsets[xroot].parent = yroot;
85          else if (subsets[xroot].rank > subsets[yroot].rank)
86              subsets[yroot].parent = xroot;
87          else { subsets[yroot].parent = xroot;
88              subsets[xroot].rank++; }
89      }
```

위는 Disjoint-set 자료구조의 활용에 필요한 요소들이다. 먼저 subset 구조체는 Disjoint-set의 원소가 되는 요소이다. parent와 rank 필드로 구성되어 있으며, parent는 실질적으로 특정 원소의 root 원소 번호를 가리킨다.Union/Find 연산에 해당 필드 값들이 필요하다.

find 함수는 Disjoint-set의 find 연산을 수행하는 함수이다. 매개변수로 입력 받은 원소의 root를 찾아주며, union 연산을 어떻게 수행할 것인지를 결정하는 과정의 일부가 된다.

Union 함수는 Disjoint-set의 Union 연산을 수행하는 함수이다. 하나의 edge를 구성하는 두 vertex 값을 매개변수로 받으며, 두 vertex값을 원소로 하는 Disjoint set의 root 원소가 다를 경우, Union 연산을 진행한다. 연산은 두 집합의 rank 값에 따라 어느 집합이 상위에 놓일지 결정된다.

```
91      // Disjoint Set을 만드는 함수
92      int dsjSet(struct Graph* graph, struct Element *element, int *n) {
93          int V = graph->V, E = graph->E;
94          int e = 0;
95
96          struct subset *subsets = (struct subset*)malloc(V * sizeof(struct subset));
97          for (int v = 0; v < V; ++v) {
98              subsets[v].parent = v;
99              subsets[v].rank = 0; }
100
101         while (e < E) {
102             struct Edge next_edge = graph->edge[e++];
103             int x = find(subsets, next_edge.src);
104             int y = find(subsets, next_edge.dest);
105             if (x != y) Union(subsets, x, y); }
106
107         int cntCmp = 0, flag = 0;
108         int root[1000] = { -1 }; // 각 vertex의 root vertex를 저장
109
110         for (int v = 0; v < V; v++) {
111             flag = 0;
112             for (int i = 0; i < cntCmp + 1; i++)
113                 if (root[i] == find(subsets, v)) flag = 1;
114             if (flag == 0) { root[cntCmp] = find(subsets, v); cntCmp++; }
115         }
116
117         return cntCmp;
118     }
```

입력 받은 텍스트 파일이 나타내는 그래프를 Disjoint-set으로 나타내는 함수이다. 해당 함수는 그래프의 Connected Components의 개수를 세는데 이용된다. 그래프를 adjacency list 혹은 adjacency matrix로 나타낸 다음 DFS를 이용하여 Connected Components의 개수를 세는 것 보다 훨씬 빠르게 연산을 수행할 수 있다. 그래프의 모든 edge 정보를 입력 받아 Disjoint set을 완성한 다음, 모든 노드의 root 노드 값을 체크하여 서로 다른 root 노드가 몇개 존재하는지를 바탕으로 전체 그래프에 총 몇개의 Connected Components가 존재하는지 조사한다.

```
120     // MST를 구하는 함수1 (Connected Component가 1개)
121     void KruskalMST1(struct Graph* graph, struct Element *element, int *n) {
122         int V = graph->V; int e = 0;
123         struct Edge *result;
124         result = (struct Edge*)malloc((V - 1) * sizeof(struct Edge));
125 
126         struct subset *subsets = (struct subset*)malloc(V * sizeof(struct subset));
127         for (int v = 0; v < V; ++v) {
128             subsets[v].parent = v;
129             subsets[v].rank = 0; }
130 
131         while (e < V - 1) {
132             int i = delete_min_heap(element, n); count++;
133             struct Edge next_edge = graph->edge[i];
134             int x = find(subsets, next_edge.src);
135             int y = find(subsets, next_edge.dest);
136             if (x != y) { // 서로 다른 tree에 속하는 경우 Union
137                 result[e++] = next_edge;
138                 Union(subsets, x, y); }
139         }
140 
141         for (int i = 0; i < e; ++i)
142             total_weight += result[i].weight;
143     }
```

Kruskal 알고리즘을 사용하여 그래프의 Minimum Spanning Tree를 만드는 핵심 함수이다. 위 함수는 KruskalMST1 함수로, 그래프에 존재하는 Connected Component가 1개인 경우 사용된다. 그래프에 존재하는 Connected Component가 2개 이상인 경우에는 그래프에 존재하는 Connected Comoponents들을 모두 분류하고, 각각의 Components들에 대하여 MST연산을 진행해야 하기 때문에 추가적인 과정이 필요하여 두가지 경우를 분리해 주었다.

