탐욕 알고리즘(Greedy Algorithm) – 21.03.01

매순간 최적이라고 생각되는 것을 선택해 나가는 방식으로 진행하여 최종적인 최적해에 도달하는 기법

탐욕 알고리즘이 잘 작동하는 문제는 **탐욕스런 선택 조건(greedy choice property)**와 **최적 부분 구조(optimal substructure)** 두 가지 속성을 만족합니다.

전자의 경우 **앞의 선택이 이후 선택에 영향을 주지 않는다는 걸 의미하고**, 후자는 **문제 전체에 대한 최적해(global optimum)가 부분문제에 대해서도 역시 최적해가 된다는** 걸 뜻합니다.

예컨대 분할가능 배낭문제(Fractional knapsack problem)가 대표적인 탐욕 알고리즘의 사례에 속합니다. 배낭문제는 한 여행가가 가지고 가는 배낭에 담을 수 있는 무게의 최댓값이 정해져 있고, 일정 가치와 무게가 있는 짐들을 배낭에 넣을 때 가치의 합이 최대가 되도록 짐을 고르는 방법을 찾는 문제인데요. 분할가능 배낭문제는 짐을 쪼갤 수 있는 경우에 해당합니다.

분할가능 배낭문제는 단위 무게당 값어치가 가장 큰 짐을 먼저 넣으면 되는데요. **순간순간의 선택이 이후 선택에 영향을 주지 않고, 순간순간의 최적 선택이 전체 문제 최적해와 일치합니다**. 따라서 분할가능 배낭문제는 탐욕 알고리즘으로 풀 수가 있습니다.

반면 짐을 쪼갤 수 없는 배낭문제를 0-1 배낭문제(0-1 Knapsack Problem)라고 합니다. **짐을 쪼갤 수 없기 때문에 가능한 모든 조합에 대해 일일이 따져본 후에 가치의 합이 최대가 되도록 하는 조합을 찾는 문제가 되는데, 이 때는 동적계획법(dynamic programming)으로 문제를 풀게 됩니다**. 다시 말해 모든 경우의 수를 따져보되 중간계산 결과를 저장해 두었다가 이를 다시 써먹는 방식으로 계산량 감소를 유도하는 전략입니다.

[출처 - <https://ratsgo.github.io/data%20structure&algorithm/2017/11/22/greedy/>]

크러스컬 알고리즘 (Kruskal Algorithm) 20.03.02

프로그래머스 레벨 3 – 섬 연결하기 문제를 풀고

탐욕적인 방법(Greedy method)를 이용하여 네트워크(가중치를 간선에 할당한 그래프)의 모든 정점을 최소 비용으로 연결하는 최적 해답을 구하는 것.

* 탐욕적인 방법
* 결정을 해야 할 때마다 **그 순간에 가장 좋다고 생각되는 것을 선택함으로써 최종적인 해답에 도달하는 것**
* 탐욕적인 방법은 그 순간에는 최적이지만, 전체적인 관점에서 최적이라는 보장이 없기 때문에 반드시 검증해야 한다.
* 다행히 Kruskal 알고리즘은 최적의 해답을 주는 것으로 증명되어 있다.
* **MST(최소 비용 신장 트리)** 가 1) 최소 비용의 간선으로 구성됨 2) 사이클을 포함하지 않음 의 조건에 근거하여 각 단계에서 사이클을 이루지 않는 최소 비용 간선을 선택 한다

크러스컬 알고리즘의 동작

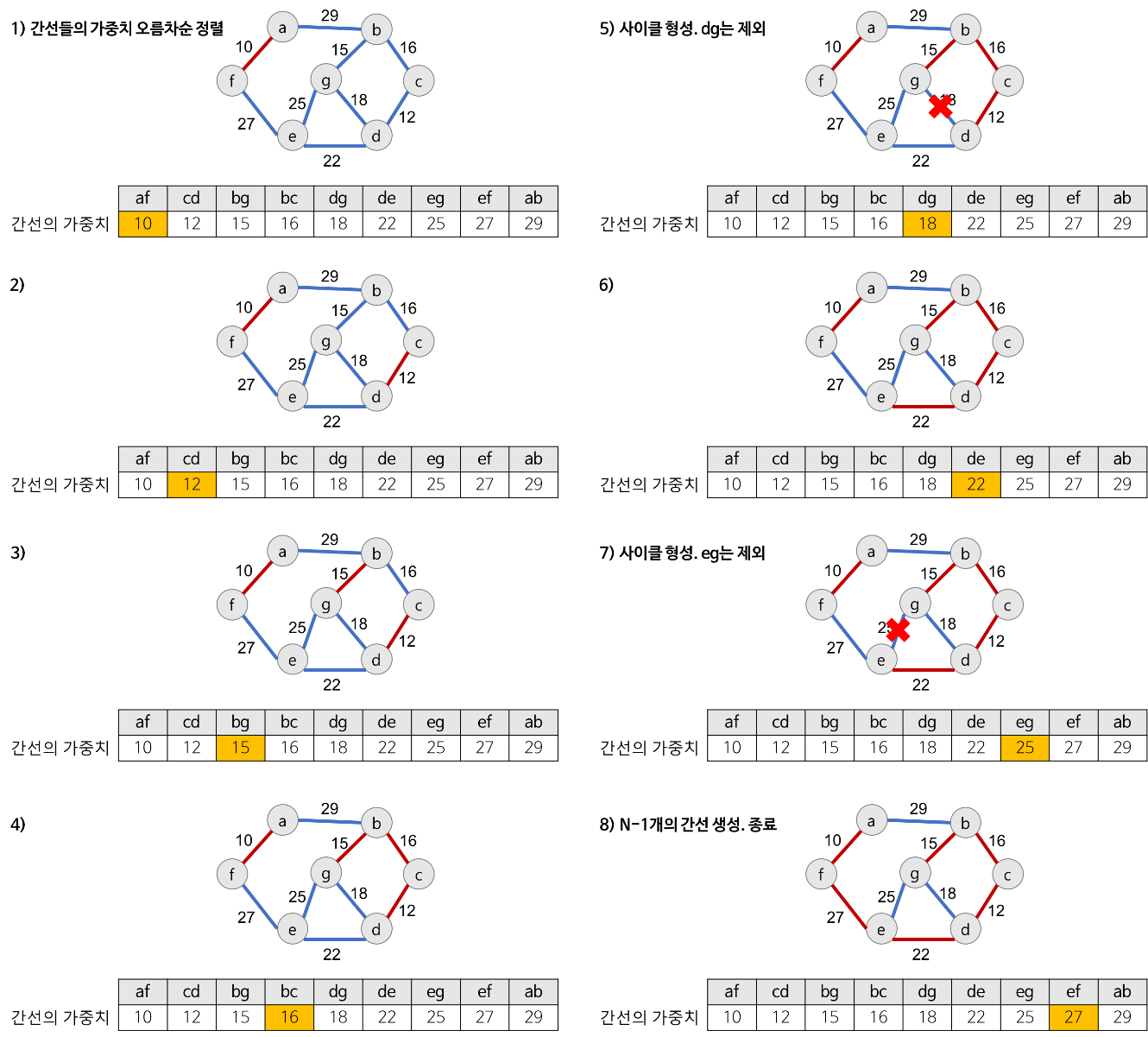
1. 그래프의 간선들을 가중치의 오름차순으로 정렬한다.
2. 정렬된 간선 리스트에서 순서대로 사이클을 형성하지 않는 간선을 선택한다.

