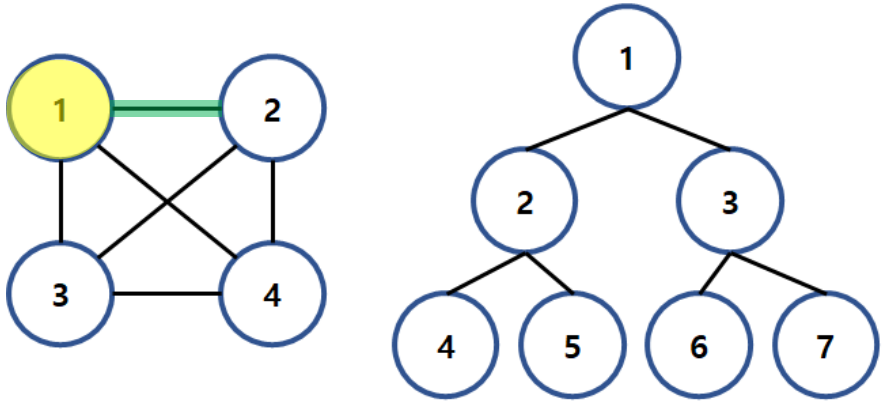
그래프

그래프 구현, 위상정렬, 최단거리 구하기(다익스트라, 플로이드), 최소 신장 트리 구하기(크루스칼, 프림)