Connected Component가 1개인 경우, 입력 파일로부터 edge 정보를 Min heap에 넣은 다음, 가장 작은 weight를 가지는 edge 정보를 하나씩 pop 하면서 MST를 만들어 나갔다. pop된 edge가 가지는 vertex를 가지고 Disjoint-set 자료구조를 이용하여 현재 상태에서 해당 edge를 선택할 수 있는지 여부를 조사하며, 해당 edge를 선택하면 tree에 cycle이 생기는 경우(동일한 Set에 포함되는 경우) 그 edge를 무시하고 다음으로 작은 edge를 pop 하여 동일한 과정을 반복한다. 총 vertex 개수 – 1 개의 edge가 선택되면 MST가 완성이 되었다는 의미이므로, 반복을 중단하고 MST 정보를 출력한다. 출력되는 정보는 오로지 MST를 구성하는 edge들의 weight 값의 총합이다.

```
145     // MST를 구하는 함수2-1 (Connected Component가 2개 이상)
146     int* KruskalMST2(struct Graph* graph, struct Element *element, int *n) {
147         int V = graph->V, E = graph->E;
148         int *vertex = malloc(sizeof(int) * V);
149         int e = 0;
150 
151         struct Edge *result;
152         result = (struct Edge*)malloc((V - 1) * sizeof(struct Edge));
153 
154         struct subset *subsets = (struct subset*)malloc(V * sizeof(struct subset));
155         for (int v = 0; v < V; ++v) {
156             subsets[v].parent = v;
157             subsets[v].rank = 0; }
158 
159         while (e < E) {
160             struct Edge next_edge = graph->edge[e++];
161             int x = find(subsets, next_edge.src);
162             int y = find(subsets, next_edge.dest);
163             if (x != y) Union(subsets, x, y); }
164 
165         int cntCmp = 0, flag = 0;
166         int root[1000] = { -1 }; // 각 vertex의 root vertex를 저장
```

위 함수는 KruskalMST2 함수로, 그래프에 존재하는 Connected Component가 2개 이상인 경우 사용된다. 정확히 말하면 Connected Component가 2개 이상일 때 모든 Components의 MST를 구하는 과정 중 일부에 해당하는 함수이며, 뒤에 나올 KruskalMST3 함수와 함께 사용하여 그래프에 존재하는 모든 Components의 MST를 계산할 수 있다.

먼저 KruskalMST2 함수는 모든 Components의 MST를 계산하기 위해, 각 Components에 포함되어 있는 vertex가 무엇인지를 저장하는 배열을 만든다. 그래프의 모든 edge를 받아 Disjoint-set을 구성한 다음, 모든 vertex를 돌면서 해당 vertex가 어느 Component에 속하는지, 다시 말해 Disjoint-sets 중에서 어떤 root의 Set에 포함되는지를 체크하고, 순서대로 vertex 배열에 값을 저장한다.

vertex 배열에는 각 Component 순서대로 구성 vertex 번호가 저장되는데, 예를 들어 그래프를 구성하는 Connected Component가 2개이고, 각 Component의 구성 요소 개수가 3개, 4개일 때, vertex 배열의 처음 3칸에는 3개짜리 Component의 구성 vertex 번호가, 그 다음 4칸에는 4개짜리 Component의 구성 vertex 번호가 저장된다.

