CSE3081-01 알고리즘 설계와 분석

[숙제 5]

Minimum Spanning Tree 알고리즘의 구현

20231604 컴퓨터공학과 정유연

2024-12-25

<목차>

1. 컴퓨터 실험환경

- OS: Windows 11 Home
- CPU: 13th Gen Intel(R) Core(TM) i5-1340P 1.90 GHz
- RAM: 16.00GB
- Compiler: Visual Studio 22 Release Mode/x64 Platform

2. 자료 구조

1) Min Heap

Kruskal알고리즘을 수행하기 위해서 가장 먼저 edge의 weight에 따라 오름차순 정렬을 하는 과정이 필요하다.

이를 위해 입력으로 주어진 graph의 정보 중에서 weight에 해당하는 정보만 Min heap에 저장했다.

```
vector<int> from(n_edges), to(n_edges), weight(n_edges, MAX_WEIGHT);
for (int i = 0; i < n_edges; i++) {
    input_file >> from[i] >> to[i] >> weight[i];
    weight[i] = (weight[i] > MAX_WEIGHT) ? MAX_WEIGHT : weight[i];
}
input_file.close();

MinHeap minheap(n_edges, MAX_WEIGHT, weight);
```

Weight에 대한 정보를 min heap 자료구조에 저장하는 main함수 코드

Min heap을 만들기 위해 edge개수만큼의 노드가 필요하므로 n_edges,

weight의 최대값을 저장할 MAX_WEIGHT,

그리고 i=0부터 n_edges-1까지의 인덱스에 각 edge의 weight를 저장한 벡터인 weight를 매개변수로 준다.

Weight를 저장할 때 최대값인 MAX_WEIGHT보다 값이 크다면 입력 받은 weight가 아닌 MAX_WEIGHT를 저장하도록 한다.

```
class MinHeap {
    vector<int> indices;
    vector<int> weight;

public:
    MinHeap(int size, int MAX, const vector<int>& unsorted_weight) {
        indices.reserve(size);
        weight.resize(size, MAX);

        for (int i = 0; i < unsorted_weight.size(); i++) {
            indices.push_back(i);
            weight[i] = unsorted_weight[i];
        }
        sort();
    }

    void sort() {
        for (int i = (indices.size() / 2) - 1; i >= 0; i--) {
            adjust(i);
        }
    }
```

1 2 3 4

정렬되지 않은 배열을 Min heap으로 만드는 생성자 코드

Min heap의 생성자에서

① edge의 인덱스를 weight순서대로 저장할 배열 indices를 만들고

② 매개변수로 받은 weight 배열을 로컬로 복사한 다음

③ min heap을 만드는 함수 sort를 실행시킨다.

weight배열은 weight의 최대값인 MAX로 먼저 초기화를 했다.

④ sort함수는 정렬되지 않은 배열을 min heap으로 만드는 함수로, 인덱스의 절반부터 0까지 adjust함수를 실행한다. 이 반복문이 끝나면 indices배열에는 0부터 n_edges-1까지 인덱스에 min heap을 만족하는 edge의 인덱스가 저장되어 있을 것이다.

```
void adjust(int idx) {
    int curIdx = idx;
    while (1) {
        int leftChild = 2 * curIdx + 1;
        int rightChild = 2 * curIdx + 2;
        int smallest = curIdx;

        if (leftChild < indices.size() && weight[indices[leftChild]] < weight[indices[smallest]]) {
            smallest = leftChild;
        }
        if (rightChild < indices.size() && weight[indices[rightChild]] < weight[indices[smallest]]) {
            smallest = rightChild;
        }
        if (smallest == curIdx) break;

        swap(indices[curIdx], indices[smallest]);
        curIdx = smallest;
    }
}
```

나와 자식 대소 비교

adjust함수에서 heap을 만드는 과정을 실행한다.

현재 인덱스의 자식을 leftChild와 rightChild에 저장한 후 이 둘의 weight를 현재 인덱스의 weight와 비교하여 더 작다면 현재 인덱스와 스왑한다.

이 과정을 내 자식이 모두 나보다 클 때까지 반복한다. 이렇게 해서 min heap을 만들 수 있다. 이때의 시간 복잡도에 관한 논의는 5. 시간복잡도 에서 다룬다.

```
int extractMin() {
    int minIdx = indices[0];
    indices[0] = indices.back();
    indices.pop_back();
    adjust(0);

    return minIdx;
}
```

kruskal알고리즘에서 쓸 extractMin함수이다.