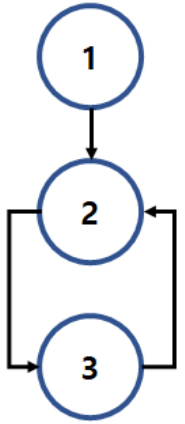
패기반 이은비

그래프

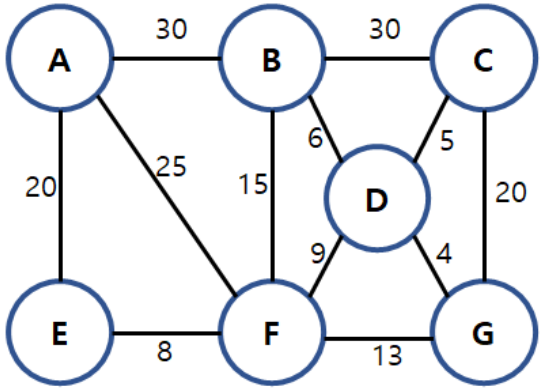
무방향 그래프



방향 그래프



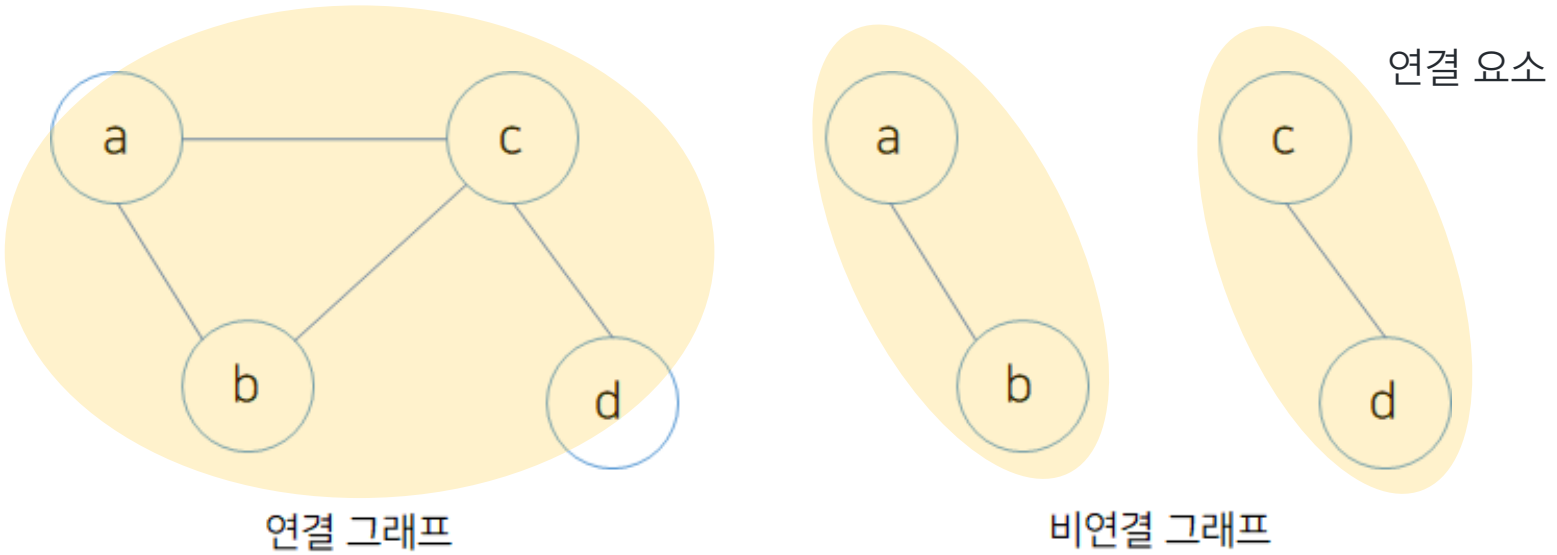
가중치 그래프



- 노드(정점, vertex) : 각각의 지점
- 간선 : 정점과 정점을 잇는 선
    - 가중치 : 간선에 부여된 값 (없을수도, 있을수도)
- 차수(degree) : 정점A에 연결된 다른 정점의 수.
    - 방향 그래프의 경우, 진입차수(In-degree), 진출차수(Out-degree)
- 사이클(cycle) : 특정 지점에서 출발하여, 본래 지점으로 돌아올 수 있는 경우

EX)
1의 차수(무방향 그래프) : 3
1의 진입차수(방향 그래프) : 0, 진출차수 : 1

그래프



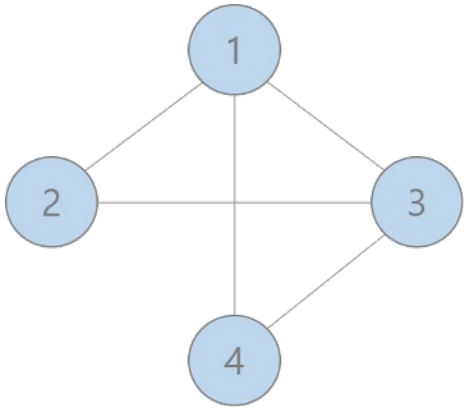
연결 그래프 : 어느 정점에서라도 모든 정점으로 갈 수 있는 경로가 존재하는 그래프 ↔ **비연결 그래프**

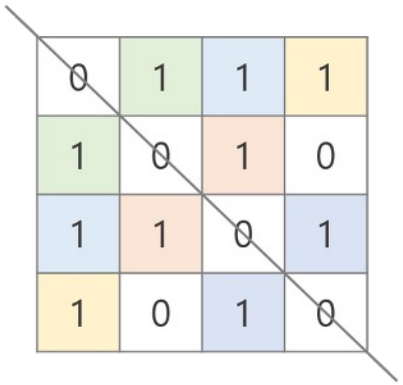
*** 서로소 집합 (Disjoint Sets Algorithm)**
→ Union-find 알고리즘
- 비연결 그래프 판별
- 사이클 판별

