*이름 : 이예준*

*학번 : 20212022*

*알고리즘설계와분석.HW5*

*Minimum Spanning Tree 알고리즘의 구현*

**1. 목적**

알고리즘 설계 기법 중 하나인 Greedy Approach를 이해하고, Kruskal의 MST 알고리즘을 구현한 뒤

이론적 시간 복잡도와 실제 수행 시간의 관계를 분석하며, 대용량 그래프 처리를 위한

효율적인 자료 구조 설계와 구현을 연습한다.

**2. 구현 방법**

본 코드는 Kruskal 알고리즘을 이용해 MST를 구하는 코드이다. Kruskal 알고리즘은 모든

간선을 가중치 순으로 정렬한 뒤, 가장 가중치가 작은 간선부터 트리에 추가해 나가되,

서로 다른 connected component에 속한 정점들을 연결할 때만 그 간선을 사용한다.

이를 위해 보통 다음 자료구조를 사용한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명



1. 간선 정렬용 자료구조

- 일반적으로는 간선을 배열 등에 담은 후 정렬(sort)을 수행하거나, 우선순위 큐(MinHeap)을

사용한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

2. 서로소 집합(Disjoint Set, Union-Find)

- 정점들이 현재 어느 집합(트리 혹은 포레스트)에 속해 있는지 추적하고, 집합 병합 연산을

빠르게 수행하기 위해 사용한다.

이 코드에서는 MinHeap 클래스를 직접 구현해, 간선을 가중치 오름차순으로 하나씩 꺼낸 뒤

DisjointSet(Union-Find) 구조를 통해 간선 추가 여부를 판단한다.

**- 시간 복잡도 분석**

1. 전체 알고리즘의 흐름

**1-1. edge 입력**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

- 파일에서 간선 정보를 읽어 `edges` 벡터에 저장한다.

- 해당 과정은 E 개의 간선을 읽으므로, 단순히 파일 I/O에 대한 반복으로

대략 O(E)시간에 이뤄진다고 볼 수 있다.

**1-2. MinHeap에 간선 삽입**

텍스트, 폰트, 스크린샷, 번호이(가) 표시된 사진

자동 생성된 설명

- E개의 간선을 힙에 삽입한다. MinHeap의 삽입 연산(`heapifyUp`)은 평균적으로

O(log E)이므로, 모든 간선을 삽입하는 데 걸리는 시간은 총 O(E log E)이다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**1-3. Kruskal 알고리즘 메인 루프**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

- MinHeap에서 가장 가중치가 낮은 간선을 `pop`하여 꺼낸다( O(log E)시간 ).

- `DisjointSet.find(u)` / `DisjointSet.find(v)` 호출을 통해 두 정점이 다른 집합에

속해 있는지 확인한다. Union-Find는 경로 압축(Path Compression)과 랭크(Rank)를

사용하므로, 거의 O(1)에 가깝게 동작한다( alpha(V), 매우 작은 상수로 취급).

- 서로 다른 집합이라면, `unionSets(u, v)` 연산을 수행한다. 이 또한 O(1)시간이다.

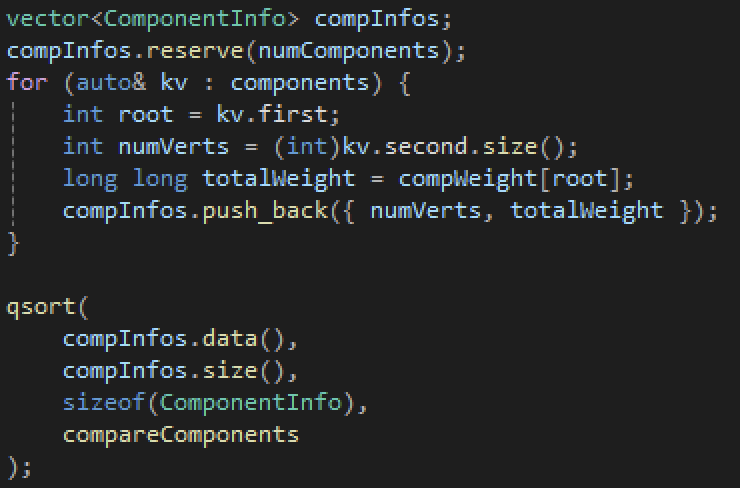
따라서, (간선 하나를 꺼낼 때(pop) : O(log E) ) +( Union-Find 작업 O(1) ) = O(log E)이다.

