MST(Minimun Spanning Tree)

정의

최소 신장 트리

- 그래프 G의 subgraph중 Tree이며(connected && acyclid && E = V-1)
- Spanning : 모든 정점을 포함하는
- Minimum : Spanning Tree 중 weight의 합이 최소인 Tree

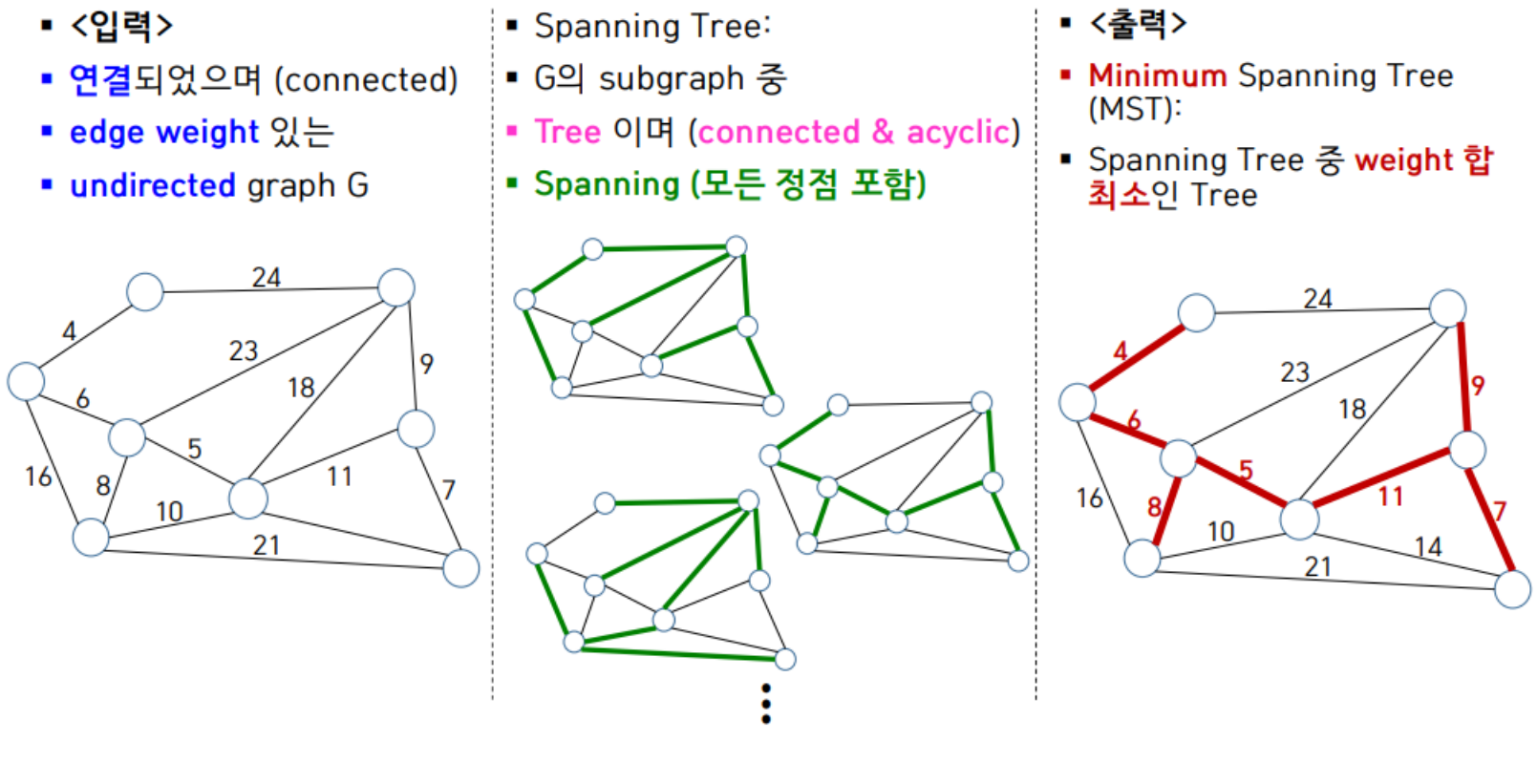
정리 : 신장 트리가 그래프의 모든 정점을 최소의 비용으로 연결한다는 의미.