Min heap을 만족하는 배열인 indices는 완벽하게 오름차순으로 정렬되어 있지는 않지만, 첫번째 인덱스인 0번 인덱스에는 가장 작은 값이 저장되어 있을 것이다. 이를 이용하여 최소값을 뽑아내는 extractMin함수를 작성했다.

0번 인덱스를 minIdx에 저장하고, 맨 마지막 인덱스를 0번 인덱스로 옮겨 adjust 함수를 실행하면 다시 min heap을 만족하는 indices배열이 완성된다. 이후 minIdx를 반환해서 현재 indices배열의 최소값을 추출한다.

이렇게 하여 현재 배열에서 가장 작은 최소값을 추출할 수 있고, 추출하여 최소값이 삭제된 후에도 min heap을 만족하는 배열을 유지할 수 있다.

```
bool empty() {
    return indices.empty();
}
```

indices배열이 비었는지 아닌지 확인하는 함수이다. MST를 만들 때 배열이 빌 때까지 첫번째 원소를 pop시키기 때문에 empty()함수를 만들었다.

2) Disjoint Set

Kruskal 알고리즘에서 노드를 결합할 때, disjoint set을 이용하면 O(ElogV)의 시간에 MST를 완성할 수 있다. 이를 위해 DisjointSet을 저장하는 클래스를 구현했다.

```
class DisjointSet {
    vector<int> parent, rank;

public:
    vector<int> componentSize;
    vector<ll> componentWeight;

    DisjointSet(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; i++) parent[i] = i;
        componentSize.resize(n, 1);
        componentWeight.resize(n, 0);
    }
    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }
```

각 노드의 root를 저장할 배열 parent와 랭크를 저장할 배열 rank를 선언하여 사이즈를 초기화한다. 초기화할 때 모든 노드의 parent는 자기 자신이다.

Component size에는 연결된 노드 덩어리의 크기를 의미한다. 즉 7개의 노드가 1번 인덱스를 root로 연결되어있을 때, componentSize[1]은 7이다.

Component weight에는 연결된 노드 덩어리의 weight의 총합을 의미한다. 즉 7개의 노드가 1번 인덱스를 root로 연결되어 있을 때, componentSize[1]은 7개 노드의 weight의 합이다.

find는 path compression을 이용하여 자신의 root노드를 저장하는 함수이다.

재귀적으로 함수를 호출해 parent[x]에 root노드를 저장할 수 있다.

```
void unionSets(int x, int y, int edgeWeight) {
    int rootX = find(x);
    int rootY = find(y);

    if (rootX != rootY) {
        if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
            componentSize[rootX] += componentSize[rootY];
            componentWeight[rootX] += componentWeight[rootY] + edgeWeight;
        }
        else if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
            componentSize[rootY] += componentSize[rootX];
            componentWeight[rootY] += componentWeight[rootX] + edgeWeight;
        }
        else {
            parent[rootY] = rootX;
            rank[rootX]++;
            componentSize[rootX] += componentSize[rootY];
            componentWeight[rootX] += componentWeight[rootY] + edgeWeight;
        }
    }
    else {
        componentWeight[rootX] += edgeWeight;
    }
}
```

두 개의 노드를 합치는 union함수이다. 인덱스 x의 root를 찾아 rootX에 저장하고, y의 root를 찾아 rootY에 저장한다. 두 노드의 부모가 다르다면 union을 시작한다. Rank가 더 큰 root에 더 작은 root가 합쳐진다.

이때 부모가 되는 root의 인덱스에 그래프의 크기와 weight를 저장하는 componentSize, componentWeight를 갱신한다.

두 노드의 부모가 같아 union하지 않을 때도 weight는 합쳐준다.

3. 알고리즘 구현

Minimum Spanning Tree를 만들기 위해서는 다음의 단계가 필요하다.

1. 모든 Edge를 weight가 작은 순으로 정렬한다.
2. Cycle을 생성하지 않을 때, 가장 작은 weight를 가지는 Edge의 시작과 끝에 해당하는 노드를 연결한다.
3. 모든 edge를 탐색하며 2번 과정을 반복하여 MST를 만든다.

1번은 weight를 저장한 벡터를 매개변수로 전달하여 min heap을 만들도록 했다.