```
168     for (int v = 0; v < V; v++) {
169         flag = 0;
170         for (int i = 0; i < cntCmp + 1; i++)
171             if (root[i] == find(subsets, v)) flag = 1;
172         if (flag == 0) { root[cntCmp] = find(subsets, v); cntCmp++; }
173     }
174
175     e = 0; int vv_count = 0;
176     for (int i = 0; i < cntCmp; i++) {
177         for (int v = 0; v < V; v++) {
178             if (find(subsets, v) == root[i]) {
179                 vertex[vv_count] = v;
180                 v_count++; vv_count++; }
181         }
182         cntCmp_arr[i] = v_count;
183         v_count = 0;
184     }
185
186     return vertex;
187 }
```

위 코드 그림에 이어지는 KruskalMST2 함수의 나머지 부분이다.

```
189 // MST를 구하는 함수2-2 (Connected Component가 2개 이상)
190 void KruskalMST3(struct Graph* graph, struct Element *element, int *n, int *inside, int number) {
191     int V = graph->V; int e = 0;
192     struct Edge *result;
193     result = (struct Edge*)malloc((V - 1) * sizeof(struct Edge));
194
195     struct subset *subsets_2 = (struct subset*)malloc(V * sizeof(struct subset));
196     for (int v = 0; v < V; ++v) {
197         subsets_2[v].parent = v;
198         subsets_2[v].rank = 0; }
199
200     while (e < number - 1) {
201         int i = delete_min_heap(element, n); count++;
202         struct Edge next_edge = graph->edge[i];
203         for (int i = 0; i < number; i++) {
204             if (inside[i] == next_edge.src) {
205                 int x = find(subsets_2, next_edge.src);
206                 int y = find(subsets_2, next_edge.dest);
207                 if (x != y) { // 서로 다른 tree에 속하는 경우 Union
208                     result[e++] = next_edge;
209                     Union(subsets_2, x, y); }
210                 break; }
211         }
212     }
213
214     for (int i = 0; i < e; ++i)
215         total_weight += result[i].weight;
216 }
```

위 함수는 KruskalMST3 함수로, KruskalMST2 함수를 사용한 다음에 호출되어 최종적으로 각 Components의 MST를 계산하는 역할을 한다. 작동 원리는 KruskalMST2 함수에서 반환하는 각 Components의 vertex 번호를 전달받아, 다시 입력 텍스트 파일을 읽으면서 해당 vertex를 포함하는 edge들만 읽어들여 새로운 graph를 구성한다. 그렇게 하나의 Component에 해당하는 edge들

로만 구성된 Connected Graph를 KruskalMST3 함수에 넣어 그 그래프의 MST를 계산하는 것이다. 즉, KruskalMST3 함수는 그래프의 Connected Components 총 개수만큼 호출된다.

```
218  int main() {
219      // commands.txt 파일 오픈
220      FILE *fp; FILE *out;
221      fp = fopen("commands.txt", "r");
222      if (fp == NULL) {
223          fprintf(stderr, "file open error ₩n");
224          exit(1); }
225      fscanf(fp, "%s", &input_file);
226      fscanf(fp, "%s", &output_file);
227      fclose(fp);
228
229      // input 파일 오픈
230      fp = fopen(input_file, "r");
231      if (fp == NULL) {
232          fprintf(stderr, "file open error ₩n");
233          exit(1); }
234      fscanf(fp, "%d", &n_vertices);
235      fscanf(fp, "%d", &n_edges);
236      fscanf(fp, "%d", &MAX_WEIGHT);
237
238      // ouput 파일 오픈
239      out = fopen(output_file, "wt");
240      if (out == NULL) {
241          fprintf(stderr, "file open error ₩n");
242          exit(1); }
```

위에서 작성한 함수들을 호출해 본격적으로 연산을 수행하는 메인 함수이다. 먼저 input file을 열어 그래프 정보를 전역변수들에 저장한다. commands.txt 파일로부터 입력파일(input file)과 출력파일(output file)의 이름을 입력 받을 수 있다. 그 다음 연산 결과를 출력할 output file을 미리 열어둔다.