Spanning Tree?

- `무향 연결 그래프`가 있을 때 그 그래프에서 간선을 부분적으로 뽑아서 만들 수 있는, 그래프의 정점 개수와 같은 정점 개수를 가지는 트리

특징

- 간선의 수가 가장 적다 : 정점의 수가 n개 일 때, spanning Tree의 간선 수는 언제나 (n-1)이다.
- 사이클이 발생하면 안된다. (사이클이 있다는 것 자체가 최소로 연결되어 있지 않기 때문!!)
  - Ex) 1-2-3으로 연결 되어 있다면, 1과 3은 직접적으로 연결되어 있는 건 아니더라도 연결된 것으로 본다. (통행이 가능하기 때문), 따라서 사이클이 형성하도록 1-3을 연결할 필요는 없다
- DFS, BFS를 사용하여 그래프에서 신장 트리를 찾을 수 있다.
- 한 그래프에서 스패닝 트리는 여러개 일 수 있다.
- MST또한 여러개 나올 `수` 있다
- MST는 최소 가중치를 갖는 crossing Edge는 반드시 MST에 포함해야된다.



왜 Tree인가? (연결 && 사이클이 없음)

- 트리는 포한된 V개의 정점을 `최소 수의 간선 사용`해 모두 연결하는 구조
- 따라서 트리에서 `하나의 간선만 제거해도 비연결`
- `하나의 간선만 추가해도 사이클이 발생` (두 정점 간에 둘 이상의 여분의 경로가 생김)
- `결론` : 최소 간선수를 사용해서 연결 비용을 절감하거나 사이클을 막기 위해 사용

언제 MST?

- 연결 자원을 가능한 적게 쓰며(간선의 합 최소) 모든 지점을 연결하게 할 때(Spanning)

MST의 이모저모

1. 그래프 G와 최소 신장 트리 T가 주어졌을 때, T에서 간선 중 하나의 가중치를 줄여도 T가 여전히 G에 대한 최소 신장 트리이다
2. 그래프 G와 최소 신장 트리 T가 주어졌을 때 T에 있지 않은 간선 중 하나의 가중치를 줄인다고 가정을 한다면. 수정한 그래프에서 최소 신장 트리를 찾는 알고리즘은?
   - T에 줄어든 Edge를 더한다. 그렇게 된다면 T는 사이클을 형성하게 된다(Tree는 순환이 없는 V-1개의 Edge를 가지기 때문이다). 거기서 DFS를 사용해서 Cycle이 존재하는 부분에서 가장 긴 Edge를 빼면 된다.

최소 신장 트리의 확장(safe edge)

최소 신장 트리를 구하는 방법은 `kruskal algorithm`과 `prim algorithm`으로 나뉜다.
이 방법들은 `greedy`방식을 기반으로 한다.

2가지 방식은 `safe edge(안전간선)`을 어떤 방식으로 찾는가에 차이점을 가진다.
안전 간선이란 MST의 특성(acyclic, minimum weight)를 지키면서 MST의 부분 트리에 선택할 수 있는 간선을 의미한다.

```
## GENERIC-MST Algorithm

1. Initialize:
   `A = ∅`

2. Loop:
   `while A does not form a spanning tree`

3. Find Safe Edge:
   Find an edge `(u, v)` that is safe for `A`

4. Add Edge to A:
   `A = A ∪ {(u, v)}`

5. Return:
   Return `A`
```

앞서 살펴본 바와 같이, MST를 찾는 알고리즘은 `safe edge`를 찾는 방식으로 나누어진다.

Kruskal
크루스칼 알고리즘에서는 집합 A는 `Forest(tree의 집합)`을 의미한다.
A에 더해지는 안전 간선은
두 개의 독립된 원소를 연결하는 최소 가중치의 간선을 의미한다.