* 즉, 가장 낮은 가중치를 먼저 선택한다.
* 사이클을 형성하는 간선은 제외한다.

1. 해당 간선을 현재의 MST(최소 비용 신장 트리)의 집합에 추가한다.

크러스컬 알고리즘의 구체적인 동작 과정

* **간선 선택을 기반**으로 하는 알고리즘
* 이전 단계에서 만들어진 신장 트리와는 상관 없이 무조건 최소 간선만을 선택하는 방법



주의!

* 다음 간선을 이미 선택된 간선들의 집합에 추가할 때 **사이클을 생성하는지를 체크!**
* 새로운 간선이 **이미 다른 경로에 의해 연결되어 있는 정점들을 연결할 때** 사이클이 형성된다.
* 즉, **추가할 새로운 간선의 양끝 정점이 같은 집합에 속해있으면** 사이클이 형성된다.

4번 그림의 경우 집합은 cd, bg, bc (양끝 정점은 d와 g)

* ***사이클 생성 여부를 확인하는 방법***
* 추가하고자 하는 간선의 양 끝 정점이 같은 집합에 속해 있는지를 먼저 검사해야 한다.

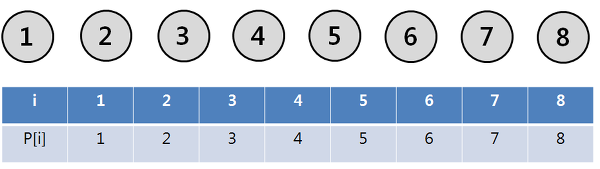
이를 ‘**union-find 알고리즘**’이라고 한다.

합집합 찾기 알고리즘 (Union-Find Algorithm)

대표적인 그래프 알고리즘으로 상호 배타적 집합(Disjoint-set)이라고도 한다.

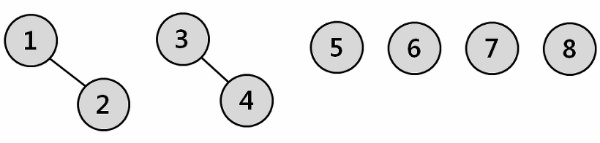
* 여러 노드가 존재할 때, 두 개의 노드를 선택해서, 현재 두 노드가 서로 같은 그래프에 속하는지 판별하는 알고리즘이다.
* 3가지 연산으로 이루어져있다.
* make-set(x) : 초기화. x를 유일한 원소로 하는 새로운 집합을 만든다.
* union(x, y) : 합하기. x와 y가 포함되어 있는 집합을 합치는 연산
* find : 찾기. x가 속한 집합의 루트 노드 값을 반환한다. 즉 x가 어떤 집합에 포함되어 있는지 찾는 연산

그림으로 보는 Union-Find 알고리즘

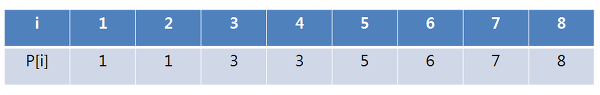


위와 같이, 모두 연결되지 않고 각자 자기 자신만을 집합의 원소로 가지고 있을 때, 모든 값이 자기 자신을 가리키도록 만든다.

i는 노드 번호, P[i]는 부모 노드번호를 의미하며, 즉 자기 자신이 어떤 부모에 포함되어 있는지를 의미한다. 정리하면, Parent[i] = i로 간단히 표현할 수 있다.

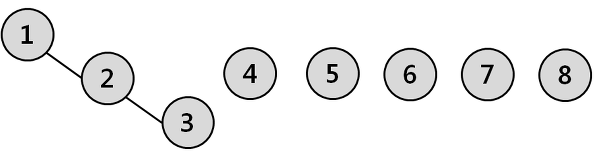


**Union(1,2); Union(3,4)** 를 해주어 위와 같이 노드를 연결해봅시다.

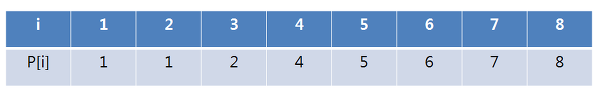


그럼 위와 같이 표에 표현이 된다. 2번째 인덱스에 '1'이 들어가고, 4번 인덱스에 '3'이 들어가는 것을 볼 수 있다.

이와 같이 부모를 합칠 때는 일반적으로 더 작은 값 쪽으로 합친다. 이것을 합침(Union) 과정이라고 말할 수 있다.