```
244      // Create Graph according to file
245      int x, y, z, n = 0; start1 = clock();
246      struct Graph* graph = createGraph(n_vertices, n_edges);
247      struct Element* element = malloc(sizeof(struct Element) * n_edges);
248      printf("계산중 . . .₩n");
249      for (int i = 0; i < n_edges; i++) {
250          fscanf(fp, "%d%d%d", &x, &y, &z);
251          graph->edge[i].src = x;
252          graph->edge[i].dest = y;
253          graph->edge[i].weight = z;
254          insert_min_heap(element, z, &(n), i);
255      } fclose(fp);
256
257      // Make Disjoint Sets - count Connected Components 개수
258      int cntCmp = dsjSet(graph, element, &(n));
259      printf("cntCmp : %d 개₩n₩n", cntCmp);
260      fprintf(out, "%d₩n", cntCmp);
```

먼저 그래프의 Connected Components의 개수를 구하기 위해 dsjSet 함수를 호출할 것이다. 이를

위해 input file로부터 edge 정보를 입력 받아 그래프와 Min heap을 하나 만들어주었다. 함수를 호출한 다음에는 연산 결과를 출력해준다.

```
262     // Make MST1 (Connected Component가 1개)
263     if (cntCmp == 1) {
264         graph = createGraph(n_vertices, n_edges); n = 0;
265         element = malloc(sizeof(struct Element) * n_edges);
266 
267         fp = fopen(input_file, "r");
268         fscanf(fp, "%d%d%d", &x, &y, &z);
269 
270         printf("계산중 . . .₩n");
271         start2 = clock();
272         for (int i = 0; i < n_edges; i++) {
273             fscanf(fp, "%d%d%d", &x, &y, &z);
274             graph->edge[i].src = x;
275             graph->edge[i].dest = y;
276             graph->edge[i].weight = z;
277             insert_min_heap(element, z, &(n), i);
278         } fclose(fp); end2 = clock();
279 
280 
281         KruskalMST1(graph, element, &(n));
282         printf("vertex_num : %d₩n", n_vertices);
283         printf("total_weight : %lld₩n", total_weight);
284         printf("K scanned : %lld₩n", count);
285         fprintf(out, "%d %lld", n_vertices, total_weight);
286     }
```

dsjSet 함수를 통해 입력 받은 그래프의 Connected Component 개수를 알게 되면 KruskalMST1을 사용할 것인지, KruskalMST2와 KruskalMST3을 사용할 것인지 결정할 수 있게 된다.

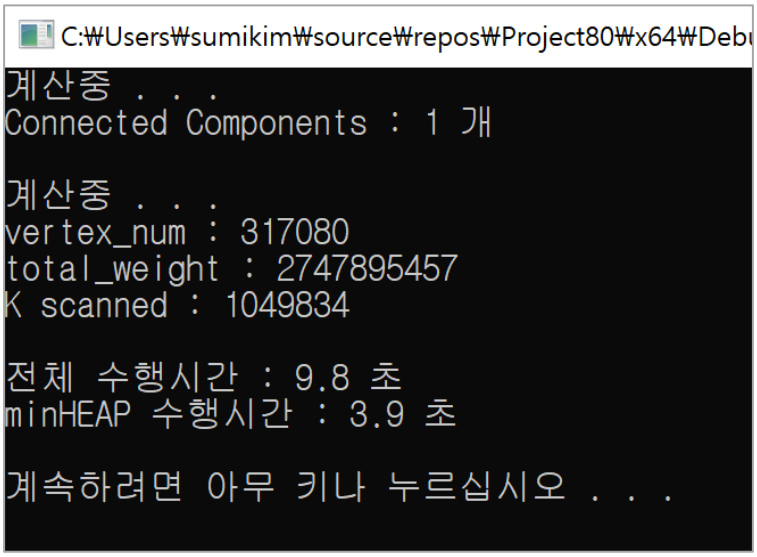
먼저 Connected Component가 1개 일 때, KruskalMST1를 사용하는 것이 위 부분이다. dsjSet 함수를 호출할 때 Min heap의 구성이 바뀌었으므로 다시 input file을 처음부터 입력 받아 처음 상태의 graph 와 Min heap을 새로 만들어줘야 한다. 새로 만든 graph와 Min heap을 KruskalMST1 함수에 넣어주면 1개의 Connected Component를 가지는 그래프의 MST weight 총합을 반환해준다. vertex_num은 'number of vertices in the component'를 의미하는데, 해당 그래프가 하나의 Component이므로 그래프의 총 vertex 개수와 동일한 값을 가진다. total_weight는 그래프의 MST를 구성하는 edge들의 weight 총합을 나타내는 값이며, K scanned는 MST의 구축이 완성될 때 까지 Min heap에서 pop된 edge의 개수를 의미한다.

