최소 스패닝 트리
(Minimum Spannig Tree)

스패닝트리(Spanning Tree)

- 신장트리라고도 하며 그래프 내의 모든 정점을 포함하여 최소 연결된 부분 그래프 이다.

- 여기서의 최소 연결은 간선의 수가 가장 적다는 의미이다.

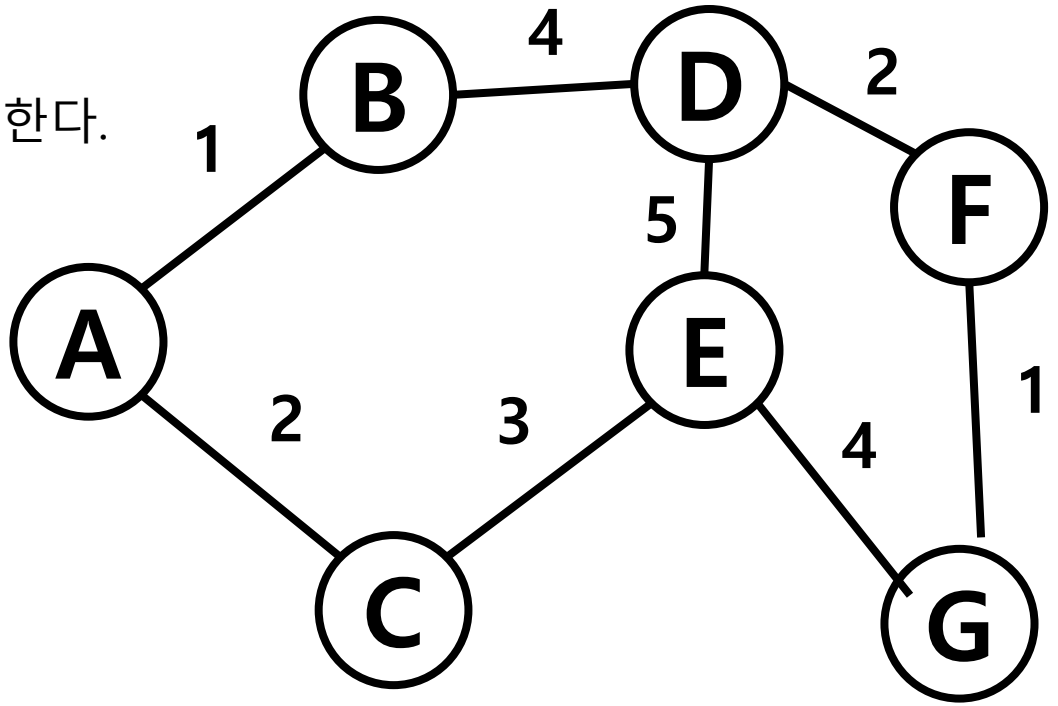
- DFS, BFS를 이용하여 그래프에서 신장 트리를 찾을 수 있다.
- 하나의 그래프는 많은 신장 트리가 존재 할 수 있다.
- 모든 정점들이 연결되어 있어야 하고 사이클이 포함해서는 안 된다.

최소 스패닝트리(MST)

- 최소 스패닝트리(MST)는 스패닝 트리의 조건을 모두 만족하여야 한다.
- 최소 스패닝트리(MST)는 간선에 가중치를 고려하여 최소 비용의 Spanning Tree를 선택하는 것을 말한다.
- 간선의 가중치의 합이 최소다.

최소 스패닝트리(MST)

오른쪽과 같이 정점이 있다고 있다.
각 정점을 연결하는데에는 비용이 든다.
여기서 비용은 A, B 를 연결하는데 드는 비용 1을 말한다.