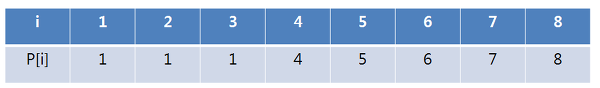


위와 같이 1, 2, 3이 연결될 때는 어떻게 표현이 될까요?? 바로 아래와 같이 표현이 됩니다.



1과 3은 부모가 다르기 때문에 '1과 3이 연결되었는지' 파악하기 힘이 듭니다. 이에 **재귀함수**가 사용됩니다.

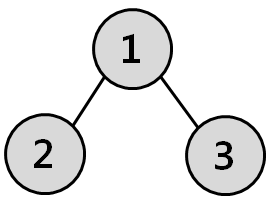
3의 부모인 2를 먼저 찾고, 2의 부모인 1을 찾아, 결과적으로 3의 부모는 1이 되는 것을 파악하는 것입니다.



Union의 과정이 수행된 후에는 다음과 같은 표로 바뀝니다.

결국 1,2,3의 부모는 모두 1이기 때문에 이 세 가지 노드는 모두 같은 그래프에 속한다는 것을 알 수 있습니다.

해당 경로를 바꿔주는 과정은 아래와 같은 그림으로 변하게 됩니다.



[출처 - <https://brenden.tistory.com/33>]

Union-Find 알고리즘을 트리 구조로 구현하는 이유

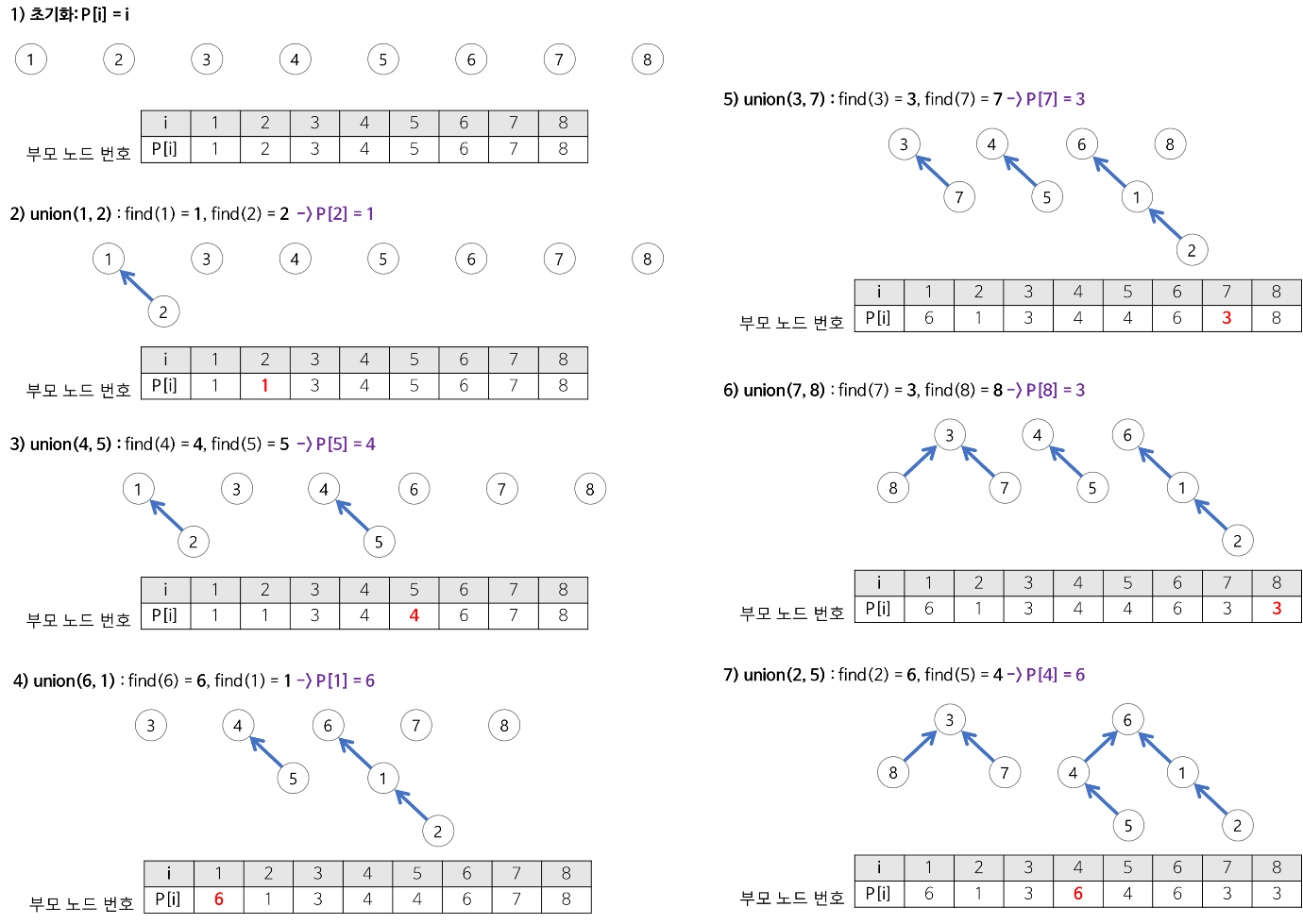
1. 배열

* Array[i] : i번 원소가 속하는 집합의 번호(즉, 루트 노드의 번호)
* make-set(x)
* Array[i] = i와 같이 각자 다른 집합 번호로 초기화한다.
* union(x, y)
* 배열의 모든 원소를 순회하면서 y의 집합 번호를 x의 집합 번호로 변경한다.
* 시간 복잡도 : O(N)
* find(X)
* 한번만에 x가 속한 집합 번호를 찾는다.
* 시간 복잡도: O(1)

1. 트리

* 같은 집합 = 하나의 트리, 즉, 집합 번호 = 루트 노드
* make-set(x)
* 각 노드는 모두 루트 노드이므로 N개의 루트 노드 생성 및 자기 자신으로 초기화한다.
* union(x, y)
* x, y의 루트 노드를 찾고 다르면 y를 x의 자손으로 넣어 두 트리를 합한다.
* 시간 복잡도 : O(N)보다 작으므로 find 연산이 전체 수행 시간을 지배한다.
* find(X)
* 노드의 집합 번호는 루트 노드이므로, 루트 노드를 확인하여 같은 집합인지 확인한다.
* 시간 복잡도: 트리의 높이와 시간 복잡도가 동일하다. (최악: O(N-1))

Union-Find의 과정



Union-Find의 사용 예시

전체 집합이 있을 때 **구성 원소들이 겹치지 않도록 분할 하는데** 자주 사용된다.