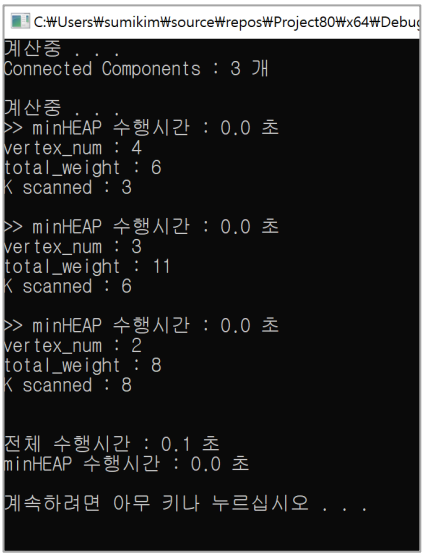
```
288     // Make MST2 (Connected Component가 2개 이상)
289     else if (cntCmp > 1) {
290         int* vertex; int start = 0;
291         graph = createGraph(n_vertices, n_edges); n = 0;
292         element = malloc(sizeof(struct Element) * n_edges);
293 
294         fp = fopen(input_file, "r");
295         fscanf(fp, "%d%d%d", &x, &y, &z);
296 
297         printf("계산중 . . .₩n");
298         for (int i = 0; i < n_edges; i++) {
299             fscanf(fp, "%d%d%d", &x, &y, &z);
300             graph->edge[i].src = x;
301             graph->edge[i].dest = y;
302             graph->edge[i].weight = z;
303             insert_min_heap(element, z, &(n), i);
304         } fclose(fp);
305 
306         vertex = KruskalMST2(graph, element, &(n));
307         for (int t = 0; t < cntCmp; t++) {
308             graph = createGraph(n_vertices, n_edges); n = 0;
309             element = malloc(sizeof(struct Element) * n_edges);
310 
311             fp = fopen(input_file, "r");
312             fscanf(fp, "%d%d%d", &x, &y, &z);
313 
314             start2 = clock();
315             for (int i = 0; i < n_edges; i++) {
316                 fscanf(fp, "%d%d%d", &x, &y, &z);
317                 graph->edge[i].src = x;
318                 graph->edge[i].dest = y;
319                 graph->edge[i].weight = z;
320                 insert_min_heap(element, z, &(n), i);
321             } fclose(fp); end2 = clock();
322 
323             float time2 = (float)(end2 - start2) / CLOCKS_PER_SEC;
324             printf(">> minHEAP 수행시간 : %.1f 초₩n", time2);
325 
326             int *inside = malloc(sizeof(int) * cntCmp_arr[t]);
327             for (int u = 0; u < cntCmp_arr[t]; u++)
328                 inside[u] = vertex[start++];
329 
330             KruskalMST3(graph, element, &(n), inside, cntCmp_arr[t]);
331             printf("vertex_num : %d₩n", cntCmp_arr[t]);
332             printf("total_weight : %lld₩n", total_weight);
333             printf("K scanned : %lld₩n₩n", count);
334             fprintf(out, "%d %lld₩n", cntCmp_arr[t], total_weight);
335             total_weight = 0; count = 0;
336         }
337     }
338 }
```

다음으로 Connected Component가 2개 이상 일 때, KruskalMST2와 KruskalMST3을 사용하는 부분이다. KruskalMST2를 통해 그래프의 각 Components가 어떤 vertex들로 구성되어 있는지를 계산하고, 계산 결과를 바탕으로 해당 vertex를 포함하는 edge들만 KruskalMST3 내부에서 선택의 대상으로 고려하며 MST를 만든다.