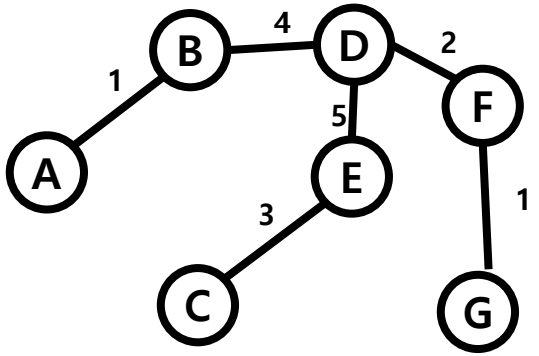


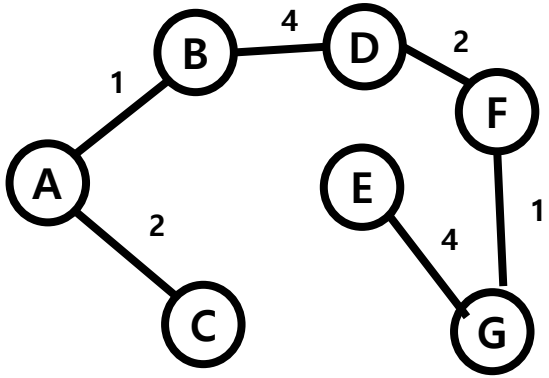
최소 스패닝트리(MST) 시간 복잡도

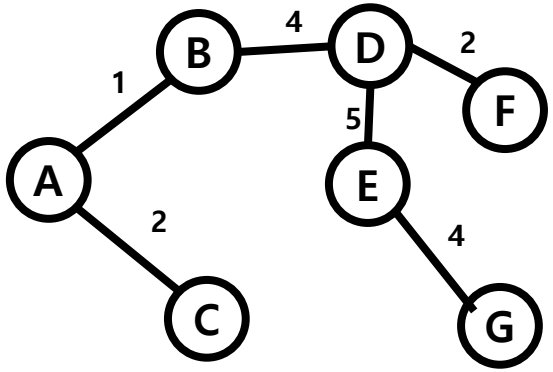
- 프림 알고리즘(Prim's algorithm): O(E log V)
  정점를 기준으로 탐색하는 알고리즘

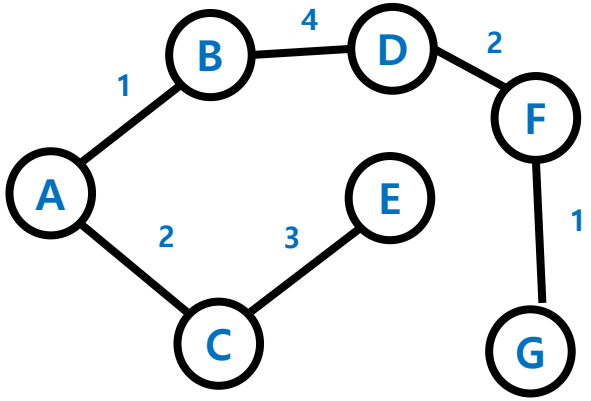
- 크루스칼 알고리즘(Kruskal algorithm): O(E*logE)
  간선을 기준으로 탐색하는 알고리즘

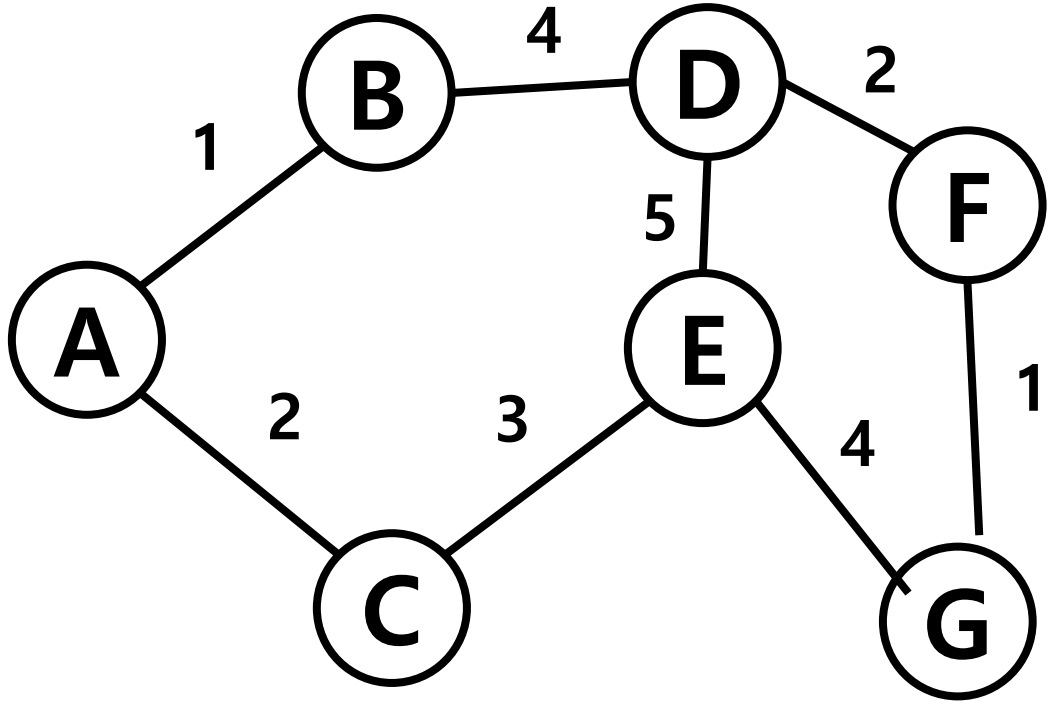
스패닝트리와 최소 스패닝트리(MST)