* Kruskal MST 알고리즘에서 새로 추가할 간선의 양끝 정점이 같은 집합에 속해 있는지(사이클 형성 여부 확인)에 대해 검사하는 경우
* 초기에 {0}, {1}, {2}, … {n} 이 각각 n+1개의 집합을 이루고 있다. 여기에 합집합 연산과, 두 원소가 같은 집합에 포함되어 있는지를 확인하는 연산을 수행하려는 경우
* ex) 집합의 표현 – 백준 1717번
* 어떤 사이트의 친구 관계가 생긴 순서대로 주어졌을 때, 가입한 두 사람의 친구 네트워크에 몇 명이 있는지 구하는 프로그램을 작성하는 경우
* ex) 친구 네트워크 – 백준 4195번

분할(Partition)이란?

* 임의의 집합을 분할한다는 것은 각 부분 집합이 아래의 두 가지 조건을 만족하는 Disjoint set(상호 배타 집합)이 되도록 쪼개는 것이다.

1. 분할된 부분 집합을 합치면 원래의 전체 집합이 된다.
2. 분할된 부분 집합끼리는 겹치는 원소가 없다.

* 예를 들어, S = {1, 2, 3, 4}, A = {1, 2}, B = {3, 4}, C = {2, 3, 4}, D = {4}라면
* A와 B는 S의 분할 O. A와 B는 Disjoint Set
* A와 C는 S의 분할 X. 겹치는 원소가 존재
* A와 D는 S의 분할 X. 두 집합을 합해도 S가 되지 않음

Union-Find 일반적 구현 방법

*/\* 초기화 \*/*

**int** root[MAX\_SIZE];

**for** (**int** i = 0; i < MAX\_SIZE; i++)

parent[i] = i;

*/\* find(x): 재귀 이용 \*/*

**int** find(**int** x) {

*// 루트 노드는 부모 노드 번호로 자기 자신을 가진다.*

**if** (root[x] == x) {

**return** x;

} **else** {

*// 각 노드의 부모 노드를 찾아 올라간다.*

**return** find(root[x]);

}

}

*/\* union(x, y) \*/*

**void** union(**int** x, **int** y){

*// 각 원소가 속한 트리의 루트 노드를 찾는다.*

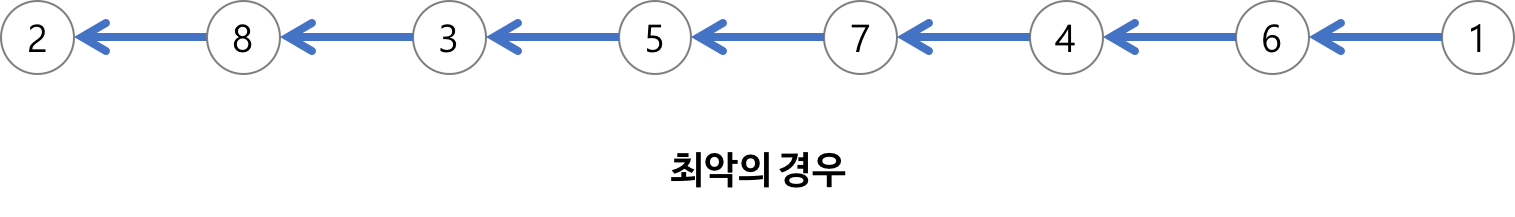
x = find(x);

y = find(y);

root[y] = x;

