알고리즘 설계와 분석 Project 5

서강대학교
SOGANG UNIVERSITY

담당 교수 : 임인성

이름 : 박민준

학번 : 20212020

1. Algorithm Description

[실험 환경]

OS: Windows 11 Home

CPU: Intel(R) Core(TM) i5-1240P @ 1.70GHz

RAM: 16.00GB

Compiler: Visual Studio 2022 Release Mode/x64 Platform

1. insert_min_heap()

```
struct Edge {
    int src, dest, weight;
};

// Min Heap 구현
vector<Edge> heap;

void insert_min_heap(Edge item) {
    heap.push_back(item);
    size_t child = heap.size() - 1;
    while (child > 0) {
        int parent = (child - 1) / 2;
        if (heap[parent].weight <= item.weight) break;
        heap[child] = heap[parent];
        child = parent;
    }
    heap[child] = item;
}
```

- insert_min_heap 함수는 간선을 최소 힙에 삽입하면서 최소 힙 조건을 유지하는 작업을 수행한다. 먼저 삽입할 간선 item을 힙의 끝에 추가한다. 이후, 삽입된 간선의 위치 child에서 시작하여 부모 노드 parent와 가중치를 비교한다. 만약 삽입된 간선의 가중치가 부모 노드의 가중치보다 작다면 부모 노드를 자식 위치로 내려 보내고 삽입된 간선의 위치를 부모 위치에 이동시킨다. 이 과정을 루트에 도달하거나 삽입된 간선이 부모보다 크거나 같은 경우까지 반복하여 삽입된 간선이 올바른 위치에 배치된다.

- 최소 힙은 완전 이진 트리 구조를 유지하여 트리의 높이를 logN으로 제한한다. (여기서 N은 힙 내 간선의 개수를 의미한다.) 삽입 과정에서 부모 노드와 비교하여 최소한의 연산으로 올바른 위치를 찾는 것이 시간 복잡도 최적화의 핵심이다.

- 최악의 경우, 삽입된 노드가 루트까지 이동해야 하므로 O(logN)의 시간 복잡도를 갖는다.

2. delete_min_heap()

```
Edge delete_min_heap() {
    if (heap.empty()) {
        cerr << "Heap is empty.\n";
        exit(1);
    }

    Edge root = heap[0];
    Edge tmp = heap.back();
    heap.pop_back();

    if (heap.empty()) return root;

    size_t parent = 0, child = 1, n = heap.size();

    while (child < n) {
        if (child + 1 < n && heap[child].weight > heap[child + 1].weight) child++;
        if (tmp.weight <= heap[child].weight) break;
        heap[parent] = heap[child];
        parent = child;
        child = 2 * parent + 1;
    }
    heap[parent] = tmp;

    return root;
}
```

- delete_min_heap 함수는 힙에서 최소값을 갖는 루트 노드를 제거하고 힙의 구조를 다시 정렬하는 역할을 한다. 먼저 힙이 비어 있는 경우를 확인하여 에러를 처리한다. 루트 노드의 간선에 해당하는 heap[0]을 저장한 뒤, 힙의 마지막 노드 tmp를 루트로 이동시킨다. 힙에서 마지막 노드를 제거한 후, 루트부터 시작하여 자식 노드들과 비교하면서 힙 구조를 재조정한다. 두 자식 노드 중 더 작은 노드와 비교하며, 부모 노드가 더 크면 자식과 자리를 바꾼다. 이 과정을 반복하여 마지막 노드 tmp가 적절한 위치에 배치될 때까지 진행한다.

- 트리의 높이가 logN이므로 삭제 후 재조정 작업은 O(logN)의 시간 복잡도를 갖는다.

3. Find()

```
// Disjoint Set 구현
vector<int> parent, node_rank, componentSize;
vector<ll> componentWeight;
int total_components;

int Find(int x) {
    int root = x;
    while (root != parent[root]) root = parent[root];
    while (x != root) {
        int next = parent[x];
        parent[x] = root;
        x = next;
    }
    return root;
}
```

- Find 함수는 Union-Find 자료구조에서 특정 노드의 루트를 찾는 역할을 한다. 경로 압축(Path Compression) 기법을 사용하여 노드의 부모를 루트로 직접 설정함으로써 이후의 연산 속도를 향상시킨다. 먼저 입력된 노드의 루트를 찾기 위해 부모 노드를 따라간다. 루트에 도달하면 경로 상의 모든 노드의 부모를 루트로 설정하여 트리의 깊이를 최소화한다.

- 위 코드에서 x는 루트를 찾고자 하는 노드이며, root는 루트를 저장하는 변수로 처음에는 x로 초기화되며 루트를 찾는 과정에서 변경된다. Parent는 각 노드의 부모를 저장하는 배열로, 트리 구조를 표현한다.

- 경로 압축은 각 노드를 루트에 직접 연결하여 트리의 높이를 최소화한다. 이후 연산에서 Find의 호출을 상수 시간에 가깝게 만드는 것이 시간 복잡도 최적화의 핵심이다.