2번에서 가장 작은 weight를 가지는 edge를 min heap의 0번 인덱스를 추출하여 얻었고, Disjoint Set의 union함수를 이용해 각 꼭짓점을 cycle이 생기지 않는 선에서 합쳤다.

3번은 min heap이 empty가 될 때까지 edge를 추출하고 union하며 MST를 만들었다.

```
int n_vertices, n_edges, MAX_WEIGHT;
input_file >> n_vertices >> n_edges >> MAX_WEIGHT;

vector<int> from(n_edges), to(n_edges), weight(n_edges, MAX_WEIGHT);
for (int i = 0; i < n_edges; i++) {
    input_file >> from[i] >> to[i] >> weight[i];
}
input_file.close();

1  MinHeap minheap(n_edges, MAX_WEIGHT, weight);
   DisjointSet ds(n_vertices);

3  while (!minheap.empty()) {
       int idx = minheap.extractMin();
       int u = from[idx];
       int v = to[idx];
       int w = weight[idx];

2      if (ds.find(u) != ds.find(v)) {
           ds.unionSets(u, v, w);
       }
   }
```

Kruskal 알고리즘

이때 유의한 점은 from, to, weight의 정보를 갖는 edge를 구조체를 이용해서 저장한 것이 아니라, 각 정보에 맞는 배열에 저장한 것이다.

```
// a structure to represent a weighted edge in graph
struct Edge {
    int src, dest, weight;
};
```

Edge 정보를 구조체에 저장한 예시

이렇게 구현한다면 입력을 받을 때도 보다 명시적이고 코드를 작성하기도 쉬워지지만 인덱스로 접근하기 어렵고 저장용량을 더 많이 사용한다.

```
vector<int> from(n_edges), to(n_edges), weight(n_edges, MAX_WEIGHT);
for (int i = 0; i < n_edges; i++) {
    input_file >> from[i] >> to[i] >> weight[i];
}
input_file.close();
```

Edge 정보를 각각의 배열에 저장한 코드

edge에서 출발, 도착, 무게에 해당하는 from, to, weight 정보를 모두 다른 배열에 저장했지만, 한 엣지에 대해서 모두 같은 인덱스로 접근이 가능하다.

```
MinHeap minheap(n_edges, MAX_WEIGHT, weight);
DisjointSet ds(n_vertices);

while (!minheap.empty()) {
    int idx = minheap.extractMin();
    int u = from[idx];
    int v = to[idx];
    int w = weight[idx];

    if (ds.find(u) != ds.find(v)) {
        ds.unionSets(u, v, w);
    }
}
```

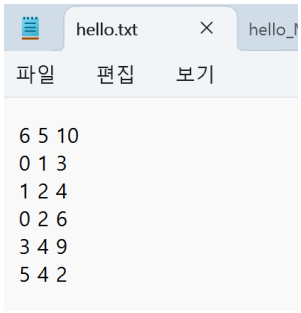
이후에는 min heap이 empty이기 전까지 weight가 가장 작은 edge를 뽑아서, 양 꼭짓점을 이을 때 cycle이 생기지 않는다면 이어주는 작업을 반복한다.

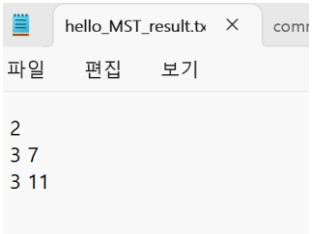
이 과정을 반복하여 Minimum Spanning Tree를 만들 수 있다.

4. 실험 결과

이 코드는 connected graph가 아닌 경우에도 동작하며, 각 connected component의 꼭짓점의 개수와 total weight를 정확히 구할 수 있다.

- Unconnected graph일 경우 input과 output예시





- 결과 출력 코드

앞서 *6,7페이지의 2. 자료구조 2) disjoint set*에서 설명한 **componentSize**와 **componentWeight**를 이용하여 unconnected graph일 때도 각 connected component의 꼭짓점의 개수와 total weight를 구하도록 했다.

```
vector<int> componentNum;
for (int i = 0; i < n_vertices; i++) {
    if (i == ds.find(i)) {
        componentNum.push_back(i);
    }
}
```

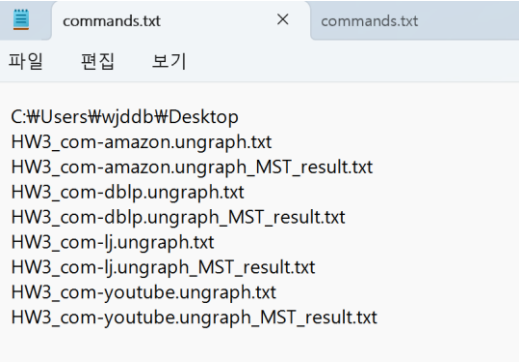
먼저 disjoint set의 모든 노드에서 parent가 자기 자신인 노드를 찾는다. Parent가 자기 자신인 노드는 루트 노드밖에 없고, 이 루트 노드의 개수가 connected component의 개수인 componentNum에 저장된다.