```
337     // 시간 측정 후 출력
338     end1 = clock();
339     float time1 = (float)(end1 - start1) / CLOCKS_PER_SEC;
340     float time2 = (float)(end2 - start2) / CLOCKS_PER_SEC;
341     printf("₩n전체 수행시간 : %.1f 초₩n", time1);
342     printf("minHEAP 수행시간 : %.1f 초₩n", time2);
343
344     fclose(out);
345     system("pause");
346     return 0;
347 }
```

마지막으로 코드 수행 시간을 출력하는 부분이다. 입출력 시간을 제외한 순수 실행 시간과 Min heap을 구축하는데 걸리는 시간을 측정하였으며, start1과 end1은 각각 245, 338번째 줄에서 측정되었고, start2과 end2는 그래프의 Connected Component가 1개일 때 271, 278번째 줄에서, 그래프의 Connected Component가 2개 이상 일 때 315, 322번째 줄에서 측정되었다.





위 화면은 코드를 실행하면 확인할 수 있는 콘솔창의 모습이다. 그래프의 Connected Components가 1개인 경우 왼쪽과 같은 화면을 확인할 수 있으며, Connected Components가 2개 이상인 경우 오른쪽과 같은 화면을 확인할 수 있다.

4. 시간복잡도 분석 : O(|E|log|V|)

Kruskal 알고리즘을 이용해 MST를 계산하는 이 코드의 시간복잡도는 최악의 경우 수행시간이 O(|E|log|V|) 이다. 이는 그래프의 edge들을 크기 순서대로 정렬하고, edge를 하나씩 선택하면서 tree를 완성해 나가는 과정에서 Min heap과 Disjoint-set을 사용했기 때문에 가능하다. Kruskal 알고리즘은 Greedy 알고리즘으로, 그래프의 모든 edges중 크기가 작은 것부터 차례대로, cycle이 발

생하지 않도록 선택해 나가며 MST를 완성한다. 이를 위해서는 먼저 그래프의 모든 edges를 크기 순서대로(작은 것부터 큰 것) 정렬하는 작업이 필요한데, 해당 작업을 Min heap으로 처리해 주었기 때문에 여기서 걸리는 시간은 $O(|E|\log|E|)$ 이다 (heap sort). 그 다음 매 단계마다 edge를 하나씩 선택함에 따라 여러 개의 forest가 발생하는데, 이를 Disjoint-set으로 관리해 tree의 생성 및 병합 작업을 수행해 주었다. 총 $|V|-1$ 개의 edge를 뽑을 때 까지 $|E|$개의 edge들을 살펴보아야 하는데, Disjoint-set의 최대 깊이는 $\log_2(|V|-1)$ 이므로 최악의 경우 $|E|\log|V|$의 시간이 소요된다. 따라서 전체 소요시간은 $O(|E|\log|E|+|E|\log|V|) = O(|E|\log|V|)$ 가 되는 것이다.

5. 느낀점

이번 과제를 통해 억 단위가 넘어가는 크기의 데이터를 처음 다뤄 보았는데, 확실히 작은 크기의 데이터보다 다루기가 까다롭고 어려웠다. 연산 결과가 너무 많아 에러를 눈에 보이게 출력할 수 없어 더욱 꼼꼼하게 코드를 작성해야 했고, 평소 사용하던 변수 타입이 해당 데이터를 담지 못해 long long, %lld과 같은 타입의 변수도 처음 사용해 보았다. 실제로 일상 속에서 사용되는 프로그램은 이보다 더 큰 '빅데이터'를 다루는 경우가 대부분이기 때문에, 크기가 큰 데이터를 많이 다뤄보는 연습이 필요함을 느낄 수 있었다.