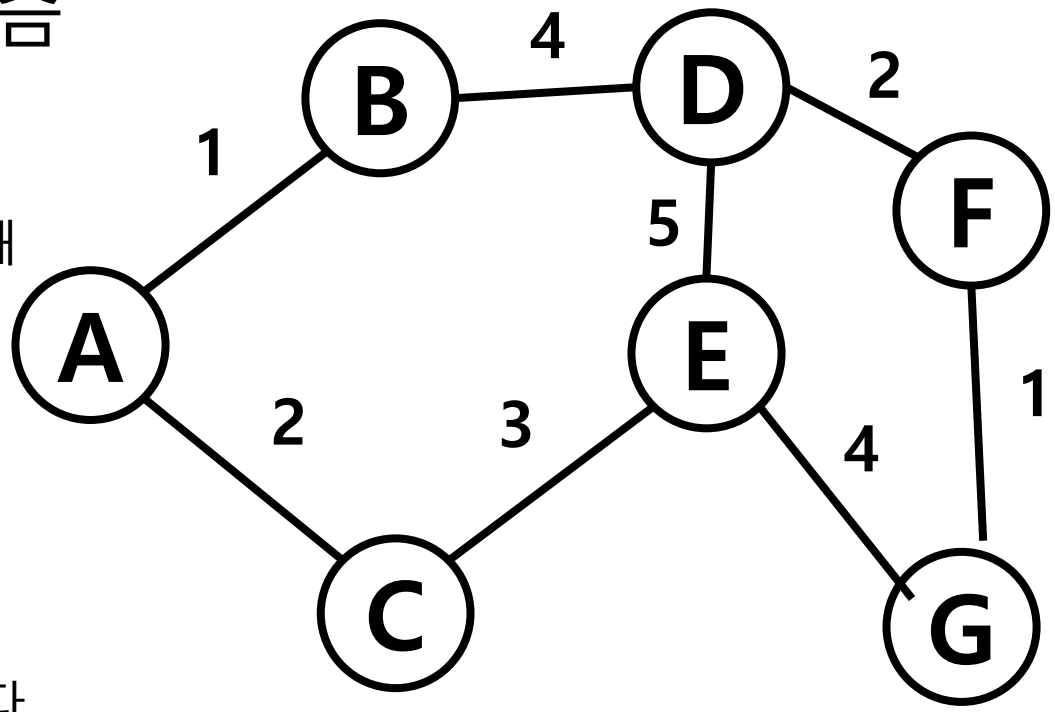






최소 스패닝트리(MST) 탐색 과정

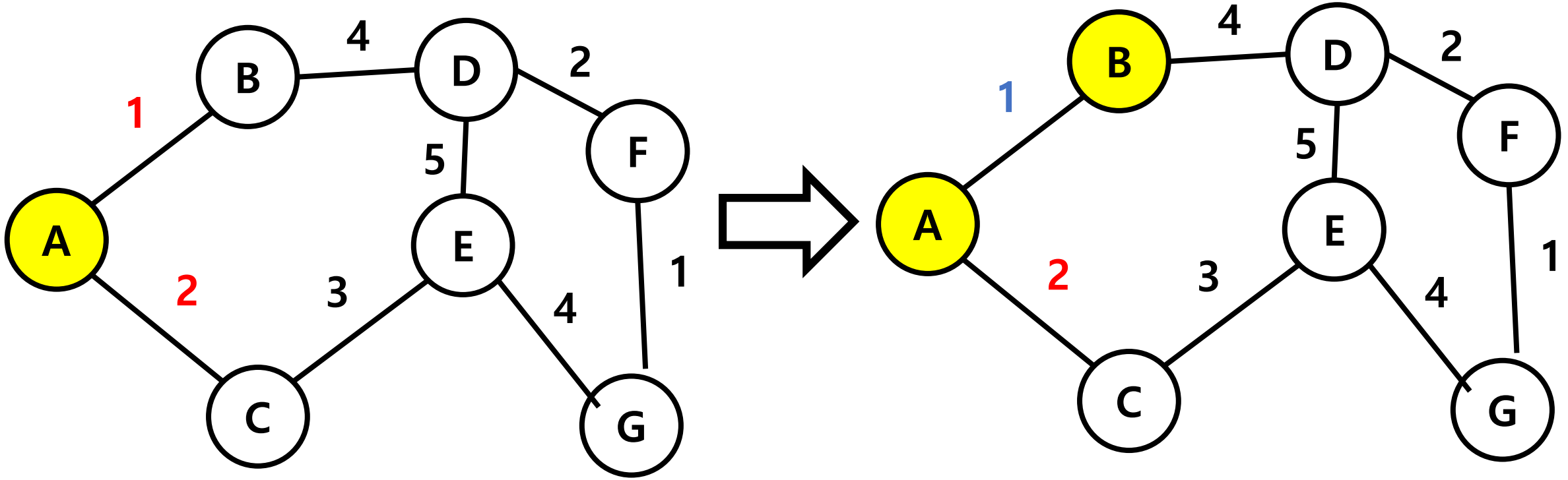
- 프림 알고리즘(Prim's algorithm): O(E log V)
  정점를 기준으로 탐색하는 알고리즘