- 경로 압축을 한 Find 함수의 시간 복잡도는 O(a(N))이 되는데, a(N)은 역아커만 함수를 의미하며 대부분의 현실적인 입력 크기에서 5보다 작은 값을 가진다. (N은 노드의 총 개수를 의미한다.) 따라서 경로 압축을 적용한 Find 함수는 거의 상수 시간에 가까운 효율적인 연산이라고 볼 수 있다.

4. Union()

```
void Union(int x, int y, int weight) {
    int rootX = Find(x);
    int rootY = Find(y);

    if (rootX == rootY) return;

    if (node_rank[rootX] < node_rank[rootY]) {
        parent[rootX] = rootY;
        componentWeight[rootY] += componentWeight[rootX] + weight;
        componentSize[rootY] += componentSize[rootX];
    }
    else {
        parent[rootY] = rootX;
        componentWeight[rootX] += componentWeight[rootY] + weight;
        componentSize[rootX] += componentSize[rootY];
        if (node_rank[rootX] == node_rank[rootY]) node_rank[rootX]++;
    }

    total_components--;
}
```

- Union 함수는 두 노드가 서로 다른 트리에 속한 경우 두 트리를 병합하는 역할을 한다. 먼저 각 노드의 루트를 찾고, 트리의 높이인 rank를 비교하여 더 낮은 rank의 트리를 높은 rank의 트리 아래로 병합한다. 이때, 병합된 트리의 가중치와 크기를 갱신한다. 만약 rank가 같을 경우 한쪽 rank를 증가시킨다.

- 위 코드에서 x,y는 병할할 두 노드를 의미하며 rootX, rootY는 각각 x와 y의 루트를 저장하는 변수다. node_rank는 각 트리의 높이를 저장하는 배열로, 병합 시 높이를 비교한다. componentWeight와 componentSize는 병합 후 트리의 가중치와 크기를 갱신한다.

- Union by rank는 트리의 높이를 최소화하여 효율성을 보장한다. 이는 경로 압축과 함께 사용되어 연산을 거의 상수 시간으로 최적화한다.

- Union 함수도 경로 압축을 한 Find 함수와 마찬가지로 O(a(N))의 시간 복잡도를 갖게 되는데, 이는 트리의 높이가 경로 압축과 rank 기반 병합으로 제한되기 때문이다.

5. getComponentInfo()

```
vector<pair<int, ll>> getComponentInfo() {
    vector<pair<int, ll>> result;
    for (int i = 0; i < parent.size(); i++) {
        int root = Find(i);
        if (root == i) {
            result.push_back({ componentSize[i], componentWeight[i] });
        }
    }
    return result;
}
```

- getComponentInfo 함수는 모든 정점에 대해 순차적으로 접근하며 그 정점의 루트를 찾는다. Find(i)를 호출하여 i번 정점이 속한 연결 컴포넌트의 루트를 얻는다.

- 루트 노드는 parent[i] == i인 정점이다. 현재 노드가 루트 노드인지 확인하고 루트 노드인 경우에만 해당 컴포넌트의 정보를 기록한다.

- 연결 컴포넌트의 크기와 가중치 합을 결과 벡터 result에 저장하고 이를 반환한다.

- Find 함수는 각 정점에 대해 호출되며, 경로 압축을 통해 O(a(N))의 시간복잡도로 수행된다. 따라서 해당 함수의 전체 시간복잡도는 O(N*a(N))이다. 여기서 N은 Union-Find 자료구조의 전체 노드 수, 즉 그래프의 정점의 개수를 의미한다.

6. Kruskal Algorithm

```
void kruskal(ifstream& graph, int n, int n_vertices) {
    Edge e;
    for (int i = 0; i < n; i++) {
        graph >> e.src >> e.dest >> e.weight;
        insert_min_heap(e);
    }

    int mst_edge_count = 0;
    int k_scanned = 0;  // 처리한 간선의 개수를 추적
    while (!heap.empty() && mst_edge_count < n_vertices - 1) {
        Edge edge = delete_min_heap();
        k_scanned++;  // 간선을 처리할 때마다 증가

        if (Find(edge.src) != Find(edge.dest)) {
            Union(edge.src, edge.dest, edge.weight);
            mst_edge_count++;
        }
    }

    cout << "Processed edges until MST completion: " << k_scanned << '\n';
}
```

- kruskal 함수는 Kruskal 알고리즘의 전체 흐름을 구현한 함수로, 주어진 그래프에서 MST를 계산

한다. 간선 데이터를 최소 힙에 삽입한 후, 최소 가중치 간선부터 하나씩 처리한다. 간선이 두 개의 다른 트리를 연결하는 경우, 해당 간선을 MST에 포함시키고 두 트리를 병합한다. MST가 완성될 때까지 이 과정을 반복한다.

- 먼저, 각 간선을 읽어서 최소 힙에 삽입한다. insert_min_heap 함수는 간선을 삽입하면서 최소 힙 구조를 유지하므로, 각 삽입의 시간 복잡도는 O(logN)이다. 총 N개의 간선을 삽입하므로 간선을 최소 힙에 삽입하는 과정은 O(NlogN)의 시간 복잡도를 갖는다.