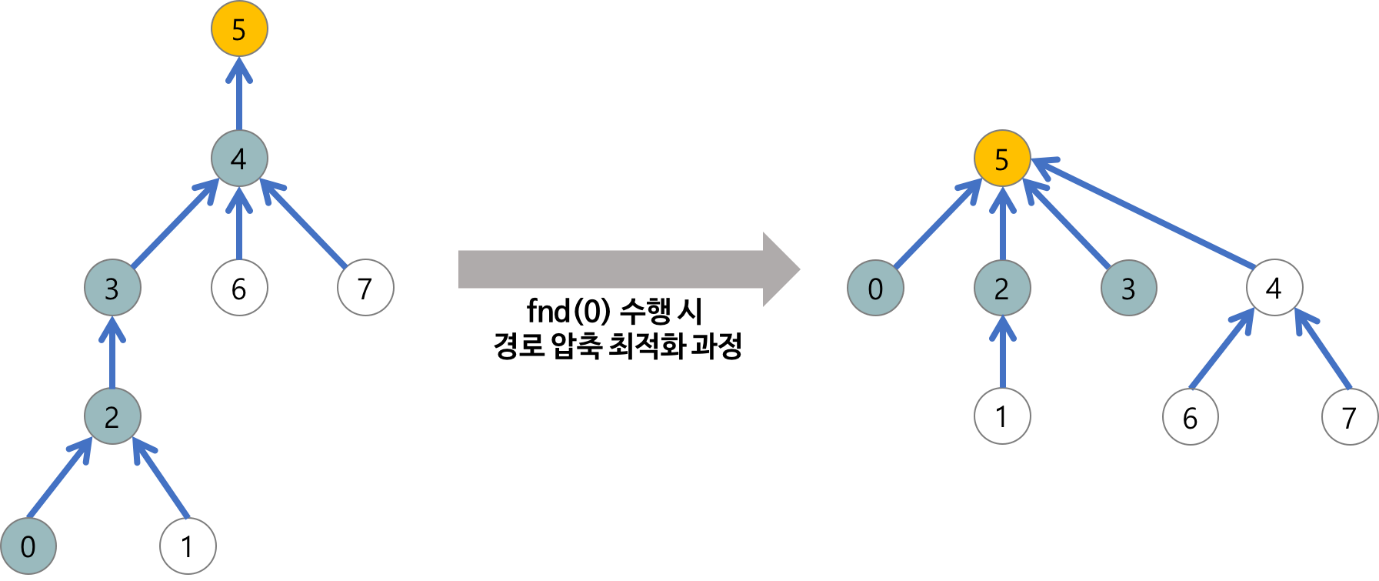
}

Union-Find 최적화한 구현 방법



* 트리 구조가 완전 비대칭인 경우
* 연결 리스트 형태
* 트리의 높이가 최대가 된다
* 원소의 개수가 N일 때, 트리의 높이가 N-1이므로 union과 find(x)의 시간 복잡도가 모두 O(N)이 된다.
* 배열로 구현하는 것보다 비효율적이다

find 연산 최적화



* 경로 압축(Path Compression)
* 시간 복잡도: O(logN)
* */\* 초기화 \*/*
* **int** root[MAX\_SIZE];
* **for** (**int** i = 0; i < MAX\_SIZE; i++) {
* root[i] = i;
* }
* */\* find(x): 재귀 이용 \*/*
* **int** find(**int** x) {
* **if** (root[x] == x) {
* **return** x;
* } **else** {
* *// "경로 압축(Path Compression)"*
* *// find 하면서 만난 모든 값의 부모 노드를 root로 만든다.*
* **return** root[x] = find(root[x]);
* }
* }

union 연산 최적화

* union-by-rank(union-by-height)
* rank에 트리의 높이를 저장한다.
* 항상 높이가 더 낮은 트리를 높은 트리 밑에 넣는다.

*/\* 초기화 \*/*

**int** root[MAX\_SIZE];

**int** rank[MAX\_SIZE]; *// 트리의 높이를 저장할 배열*

**for** (**int** i = 0; i < MAX\_SIZE; i++) {

root[i] = i;

rank[i] = 0; *// 트리의 높이 초기화*

}

*/\* find(x): 재귀 이용 \*/*

**int** find(**int** x) { *// 동일*

}

*/\* union1(x, y): union-by-rank 최적화 \*/*

**void** **union**(**int** x, **int** y){

x = find(x);

y = find(y);

*// 두 값의 root가 같으면(이미 같은 트리) 합치지 않는다.*

**if**(x == y)

**return**;

*// "union-by-rank 최적화"*

*// 항상 높이가 더 낮은 트리를 높이가 높은 트리 밑에 넣는다. 즉, 높이가 더 높은 쪽을 root로 삼음*

**if**(rank[x] < rank[y]) {

root[x] = y; *// x의 root를 y로 변경*

} **else** {

root[y] = x; *// y의 root를 x로 변경*

**if**(rank[x] == rank[y])

rank[x]++; *// 만약 높이가 같다면 합친 후 (x의 높이 + 1)*

}

}

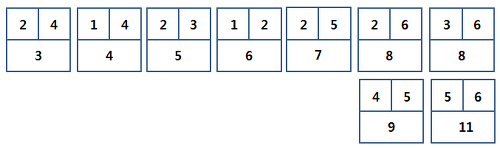
[여기까지의 출처 - <https://gmlwjd9405.github.io/2018/08/31/algorithm-union-find.html>]

다시 크러스컬로 돌아와서…

* **최소 비용 신장 트리**를 찾는 알고리즘이다.
* **가장 적은 비용으로 모든 노드를 연결하기 위해 사용**하는 알고리즘이다.
* **최소 스패닝 트리(Minimum Spanning Tree)**를 찾음으로서 간선의 가중치의 합이 최솟값이 되게한다.
* **스패닝 트리** : 그래프에서 일부 간선을 선택해서 만든 트리
* **최소 스패닝 트리** : 스패닝 트리 중에 선택한 간선의 가중치의 합이 최소인 트리
* 변의 개수 **E(간선)**, 꼭짓점의 개수 **V(노드)**라고 하면 이 알고리즘은 O(E log V)의 시간 복잡도를 가진다.
* **Union-Find** 개념에 대한 이해가 필요하다.

**크러스컬 알고리즘은 다음과 같은 절차로 진행된다.**

1. 모든 간선들을 거리(비용,가중치)를 기준으로 오름차순으로 정렬한다.

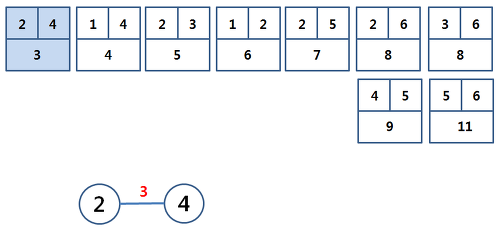


2. 정렬된 간선을 순서대로 선택한다.

3. 선택한 간선의 두 정점이 연결되어 있지 않으면, 해당 두 정점을 연결시킨다.

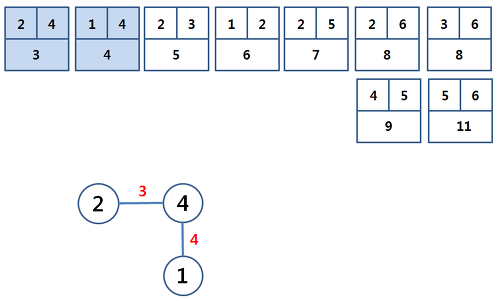
- 즉, 사이클 테이블을 통해 두 점이 연결되어있는지 여부를 파악한다

(**Union-Find 알고리즘 사용**)

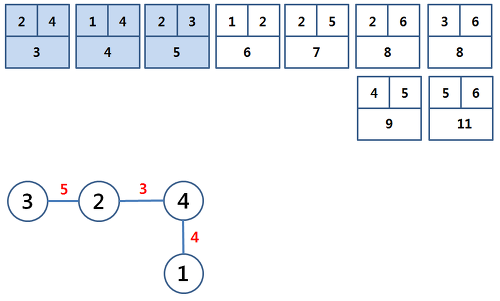


STEP 2, 3을 진행한다. 가장 가중치가 낮은 간선 3을 선택하여 두 노드를 연결해준다.