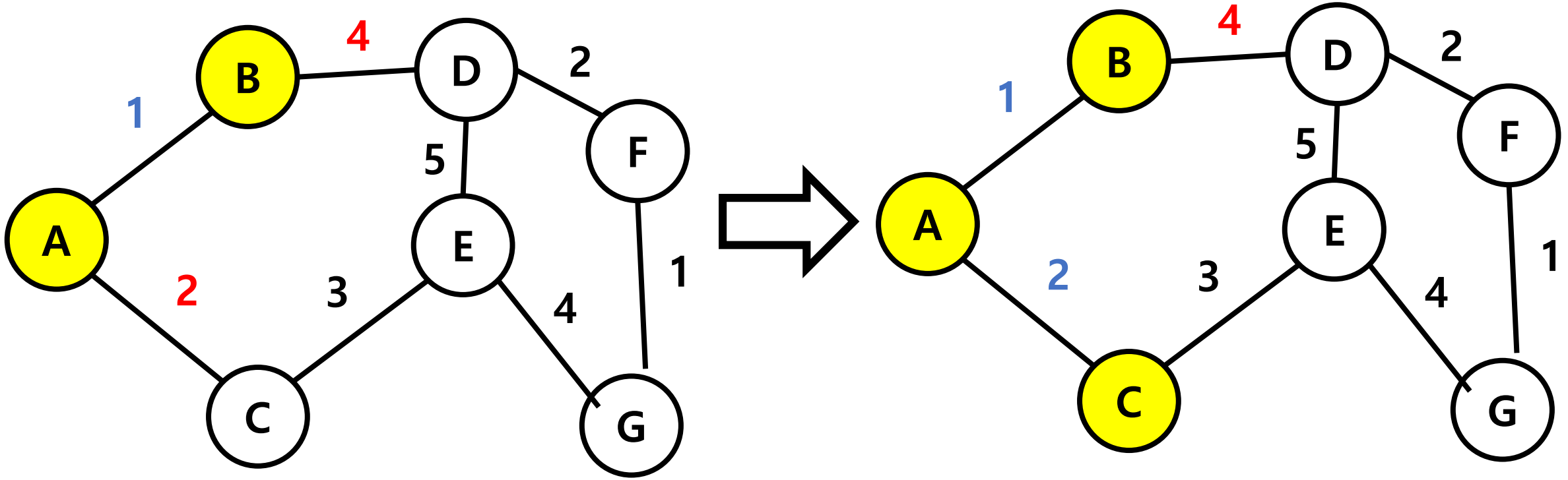
- 모든 정점을 연결해야 하기 때문에 어떤 정점부터 시작해도 상관이 없다.
- 방문체크를 해 방문한 곳은 또 방문하지 않도록 한다.
- 우선순위큐를 사용한다.(정점을 연결하는데 드는 비용을 오름차순 정렬)

1. 선택한 정점을 기준으로 인접한 정점에 드는 비용을 우선순위큐(PriorityQueue)에 삽입한다.
2. 우선순위큐(PriorityQueue)의 front 값의 정점과 연결한다.