최대 E번까지 pop할 수 있으므로 전체가 O(E log E)가 된다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**1-4. 결과 후처리**

****

- MST(혹은 MSF)에 포함된 간선을 다시 이용하여 각 연결 요소(Component)의 정보를

구한다.

- 연결 요소 개수가 C라고 했을 때, 결과 정렬(`qsort`)은 O(C log C)가 소요된다.

보통 ( C ≤ V )이므로, 간선 개수 E가 정점 수 V보다 훨씬 많을 경우,

전체 시간 복잡도는 여전히 O(E log E)가 지배적이다.

**2. 코드상 핵심 포인트**

**2-1. MinHeap의 삽입(`push`)과 삭제(`pop`)**

- `heapifyUp()`과 `heapifyDown()`이 각각 O( log E) 시간이다.

- 전체 간선 E개에 대해 삽입과 삭제가 일어나므로, O(E log E)가 된다.

**2-2. DisjointSet(Union-Find)의 경로 압축**

- `find()`가 O(alpha(V))시간이며, alpha는 매우 작은 함수(실무적으로 거의 상수)이다.

**2-3. 정렬(`qsort`)로 컴포넌트 정보 출력**

- 컴포넌트 수 (C ≤ V)이므로, 이는 O(V log V) 정도이다.

- 만약 ( E >> V )인 상황이라면, MST 알고리즘 전체 시간은 결국 O(E log E)로 결정된다.

**3. 결론**

결과적으로 이 코드는 O(E log E) 시간 복잡도를 가진다.

**3. 실험 결과**

4개의 서로 다른 파일을 가지고 본인의 코드를 실행한 결과 아래와 같은 결과들을

얻을 수 있었다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **파일 이름** | **작동여부** | **MST Weight** | **수행시간(초)** | **Kscanned** |
| HW3\_com-amazon.ungraph.txt | YES | 2,729,670,156 | 0.30 | 334,862 |
| HW3\_com-dblp.ungraph.txt | YES | 2,747,895,457 | 0.38 | 317,079 |
| HW3\_com-youtube.ungraph.txt | YES | 14,578,691,475 | 2.13 | 1,134,889 |
| HW3\_com-lj.ungraph.txt | YES | 28,308,045,762 | 54.7 | 3,997,961 |

\*MST weight : 가장 꼭지점이 많은 컴포넌트에 대한 minimum spanning tree의 결과 weight

\*수행 시간 (초) : 파일 입출력 시간을 제외한 minimum spanning tree 구축에 소요된

순수한 수행 시간

\*kscanned : minimum spanning tree의 구축이 완성될 때까지 처리한 edge의 개수

**4. 이론적인 시간 복잡도 반영 여부**

실제로 O(E log V)에 비춰보았을 때,

- 파일 1 to 파일 2: 정점 수는 조금 줄었지만 간선 수는 약간 증가.

수행시간도 0.30에서 0.38초로 증가.

- 파일 2 to 파일 3: 정점과 간선이 각각 약 3~4배, 2~3배 늘어나면서 수행시간은

0.38에서 2.13초로 (약 5~6배 증가).

- 파일 3 to 파일 4 : 정점이 ~3.5배, 간선이 ~11.6배 늘어나면서 수행시간은

2.13에서 54.7초로 (약 25~26배 증가).

완전히 이상적인 O(E log V) 스케일링을 정확히 따라간다고 보긴 어렵지만,

정점/간선이 크게 늘어날수록(특히 간선이 대폭 늘어날수록) 수행 시간이 비약적으로

증가한다는 점에서 큰 틀의 “많은 간선 🡪 더 많은 연산”이라는 MST 복잡도의 흐름은

전반적으로 반영되어 있다고 볼 수 있다.

**\*실제 측정 시간이 이론적 복잡도와 다른 이유**

**I/O 및 환경 제약**

-현재 본인의 코드를 Mac Air 노트북에서 가상머신으로 돌렸기 때문에

테스트 환경(메모리 용량, CPU 코어 수, 운영체제, 컴파일러 최적화 등)에 따라

실제 시간이 크게 달라졌다고 예상된다.

**5. Setting**

-실험환경

OS: macOS Sonoma 14.3.1

CPU: Apple M1

RAM: 8.00GB

-가상머신 실험환경

OS: Windows 11 Home

CPU: virt-7.2 1GHz

RAM: 4.00GB

Compiler: Visual Studio 22 Release Mode/ARM64 Platform