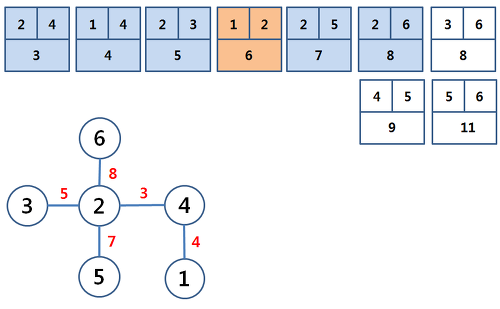
사이클 테이블에 4의 경우 2로 갱신하게 되고 나머지 경우 모두 1로 갱신하게 된다.

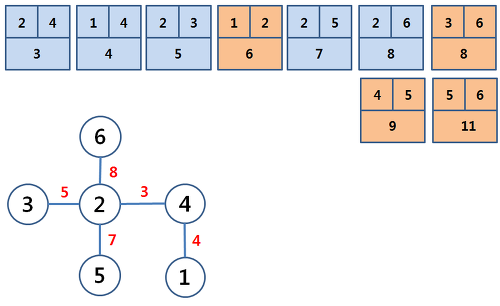


위 그림처럼 가장 가중치가 낮은 두 번째 간선을 선택한다. 사이클 테이블도 갱신해준다.



위 그림처럼 가장 가중치가 낮은 세 번째 간선을 선택한다. 사이클 테이블도 갱신해준다.





주황색으로 표시된 간선은 사이클이 생기기 때문에 선택하지 않은 간선이다.

위 그림처럼 가장 가중치가 낮은 여섯 번째 간선을 선택한다.

6개의 모든 노드가 연결되었기 때문에 이 후에 있는 간선들은 확인하지 않아도 된다.

사이클 테이블의 모든 값이 1이 되면서 최소 비용 신장 트리가 만들어진 것을 볼 수 있다.

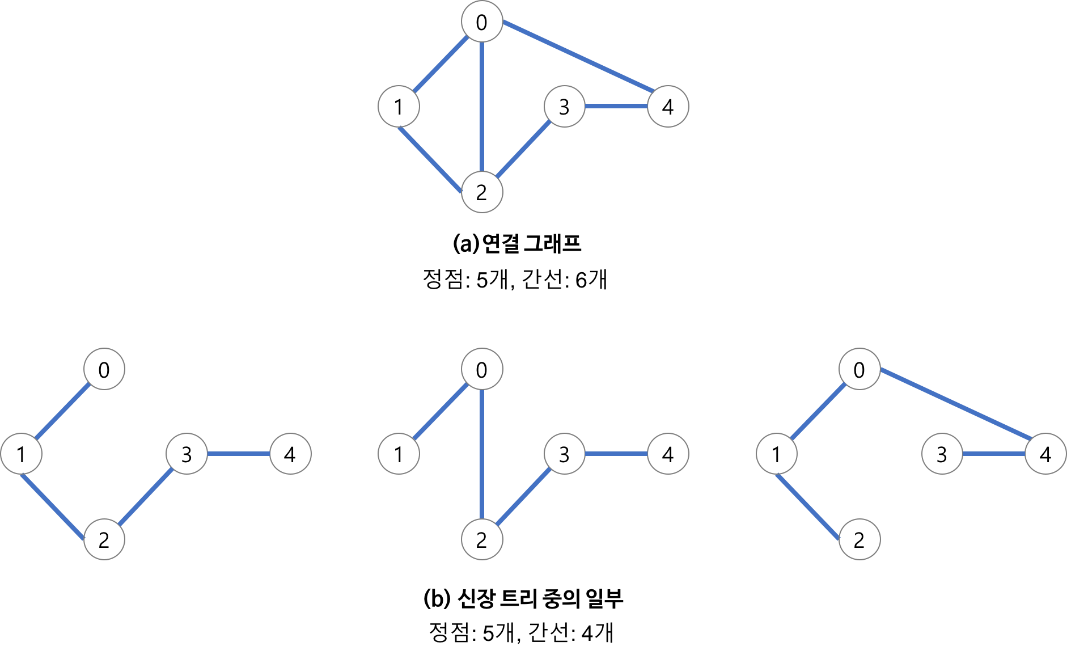
최소 신장 트리(MST, Minimum Spanning Tree) – 21.03.03

**Spanning Tree란**

그래프 내의 모든 정점을 포함하는 트리

* **Spanning Tree = 신장 트리 = 스패닝 트리**
* Spanning Tree는 그래프의 **최소 연결 부분 그래프**이다.
* 최소 연결 = 간선의 수가 가장 적다.
* n개의 정점을 가지는 그래프의 최소 간선의 수는 (n – 1)개 이고, **(n – 1)개의 간선으로 연결되어 있으면 필연적으로 트리 형태가 되고** 이것이 바로 Spanning Tree가 된다.
* 즉, 그래프에서 일부 간선을 선택해서 만든 트리

Spanning Tree의 특징



* **DFS, BFS**를 이용하여 그래프에서 신장 트리를 찾을 수 있다.
* 탐색 도중에 사용된 간선만 모으면 만들 수 있다.
* 하나의 그래프에는 많은 신장 트리가 존재할 수 있다.
* Spanning Tree는 트리의 특수한 형태이므로 **모든 정점들이 연결되어 있어야하고 사이클을 포함해서는 안된다**.
* 따라서 Spanning Tree는 그래프에 있는 n개의 정점을 정확이 (n - 1)개의 간선으로 연결한다.