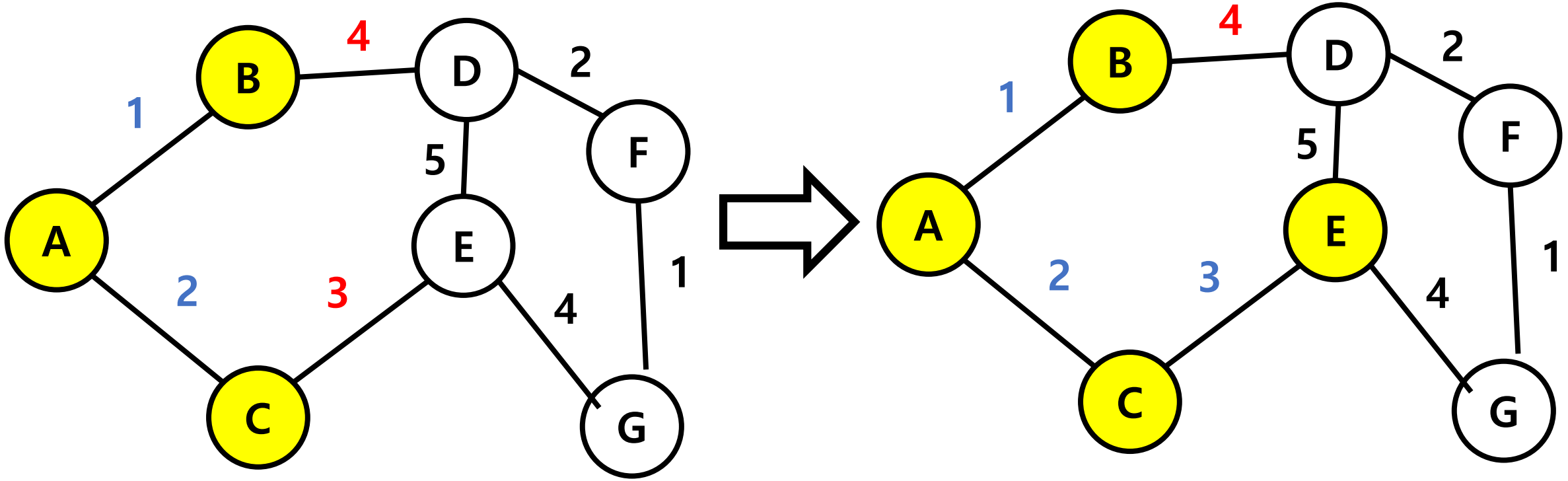
최소 스패닝트리(MST) 탐색 과정



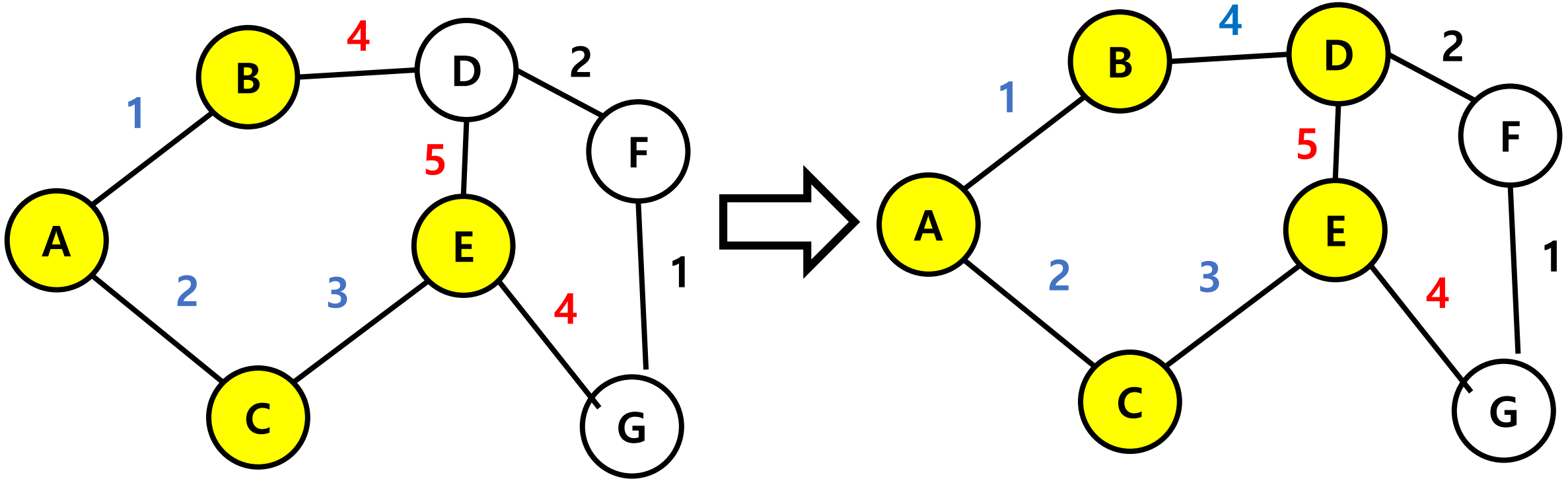
최소 스패닝트리(MST) 탐색 과정



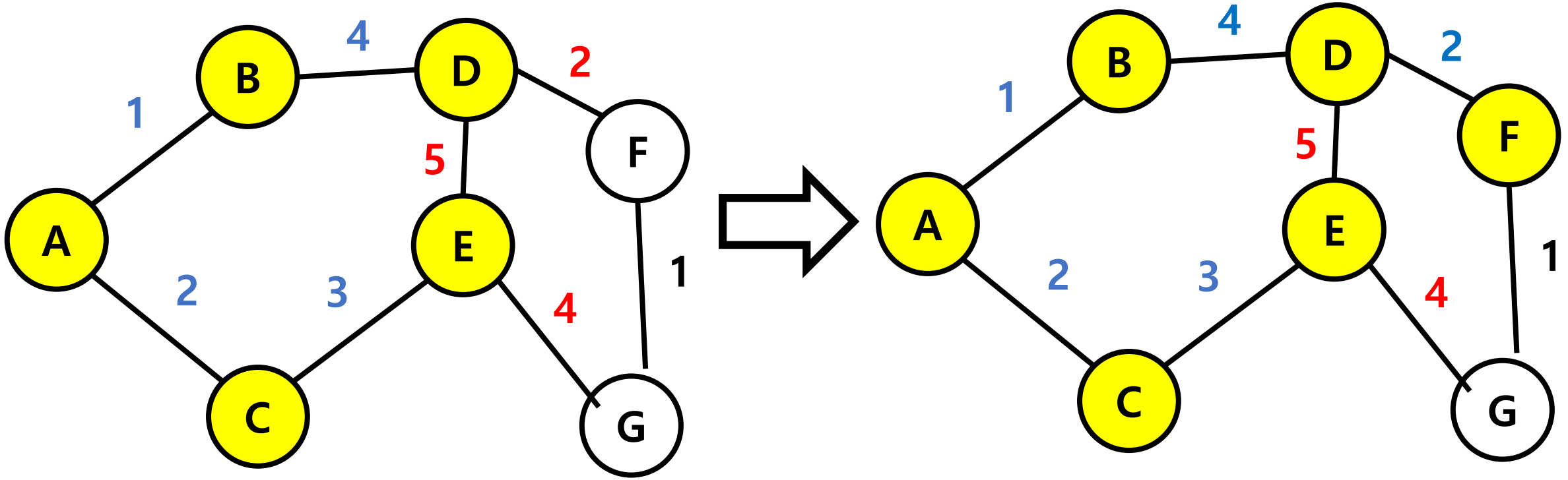
최소 스패닝트리(MST) 탐색 과정



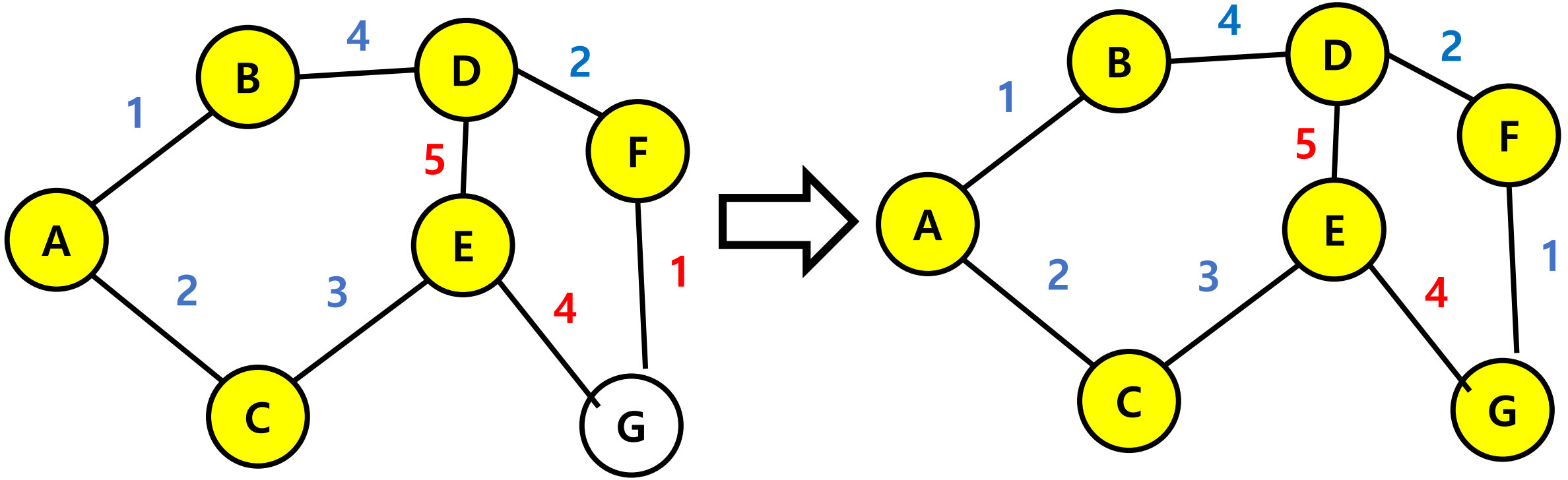
최소 스패닝트리(MST) 탐색 과정



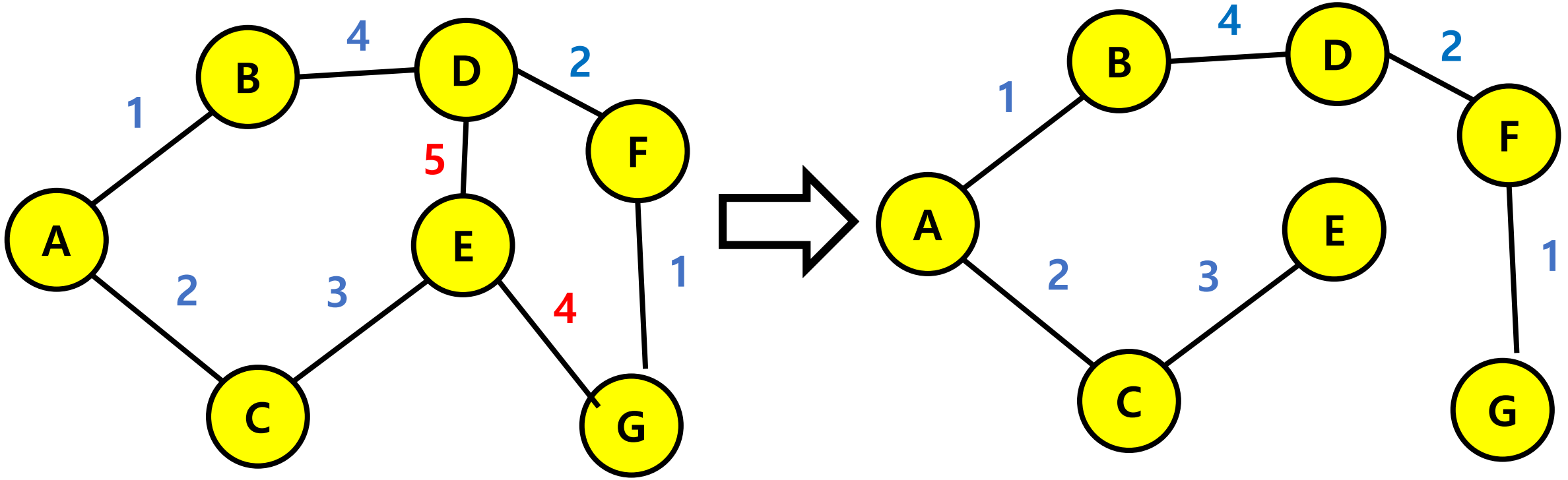
최소 스패닝트리(MST) 탐색 과정



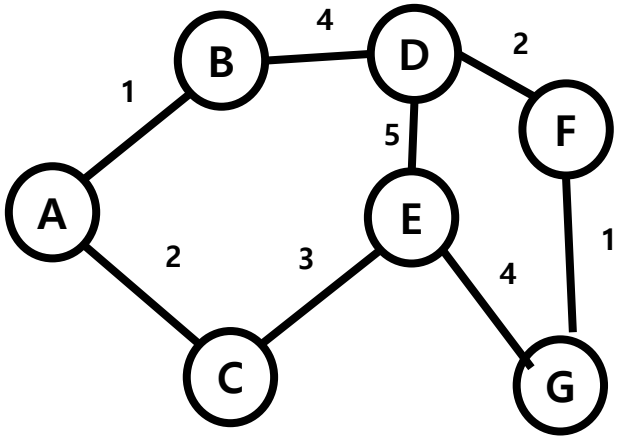
최소 스패닝트리(MST) 탐색 과정

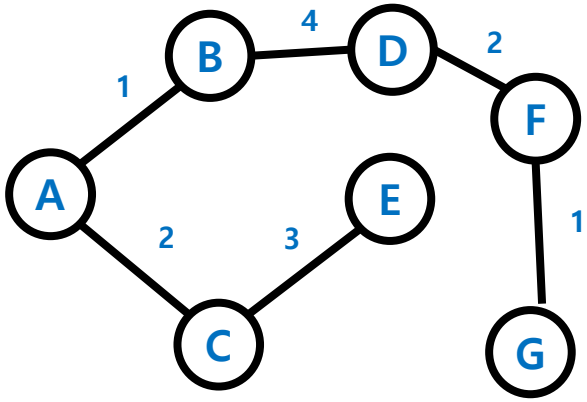


최소 스패닝트리(MST) 탐색 과정

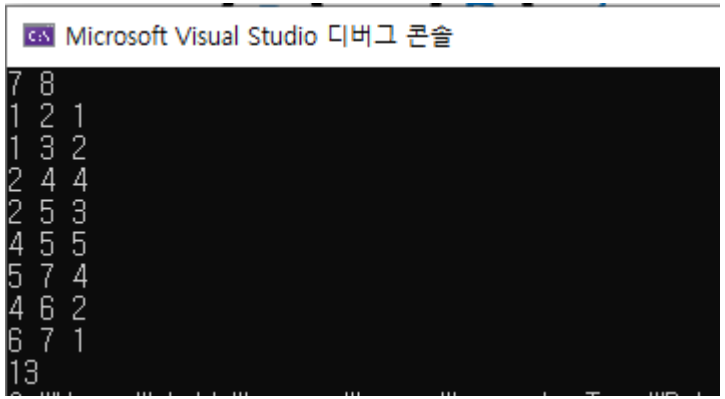


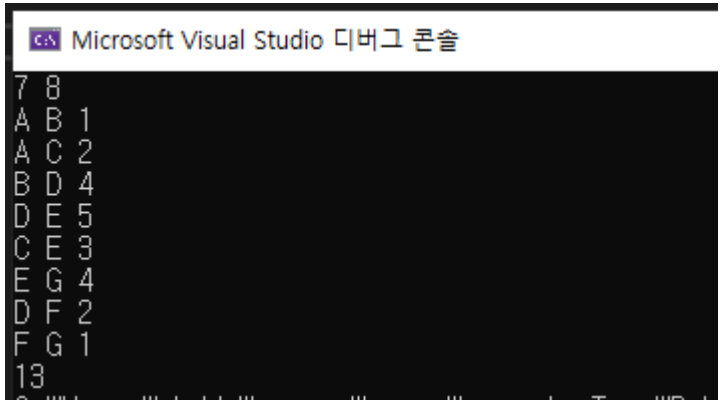
최소 스패닝트리(MST) 구현





입력
정점 개수(V) 간선 개수(E)
연결할 정점 그리고 두 정점을 연결하는데 드는 비용을
입력 받는다.





최소 스패닝트리(MST) 코드

```cpp
#include <iostream>
#include <queue>
#include<stack>
#include<vector>
#include<algorithm>

using namespace std;

// 정점에 대한 정보와 비용이 담긴 그래프
vector<pair<int, int>> graph[100001];

struct cmp
{
    bool operator()(pair<int, int> a, pair<int, int> b)