Prim
프림 알고리즘에서는 집합 A는 단일한 Tree를 형성한다.
A에 추가되는 safe edge는 트리안에 있지 않는 vertex를 연결하는 가중치가 낮은 edge를 의미한다.

Kruskal Algorithm

의사 코드

```
## MST-KRUSKAL Algorithm

1. **Initialize:**
   `A = ∅`

2. **Make Sets for Vertices:**
   `for each vertex v ∈ G.V`
   - `MAKE-SET(v)`

3. **List of Edges:**
   Create a single list of the edges in `G.E`.

4. **Sort Edges:**
   Sort the list of edges into monotonically increasing order by weight `w`.

5. **Process Each Edge:**
   `for each edge (u, v)` taken from the sorted list in order:
   - **Check Sets:**
     `if FIND-SET(u) ≠ FIND-SET(v)`
   - **Add to A:**
     `A = A ∪ {(u, v)}`
   - **Union:**
     `UNION(u, v)`

6. **Return MST:**
   Return `A`.
```

kruskal-algorithm의 분석

1. 집합 A를 공집합으로 초기화 하고, V개의 트리를 생성한다. (각 트리는 각 정점으로 만들어진다.)

2. for루프는 간선을 가중치가 증가하는 순서로 조사한다.
    - u와 v가 같은 트리에 속하는지 확인한다.
        - u와 v가 같은 트리에 속하면 -> 간선을 포기한다(순환을 만들지 않으면선 간선 u,v를 A에 추가할 수 없으므로)
        - u와 v가 같은 트리에 속하지 않는다면 -> 간선 (u,v)를 A에 추가한다. (두 정점은 서로 다른 트리에 속한다.)

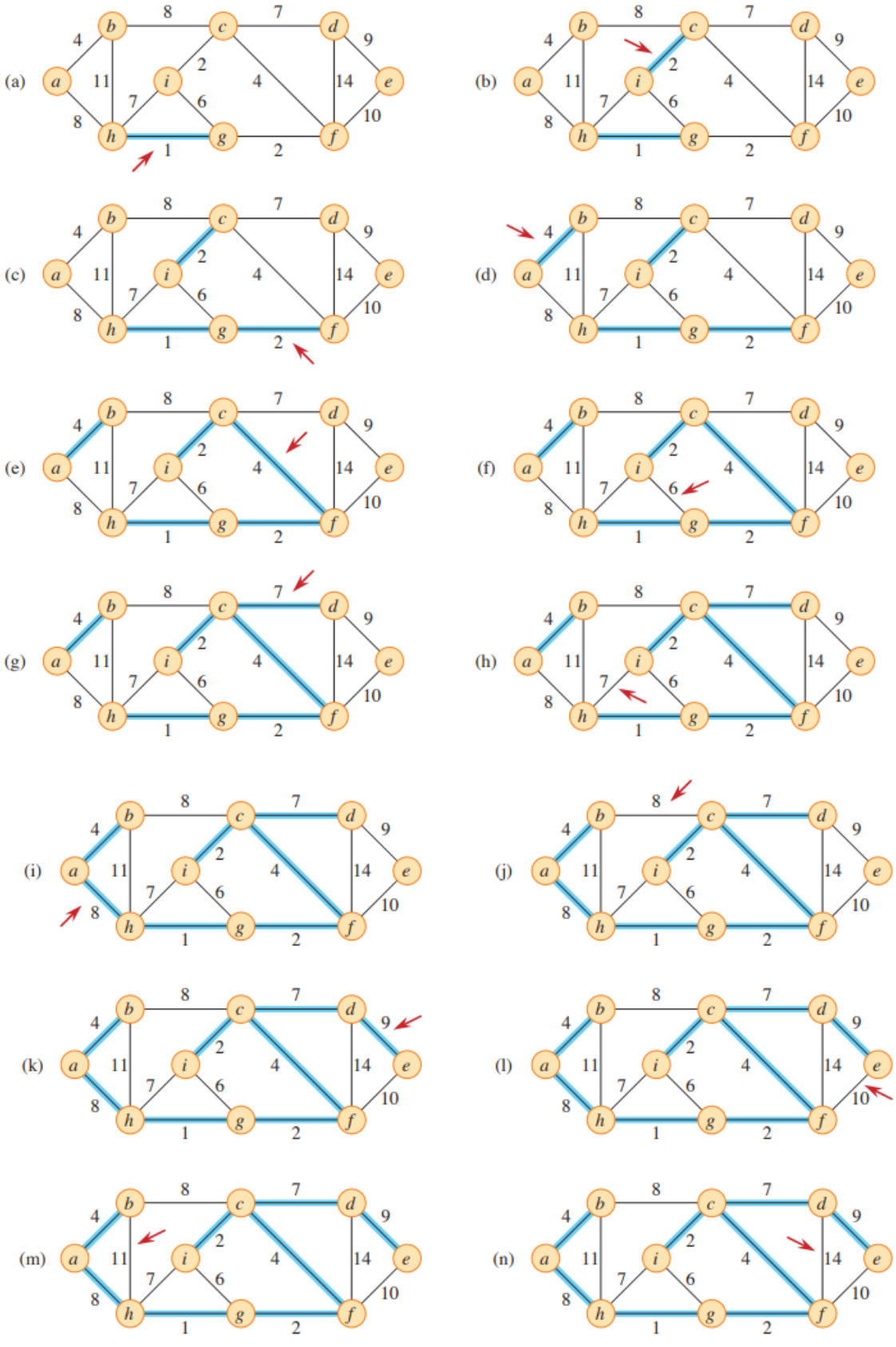
같은 집합에 포함되는 것을 어떻게 알 수 있나요?
`kruskal algorithm`을 살펴보면, "같은 트리에 속하면"이라는 말이 나온다.
즉 같은 집합에 있는 지를 빠르게 파악하기 위해서는 `Union-Find` 자료구조가 필요하다.
`Union-Find`를 경로 압축과, weight-by-rank를 사용해서 최적화를 하면 상수 시간에 근접하게 알아낼 수 있다.