다음으로는 componentNum 벡터의 크기를 total_component에 저장하여 총 component의 size를 결과 파일에 출력한다.

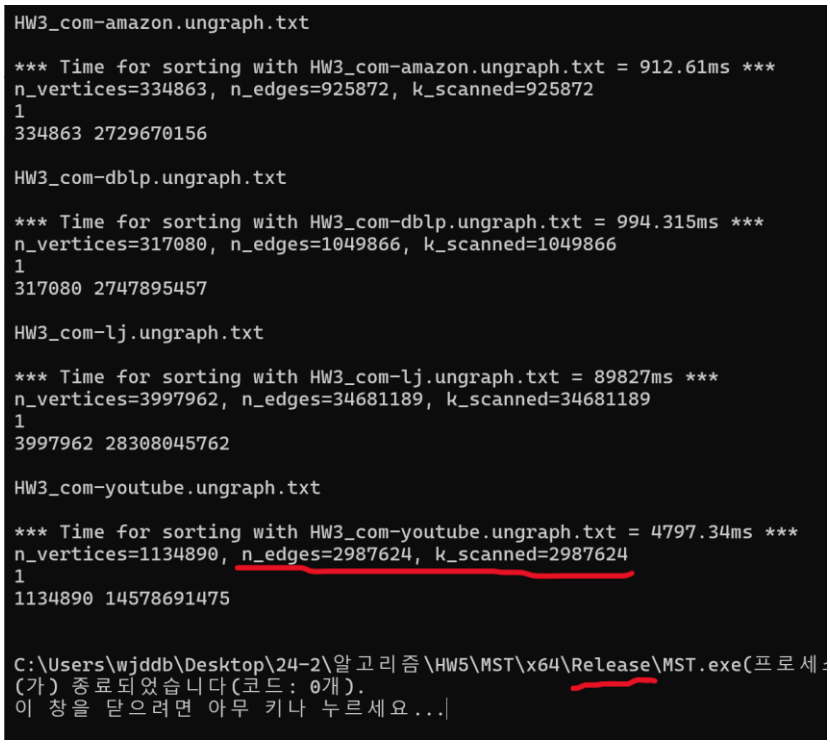
```
int total_component = componentNum.size();
cout << total_component << "₩n";
MSTresult << total_component << "₩n";

for (int i = 0; i < total_component; i++) {
    cout << ds.componentSize[componentNum[i]] << " " << ds.componentWeight[componentNum[i]] << "₩n₩n";
    MSTresult << ds.componentSize[componentNum[i]] << " " << ds.componentWeight[componentNum[i]] << "₩n";
}
```

이후 total_component만큼의 반복문 안에서 **컴포넌트의 꼭짓점 개수**와 **총 weight**를 결과 파일에 출력한다.

프로그램이 정상적으로 종료되면 모든 입력 파일에 대한 출력 파일 이름이 commands.txt에 저장 되어있다.



입력 파일 4개에 대하여 5번씩 실행한 수행시간의 평균을 표에 넣었다.



release모드에서 실행결과 예시

파일 이름	작동 여부	MST weight	수행 시간(초)	$k_{scanned}$
HW3_com-amazon.ungraph.txt	YES	2729670156	0.811397	925872
HW3_com-dblp.ungraph.txt	YES	2747895457	0.833812	1049866
HW3_com-lj.ungraph.txt	YES	28308045762	84.43866	34681189
HW3_com-youtube.ungraph.txt	YES	14578691475	4.101866	2987624

Min heap에 모든 edge를 넣고 heap이 empty일 때까지 반복문을 돌리며 Kruskal 알고리즘을 수행했기 때문에 k_scanned가 edge의 개수와 같음을 알 수 있다.