Spanning Tree의 사용 사례

통신 네트워크 구축

* 예를 들어, 회사 내의 모든 전화기를 가장 적은 수의 케이블을 사용하여 연결하고자 하는 경우
* n개의 위치를 연결하는 통신 네트워크를 최소의 링크(간선)을 이용하여 구축하고자 하는 경우, 최소 링크의 수는 (n – 1)개가 되고, 따라서 Spanning Tree가 가능해진다.

MST란

Spanning Tree 중에서 **사용된 간선들의 가중치 합이 최소**인 트리

* 각 간선의 가중치가 동일하지 않을 때 단순히 가장 적은 간선을 사용한다고 해서 최소 비용이 얻어지는 것은 아니다.
* MST는 간선에 가중치를 고려하여 최소비용의 Spanning Tree를 선택하는 것을 말한다.
* 즉, 네트워크에 있는 모든 정점들을 가장 적은 수의 간선과 비용을 연결하는 것이다.

MST의 구현 방법

1. Kruskal MST 알고리즘

탐욕적인 방법(Greedy Method)을 이용하여 네트워크의 모든 정점을 최소 비용으로 연결하는 최적 해답을 구하는 것

* 각 단계에서 **사이클을 이루지 않는 최소 비용 간선을 선택**한다.
* **간선 선택**을 기반으로 하는 알고리즘이다.
* 이전 단계에서 만들어진 신장 트리와는 상관 없이 무조건 최소 간선만을 선택하는 방법이다.

1. Prim MST 알고리즘

시작 정점에서부터 출발하여 **신장트리 집합을 단계적으로 확장**해나가는 방법

* **정점 선택**을 기반으로 하는 알고리즘이다.
* 이전 단계에서 만들어진 신장 트리를 확장하는 방법이다.

1. 결론

* 프림은 시작점을 정하고, 시작점에서 가까운 정점을 선택하면서 트리를 구성 하므로 그 과정에서 사이클을 이루지 않지만 크루스칼은 시작점을 따로 정하지 않고 최소 비용의 간선을 차례로 대입 하면서 트리를 구성하기 때문에 사이클이 이루어지는 항상 확인 해야한다.
* 프림의 경우 최소 거리의 정점을 찾는 부분에서 자료구조의 성능에 영향을 받는다.
* 크루스칼은 간선을 기준으로 정렬하는 과정이 오래 걸린다.
* 간선의 개수가 작은 경우에는 크루스칼, 간선의 개수가 많은 경우에는 프림.

[출처 - <https://gmlwjd9405.github.io/2018/08/28/algorithm-mst.html>]

프림 알고리즘 (Prim Algorithm)

시작 정점에서부터 출발하여 신장트리 집합을 단계적으로 확장해나가는 방법. 크러스컬과 같이 Greedy Method를 사용한다.

Prim 알고리즘의 동작

1. 시작 단계에서는 시작 정점만이 MST(최소 비용 신장 트리) 집합에 포함된다.
2. 앞 단계에서 만들어진 MST 집합에 인접한 정점들 중에서 최소 간선으로 연결된 정점을 선택하여 트리를 확장한다

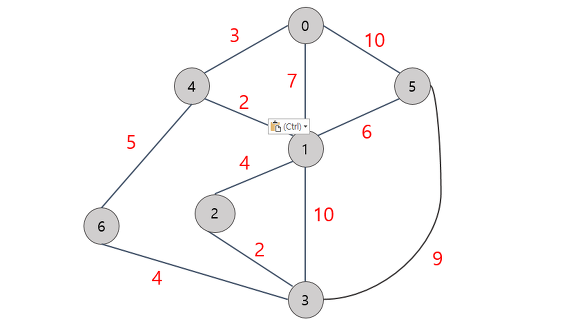
* 즉, 가장 낮은 가중치를 먼저 선택한다.
* 여기서 우선순위 큐를 사용한다.

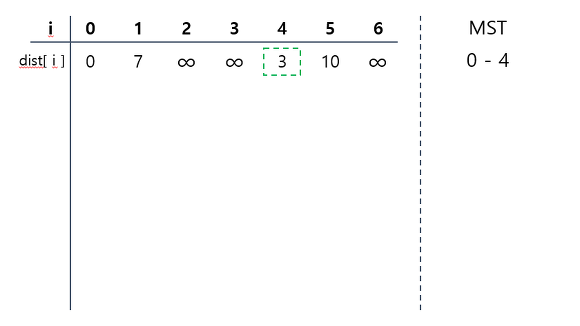
1. 위의 과정을 트리가 (N – 1)개의 간선을 가질 때까지 반복한다.