Kruskal Algorithm의 시간복잡도

- Edge를 weight를 기준으로 정렬하는데 `Elog(E)`
- for 반복문을 `E`만큼 순회함
- union-find의 parent를 초기화 하는데 필요한 `V`

결론적으로 전체 알고리즘의 수행시간은 : `Elog(E)`

Kruskal Algorithm의 예시 그림



Kruskal Algorithm Code

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
```

```
class Edge
{
public:
    int u;
    int v;
    int w;

    Edge(const int& u_input, const int& v_input, const int& w_input) : u(u_input), v(v_input), w(w_input) {};
    bool operator < (const Edge& other) const
    {
        return this -> w < other.w;
    }

    ostream& operator << (ostream& out)
    {
        out << this-> u << " " << this->v << " " << this->w;
        return out;
    }
};

int V, E;
vector<vector<pair<int,int>>> adj_list; // {to, weight}
vector<Edge> Edges;
vector<int> parents;
vector<int> weights;
```

```
int Find(const int& a)
{
    if (parents[a] == a) return (a);
    else
        return parents[a] = Find(parents[a]);
}

void Union(int a, int b)
{
    a = Find(a); b = Find(b);
    if (a == b) return;

    else if (weights[a] <= weights[b])
    {
        parents[a] = b;
        weights[b] += weights[a];
    }
    else
    {
        parents[b] = a;
        weights[a] += weights[b];
    }
}



void Init()
{
```

```
    int u,v,w;
    cout << "input the numbers of Edge and Vertex : ";
    cin >> E >> V;
    adj_list.resize(V+1);
    parents.resize(V+1);
    weights.resize(V+1, 1);
    for(size_t i = 0; i <=V; ++i)
        parents[i] = i;
    cout << "input u v w : ";
    for(size_t i= 0; i < E; ++i)
    {
        cout << "u v w : "; cin >> u >> v >> w;
        adj_list[u].emplace_back(make_pair(v,w));
        adj_list[v].emplace_back(make_pair(u,w));
        Edges.emplace_back(u,v,w);
    }
}