    {
        return a.second > b.second;
    }
};
```

```cpp
int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);

    // 정점 개수와 간선 개수
    int V, E;
    cin >> V >> E;

    // 초기 세팅
    for (int idx = 1; idx <= E; idx++)
    {
        int start, end, value;   cin >> start >> end >> value;
        graph[start].push_back({ end, value });
        graph[end].push_back({ start,value });
    }

    // 프림 알고리즘(Prim's algorithm)
    Prim();

    return 0;
}
```

최소 스패닝트리(MST) 코드

```cpp
#include <iostream>
#include <queue>
#include<stack>
#include<vector>
#include<algorithm>

using namespace std;

// 정점에 대한 정보와 비용이 담긴 그래프
vector<pair<int, int>> graph[100001];

struct cmp
{
    bool operator()(pair<int, int> a, pair<int, int> b)
    {
        return a.second > b.second;
    }
};
```

```cpp
int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);

    // 정점 개수와 노드 개수
    int V, E;
    cin >> V >> E;

    // 초기 세팅
    for (int idx = 1; idx <= E; idx++)
    {
        int start, end, value;   cin >> start >> end >> value;
        graph[start].push_back({ end, value });
        graph[end].push_back({ start,value });
    }

    // 프림 알고리즘(Prim's algorithm)
    Prim();

    return 0;
}
```

```cpp
void Prim()
{
    // 정점 방문체크용 배열
    vector<int> visted(10001);

    // 1을 초기위치로 설정
    int start = 1;

    // 초기 위치 세팅
    visted[start] = true;

    // 정점을 연결하는데 최소로 드는 비용
    int answer = 0;

    // 우선순위 큐 사용
    priority_queue<pair<int, int>, vector<pair<int, int>>, cmp> priority_queue;

    // 현재 정점의 인접한 정점 push
    for (int idx = 0; idx < graph[start].size(); idx++)
    {
        priority_queue.push({ graph[start][idx].first,graph[start][idx].second });
    }

    // 모든 정점을 방문할때까지 search
    while (!priority_queue.empty())
    {
        // 현재 정점
        int current_node = priority_queue.top().first;

        // 현재 정점으로부터 다음 정점까지 드는 비용
        int cost = priority_queue.top().second;

        priority_queue.pop();

        // 이미 방문한 정점이라면 continue
        if (visted[current_node] == true) continue;

        // 방문 체크
        visted[current_node] = true;

        // 비용 누적
        answer += cost;

        // 인접한 다음 노드 search
        for (int idx = 0; idx < graph[current_node].size(); idx++)
        {
            // 다음 정점
            int next = graph[current_node][idx].first;

            // 다음 정점으로부터 다다음 정점까지 드는 비용
            int next_cost = graph[current_node][idx].second;

            // 우선순위 큐 push(비용 기준으로 오름차순 자동 정렬)
            priority_queue.push({ next,next_cost });
        }
    }

    // 모든 정점을 연결하는데 사용한 비용
    cout << answer;
}
```