5. 시간 복잡도

이번 프로그램은 크게 두 부분으로 나누어 설명할 수 있다. Edge의 weight의 크기에 따라 min heap을 만드는 부분과, min heap의 0번 인덱스를 pop하여 얻은 edge의 양 꼭짓점을 union하는 부분이다.

1) Min Heap 만들기

먼저 2. 자료구조 1) min heap에서 소개한, weight의 크기 순으로 min heap을 만드는 코드의 주요 내용이다.

```
void sort() {
    for (int i = (indices.size() / 2) - 1; i >= 0; i--) {
        adjust(i);
    }
}
void adjust(int idx) {
    int curIdx = idx;
    while (1) {
        int leftChild = 2 * curIdx + 1;
        int rightChild = 2 * curIdx + 2;
        int smallest = curIdx;

        if (leftChild < indices.size() && weight[indices[leftChild]] < weight[indices[smallest]]) {
            smallest = leftChild;
        }
        if (rightChild < indices.size() && weight[indices[rightChild]] < weight[indices[smallest]]) {
            smallest = rightChild;
        }
        if (smallest == curIdx) break;

        swap(indices[curIdx], indices[smallest]);
        curIdx = smallest;
    }
}
```

sort함수는 정렬되지 않은 배열을 min heap을 만족하게 만드는 함수이다.

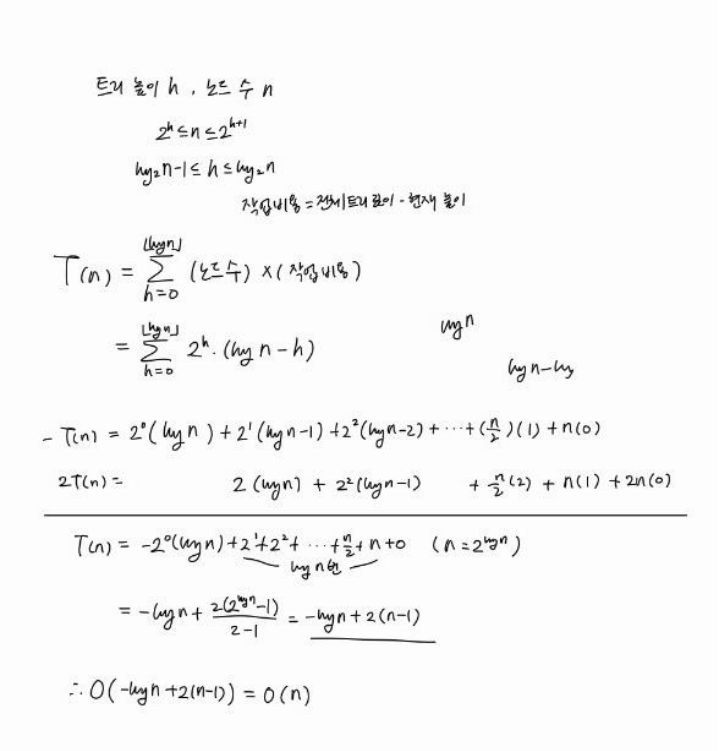
이는 배열의 중간 인덱스부터 루트 노드까지 내려가며 adjust함수를 호출한다.

Adjust함수는 각 노드에 대해 최악의 경우 해당 서브 트리의 높이만큼 반복하기 때문에 노드 수가 n일 때 높이는 logn이다.

이에 대한 계산 크기를 T(n)으로 표현했을 때

$$T(n) = \sum_{h=0}^{\log n} 2^h \times (\log n - h)$$

이고 이를 정리하면 T(n)=2(n-1)-logn 이기 때문에 sort함수의 시간 복잡도는 **O(E)**가 된다.



Heapify 비용 T(n)의 계산 과정

2) 양 끝 꼭짓점 union하기

다음은 3. 알고리즘에서 소개한 kruskal알고리즘의 주요 내용이다.

```
while (!minheap.empty()) {
    int idx = minheap.extractMin();
    int u = from[idx];
    int v = to[idx];
    int w = weight[idx];

    if (ds.find(u) != ds.find(v)) {
        ds.unionSets(u, v, w);
    }
}
```

Kruskal 알고리즘

모든 edge에 대해서 가장 작은 weight를 가진 edge의 from, to, weight를 저장하

고 from과 to가 연결되어 있지 않다면, 즉 cycle을 생성하지 않는다면 union을 한다.