void Kruskal()
{
    sort(Edges.begin(), Edges.end()); // 간선들을 가중치 순으로 정렬
    int mst_weight = 0;
    vector<Edge> mst; // 최소 신장 트리에 포함되는 간선들

    for (const auto& edge : Edges)
    {
```

```
        int u = edge.u;
        int v = edge.v;
        int w = edge.w;

        if (Find(u) != Find(v)) // 사이클을 형성하지 않는다면
        {
            Union(u, v); // 두 정점을 연결
            mst.push_back(edge); // 최소 신장 트리에 간선 추가
            mst_weight += w; // MST의 총 가중치 갱신
        }
    }
   // 결과 출력
    cout << "Minimum Spanning Tree Weight: " << mst_weight << endl;
    cout << "Edges in the MST:" << endl;
    for (const auto& edge : mst)
    {
      cout << edge.u << " - " << edge.v << " : " << edge.w << endl;
    }
}

int main(void)
{
    ios::sync_with_stdio(0);
    cin.tie(0); cout.tie(0);
    Init(); // 입력 초기화
    Kruskal(); // 크루스칼 알고리즘 수행

    return (0);
}
```

Prim's Algorithm

프림 알고리즘은 `다익스트라 알고리즘`과 매우 유사하게 동작한다.
프림 알고리즘은 집합 A의 간선이 항상 하나의 `트리`를 이루는 특성을 가지고 있다.
트리는 임의의 루트 정점 r로부터 시작해 그 트리가 V에 있는 모든 정점을 포함할 때까지 자라게 된다.
각 단계는 트리 A를 고립된 정점에 연결하는 경량 간선을 A에 추가한다.

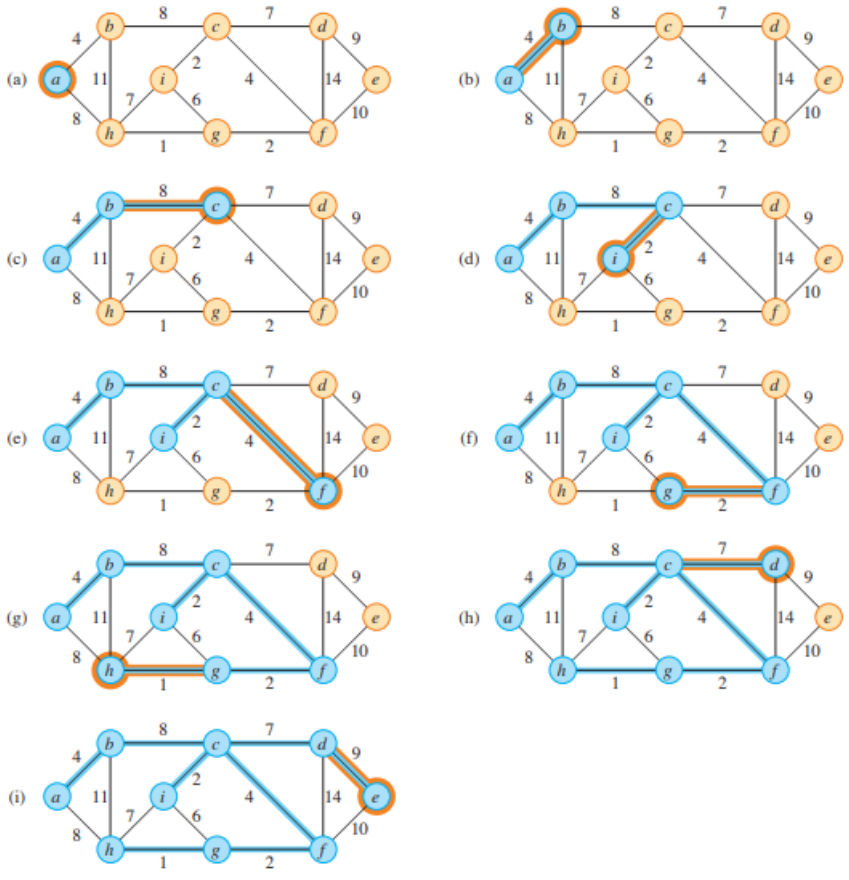
프림 알고리즘의 시간 복잡도는 `O(ElgV)`이다.
만약 우선순위 큐의 구현을 피보나치 힙을 사용하게 된다면 `O(E + VlgV`로 개선된다.

Prim Algorithm의 의사코드