Prim 알고리즘의 구체적인 동작 과정

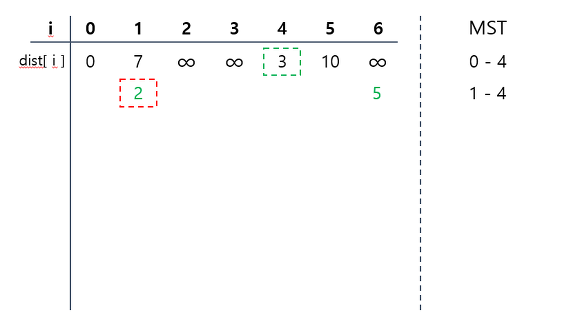
Prim 알고리즘을 이용하여 MST를 만드는 과정

* 정점 선택을 기반으로 하는 알고리즘
* 이전 단계에서 만들어진 신장 트리를 확장

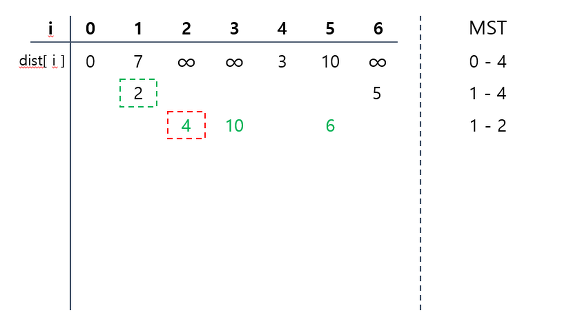




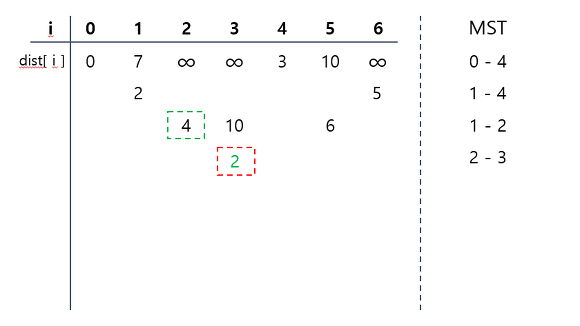
1. 시작점을 0번 정점으로 설정해둔 후에 각 정점까지의 거리가 얼마인지 적는다. 무한대로 표시되어 있는 것은 바로 가는 경로가 없고 어딘가를 거쳐서 가야함을 뜻한다. 우리는 **직접적으로 연결되어 있는 정점의 거리만 표시**한다. 표시한 거리 중 **가장 짧은 거리는 0번 정점과 4번 정점의 거리인 3**이다. 따라서 첫 번째 경로인 0 - 4 경로(간선)을 찾았다.



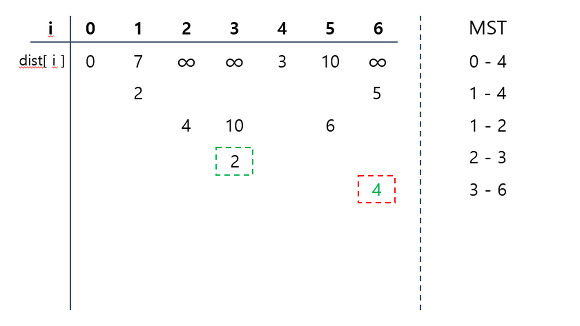
1. 기존의 경로인 0 - 4를 기준으로 두고 생각해보자. **4번 정점을 기준으로** 새롭게 나아갈 수 있는 곳들의 거리를 표시해보자. 4번 정점에서는 1번 정점, 그리고 6번 정점으로 갈 수 있으므로 각 거리를 적어준다. 그 중에서 **1번 정점까지의 거리가 가장 짧으므로 1번 정점을 택한다**. 이로서 1- 4의 경로가 선택됐다.



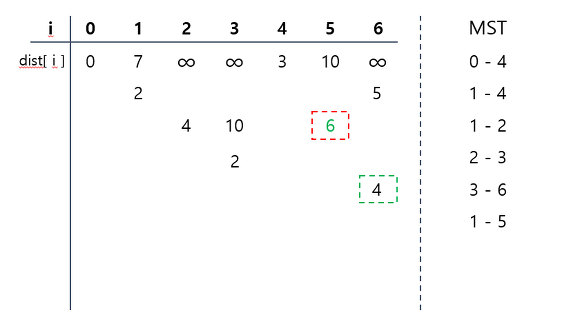
1. 이번에는 **1번 정점**에서 어디로 나아갈 수 있는지를 찾아본다. 2번과 3번, 그리고 5번으로 나아갈 수 있다. 여기서 5번 정점을 주목해보자. **이미 길이가 10인 경로가 존재했지만, 1번 정점을 통하면 거리가 6으로 줄어든다**. 우린 **최소경로를 찾고 있는 것이기 때문에 이렇게 더 짧은 경로가 발견되면 그 정보를 갱신**해준다. 이미 선택된 최소경로를 제외하면, 새롭게 나아갈 곳 중 **가장 짧은 경로는 1 - 2 경로**다. 따라서 **2번 정점을 선택**한다.



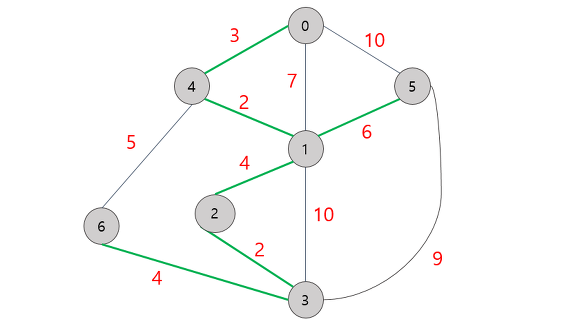
1. 위에서 선택한 2번을 토대로 갱신할 수 있는 경로 혹은 새롭게 나아갈 수 있는 경로가 있는지 탐색한다. **새롭게 나아갈 수 있는 경로는 없었지만, 갱신할 수 있는 정보가 있다**. **2번에서 3번 경로가 기존의 10길이에서 2길이로 줄어드므로 갱신해준다**. 그 후 현재까지 선택된 경로 중 가장 거리가 짧은 경로인 2 - 3 경로를 선택해준다. **이로써 3번 정점이 선택**됐다.



1. 위에서 선택한 3번 정점을 토대로 새로얻을 수 있는 정보는 3 - 6 경로의 거리 정보다. **기존의 길이인 5보다 더 짧은 4가 나오므로 정보를 갱신해준다**. 그 후 지금까지 선택된 정점을 제외한 것 중 **가장 짧은 경로인 3 - 6 경로를 선택**한다. 즉 우리는 **6번 정점**을 선택한 것이다.



1. 마지막이다. **지금까지의 경로 중 선택되지 않은 정점은 5번 정점**이다. 5번 정점을 선택하고 새롭게 갱신할 수 있는 정보가 있는지 살펴본다. **없다. 5번 정점의 거리는 1번 정점을 선택할 때 나온 것이므로, 1 - 5의 경로가 선정됐다**. 이로써 **Prim 알고리즘의 종료**다. MST에 적어둔 경로들로 종합한 것이 그래프의 최단 경로다.



1. Prim을 직접 손으로 작성해보고, 그 결과인 간선들을 토대로 탄생한 최소 신장 그래프는 위의 그래프다.