최솟값을 찾는 extractMin함수는 0번 인덱스를 추출하고 맨 마지막 노드와 교환한 후, adjust함수를 실행한다. Adjust함수는 각 노드에 대해 최악의 경우 해당 서브트리의 높이만큼 반복하기 때문에, 노드 수가 E인 min heap에서 시간복잡도는 O(1)+O(logE)인 O(logE)가 된다. 2. 자료구조 1) min heap의 adjust에 사진과 함께 한 설명이 있다.

find는 자신의 루트 노드까지 재귀적으로 탐색을 하므로, 최악의 경우 꼭짓점이 V인 트리의 최대 높이인 logV까지 탐색한다. 즉 find의 시간 복잡도는 O(logV)이다.

Union 함수는 꼭짓점 두개에 대해서 find를 한 다음 둘의 rank를 비교하는 연산을 하기 때문에, 2*Find op's+O(1)을 가진다. 즉 union의 시간 복잡도도 O(logV)이다.

이러한 연산을 min heap이 empty일 때까지 pop시키며 진행하기 때문에 Edge의 개수만큼 반복한다.

즉 Kruskal 알고리즘의 시간복잡도는 O(ElogE)+O(ElogV)가 된다.

이때 E는 꼭짓점의 개수 V에 대해 0<=E<=V(V+1)/2 를 만족하기 때문에 logE<=logV^2=2logV를 만족하므로 bigO notation에서 O(ElogE)=**O(ElogV)**로 표현할 수 있다.

따라서 이번 프로그램의 시간복잡도는 O(E)+O(ElogV)=**O(ElogV)**이다.

6. 시간 복잡도 비교

Min heap과 Disjoint Set을 이용한 Kruskal 알고리즘의 이론적 시간 복잡도는 꼭짓점의 개수 V, 모서리의 개수 E일 때 **O(ElogV)**이다.

시간 복잡도에 대한 계산은 입력 데이터가 n배 증가할 때 수행 시간이 몇 배 증가하느냐를 통해 알 수 있다. 그러나 이번 과제의 테스트 케이스는 4개이고 입력 데이터의 크기를 조정할 수 없으므로 위와 같은 방법을 이용할 수 없다.

따라서 측정한 수행시간이 얼마나 ElogV에 비례하는지를 확인해보았다.

이를 확인하기 위해 4개의 수행시간을 ElogV로 나눈 ratio와 E^2, V, logE, E*V로 나눈 ratio를 구했다.

	amazon	dblp	lj	youtube
V	334863	317080	3997962	1134890
E	925872	1049866	34681189	2987624
time(s)	0.8113974	0.833812	84.43866	4.101866
time/ElogV	4.77497E-08	4.35E-08	1.11E-07	6.83E-08
time/E^2	9.46524E-13	7.56E-13	7.02E-14	4.6E-13
time/V	2.42307E-06	2.63E-06	2.11E-05	3.61E-06
time/logE	0.040937379	0.041687	3.371121	0.190691
time/E*V	2.61707E-12	2.5E-12	6.09E-13	1.21E-12

이 5가지 방법(ElogV, E^2, V, logE, E*V) 중에서 어떤 것이 가장 수행시간과 유사한 경향을 보이는지 확인하기 위해 Normalized Range를 구해 확인했다.

$$Normalized\ Range = \frac{Max - Min}{Mean}$$

5개의 방법으로 구한 4가지 경우의 ratio중에서 최대값과 최소값의 차이를 4개 수행시간의 평균으로 나누었다.

	range	mean	range/mean
time/ElogV	6.75578E-08	6.76214E-08	**0.99906**
time/E^2	8.76322E-13	5.5819E-13	**1.569935**
time/V	1.86974E-05	7.44687E-06	**2.510766**
time/logE	3.330183661	0.911109013	**3.655088**
time/E*V	2.00808E-12	1.73515E-12	**1.1573**

최대값과 최소값의 차는 데이터의 범위를 나타내고, 이를 평균으로 나누면 데이터 분포가 평균 대비 얼마나 일관되었는지를 알 수 있다.

즉 Normalized Range값이 작을수록 평균에 비해 더 유사한 패턴을 보인다는 결론을 낼 수 있다.

이때 5가지 방법 중에서 ElogV의 Normalized Range가 다른 방법의 Normalized Range보다 확연히 작은 모습을 확인했다. 이를 통해 우리가 작성한 프로그램의 수행 시간은 **O(ElogV)를 따른다**는 사실을 검증된 결과로 알 수 있다.