```
MST-PRIM(G, w, r)
 1  for each vertex u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = ∅
 6  for each vertex u ∈ G.V
 7      INSERT(Q, u)
 8  while Q ≠ ∅
 9      u = EXTRACT-MIN(Q)          // add u to the tree
10      for each vertex v in G.Adj[u]   // update keys of u's non-tree neighbors
11          if v ∈ Q and w(u, v) < v.key
12              v.π = u
13              v.key = w(u, v)
14              DECREASE-KEY(Q, v, w(u, v))
```

```
아무 간선을 포함하지 않은 상태에서 시작 (MST = [])
시작 정점은 MST에 포함된 상태라고 봄
MST와 나머지 정점 연결하는 간선 중
        weight 가장 작은 간선을 MST에 추가하는 것을 반복
총 V-1개의 간선이 포함되면 종료
```

Prim Algortith의 예시 그림



Prim Algorithm 예시 코드

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <functional>

using namespace std;

// 그래프의 간선 표현
typedef pair<int, int> Edge; // (간선의 가중치, 목적지 정점)

// 프림 알고리즘 함수
int prim(int start, vector<vector<Edge>>& graph) {
    int V = graph.size();  // 정점의 개수
    vector<bool> visited(V, false);  // 방문 여부를 저장하는 배열
    priority_queue<Edge, vector<Edge>, greater<Edge>> pq;  // 최소 힙 (우선순위 큐)
    int mst_cost = 0;  // 최소 신장 트리의 총 가중치

    // 시작 정점에서 출발
    pq.push({0, start});  // 가중치 0, 시작 정점

    while (!pq.empty()) {
        int weight = pq.top().first;  // 현재 간선의 가중치
        int u = pq.top().second;  // 현재 간선의 목적지 정점
        pq.pop();

        // 이미 방문한 정점이라면 스킵
        if (visited[u]) continue;

        // 해당 정점을 방문 처리
        visited[u] = true;
        mst_cost += weight;  // 최소 신장 트리의 가중치 갱신

        // 현재 정점에 연결된 간선들을 확인
        for (auto& edge : graph[u]) {
            int v = edge.second;  // 연결된 정점
            int w = edge.first;  // 연결된 간선의 가중치
            if (!visited[v]) {
                pq.push({w, v});  // 방문하지 않은 정점이라면 우선순위 큐에 추가
            }
        }
    }

    return mst_cost;
}

int main() {
    // 정점의 수와 간선의 수
    int V = 6;  // 예시에서 6개의 정점
    int E = 9;  // 예시에서 9개의 간선
```

```
    // 그래프의 인접 리스트 표현 (예시 그래프)
    vector<vector<Edge>> graph(V);

    // 간선 추가 (시작 정점, 끝 정점, 가중치)
    graph[0].push_back({4, 1});
    graph[1].push_back({4, 0});

    graph[0].push_back({4, 2});
    graph[2].push_back({4, 0});

    graph[1].push_back({2, 2});
    graph[2].push_back({2, 1});

    graph[1].push_back({6, 3});
    graph[3].push_back({6, 1});

    graph[2].push_back({8, 3});
    graph[3].push_back({8, 2});

    graph[2].push_back({9, 4});
    graph[4].push_back({9, 2});

    graph[3].push_back({3, 4});
    graph[4].push_back({3, 3});

    graph[3].push_back({5, 5});
    graph[5].push_back({5, 3});

    graph[4].push_back({7, 5});
    graph[5].push_back({7, 4});

    // 시작 정점을 설정 (예시에서는 0번 정점)
    int start = 0;

    // 최소 신장 트리의 총 가중치 계산
    int mst_cost = prim(start, graph);
    cout << "최소 신장 트리의 총 가중치: " << mst_cost << endl;

    return 0;
}
```