- 그 다음 힙에서 간선 중 가장 작은 가중치를 가진 간선을 하나씩 추출한다. delete_min_heap 함수는 루트 노드를 제거한 후 힙 구조를 재조정하므로 각 연산의 시간 복잡도는 O(logN)이다. 최악의 경우, 모든 간선을 처리해야 하므로 이 작업은 최대 N번 수행된다. 총 N개의 간선을 제거하므로 O(NlogN)의 시간 복잡도를 갖는다.

- 추출한 간선에 대해 두 정점의 루트를 찾고, 루트가 다를 경우 두 컴포넌트를 병합한다. Find와 Union의 연산은 경로 압축과 rank 기반 병합을 사용하여 시간 복잡도가 O(a(N))으로 최적화된다. 간선은 최대 N – 1번 MST에 포함될 수 있으므로 Find와 Union 연산이 총 N – 1번 호출된다. 각 간선에서 최대 두 번의 Find와 한 번의 Union이 호출되므로 O(N*a(N))의 시간 복잡도르 갖는다.

- Kruskal 알고리즘의 전체 시간 복잡도는 가장 큰 시간 복잡도를 가지는 작업에 의해 지배된다. 그래프에서 간선의 수 E와 정점의 수 V 사이의 관계는 보통 E >> V이므로 O(ElogE)와 O(E*a(V))가 주요 복잡도를 결정한다. 여기서 O(ElogE)가 가장 지배적이므로 일반적으로 Kruskal 알고리즘의 최종 시간 복잡도는 O(ElogE)으로 간주된다. 또한 E는 최악의 경우 $O(V^2)$이다. 이 경우 $\log E \approx \log(V^2) = 2\log V$ 이므로 $O(E\log E) \approx O(E\log V)$가 된다. 따라서 Kruskal 알고리즘의 최종 시간 복잡도는 O(ElogV)라고 볼 수 있다.

2. **Actual Running Time**

파일 이름	작동 여부	MST weight	수행 시간(초)	k_scanned
HW3_com-amazon.ungraph.txt	YES	2,729,670,156	1.654	925,855
HW3_com-dblp.ungraph.txt	YES	2,747,895,457	1.878	1,049,834
HW3_com-lj.ungraph.txt	YES	28,308,045,762	106.399	34,681,165
HW3_com-youtube.ungraph.txt	YES	14,578,691,475	6.364	2,987,623

[이론적인 시간 복잡도와의 비교 분석]

파일 이름	E	$\log_2 V$	$E\log_2 V$	이론적 시간(비례값)
HW3_com-amazon.ungraph.txt	925,872	18.36	16,994,003	16.99M
HW3_com-dblp.ungraph.txt	1,049,866	18.28	19,178,457	19.18M
HW3_com-lj.ungraph.txt	34,681,189	21.96	761,728,631	761.72M
HW3_com-youtube.ungraph.txt	2,987,624	20.11	60,108,505	60.11M

- 로그 값은 log2를 기준으로 계산하며, 이를 통해 각 데이터의 이론적 시간 복잡도 O(ElogV)를 추정한다.

- 이론적 시간에서 M은 연산 단위 Million을 나타낸다. 즉, 16.99M은 초당 16,990,000번의 연산을 처리한다는 의미이다.

파일 이름	이론적 시간(비례값)	수행 시간(초)	비례 계수(이론/실제)
HW3_com-amazon.ungraph.txt	16.99M	1.654	16.99M/1.654 ≈ 10.27M/s
HW3_com-dblp.ungraph.txt	19.18M	1.878	19.18M/1.878 ≈ 10.22M/s
HW3_com-lj.ungraph.txt	761.72M	106.399	761.72M/106.399 ≈ 7.16M/s
HW3_com-youtube.ungraph.txt	60.11M	6.364	60.11M/6.364 ≈ 9.44M/s

- 작은 데이터셋(amazon, dblp 파일)의 경우, 실제 수행 시간은 이론적 시간에 비례하며 비례 계수는 약 10M/s로 일정한 것을 확인할 수 있다.

- 중간 데이터셋(youtube 파일)은 비례 계수가 약 9.44M/s로, 이론과 실제 비율이 유지되고 있다.

- 반면 큰 데이터셋(lj 파일)의 경우, 비례 계수가 약 7.16M/s로 낮아졌다. 이는 대규모 데이터셋에서 힙 연산이나 메모리 접근 속도가 병목이 되었음을 암시한다. 즉, 간선 수가 매우 크기 때문에 최소 힙 연산이 더 많은 캐시 미스를 발생시켰을 가능성이 크다.

- 따라서 소규모와 중간 규모 데이터셋에서는 이론적인 시간 복잡도를 어느 정도 반영한다고 할 수 있지만, 대규모 데이터셋에서는 캐시 미스와 메모리 접근 속도가 시간 복잡도에 큰 영향을 미칠 수 있으며 이로 인해 이론적 시간 대비 실제 수행 시간이 더 느려졌음을 알 수 있다.