그래프 구현 : 인접 행렬, 인접 리스트

정점의 수 |V|, 간선의 수 |E|

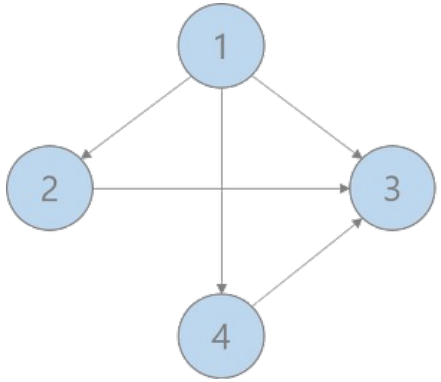
인접 행렬

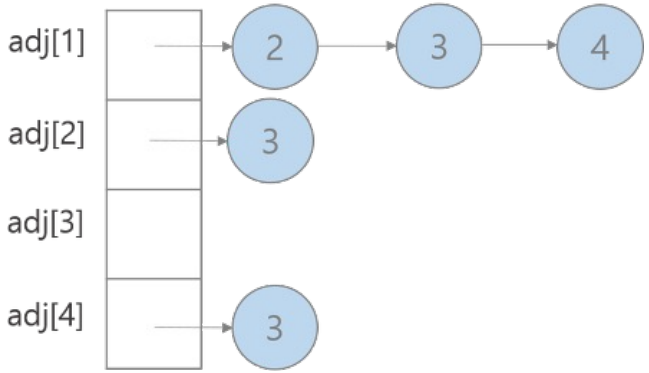




- |V|×|V| 크기의 2차원 배열
- A → B 로 가는 길이 있다면 배열의 값을 1로, 가는 길이 없다면 배열의 값을 0으로 지정함

인접 리스트



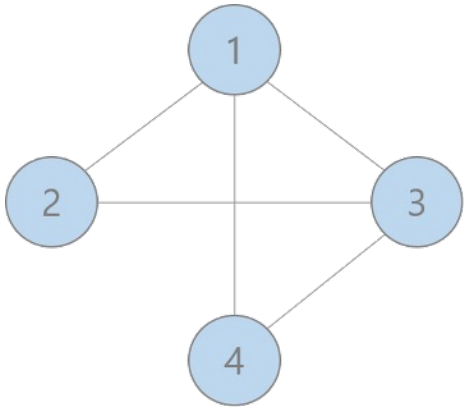


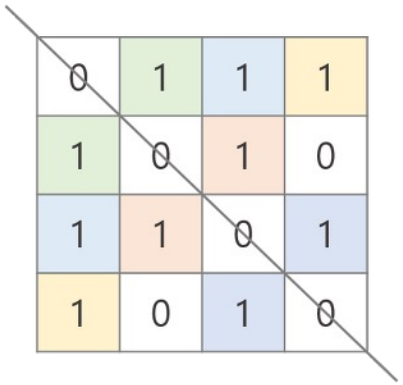
- |V|개의 연결 리스트
- 연결 리스트에 특정 정점과 인접해있는 정점들의 정보를 담음.

그래프 구현 : 인접 행렬, 인접 리스트

정점의 수 |V|, 간선의 수 |E|

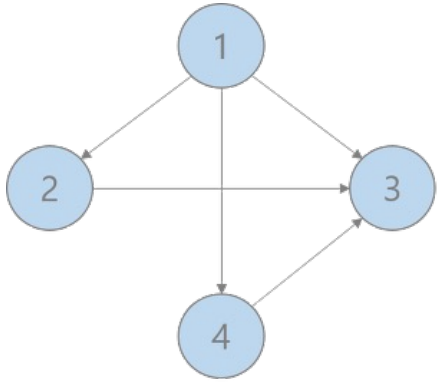
인접 행렬

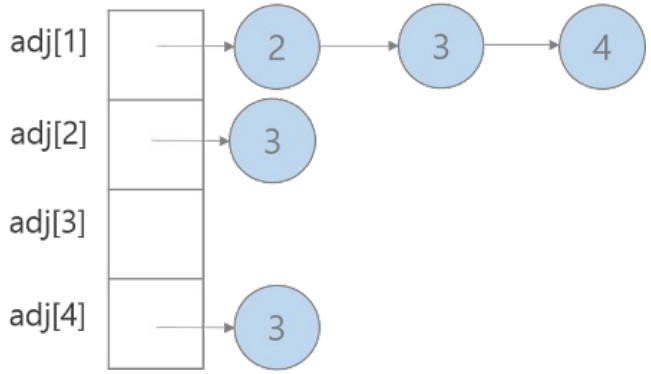




- 정점 I, J 가 연결되어 있는지를 확인: O(1)
- 특정 정점과 연결되어 있는 모든 정점을 확인: O(|V|)
- 공간 복잡도: O(|V|*|V|)

인접 리스트



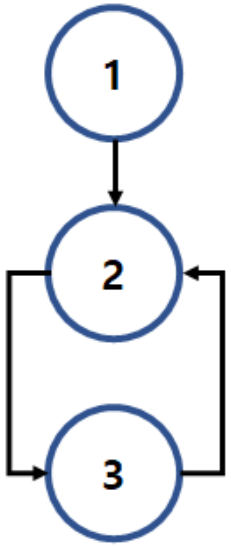


- 정점 I, J 가 연결되어 있는지를 확인:
  O(min(degree(I),degree(J)))
- 특정 정점과 연결되어 있는 모든 정점을 확인:
  O(degree(X))
- 공간 복잡도: O(|V|+|E|)

→ 일반적으로 인접 행렬이 인접 리스트보다 메모리 더 차지
→ 두 정점이 연결되었는지 확인은 인접 행렬에서 더 빠름

방향 그래프의 순서 : 위상정렬

방향 그래프

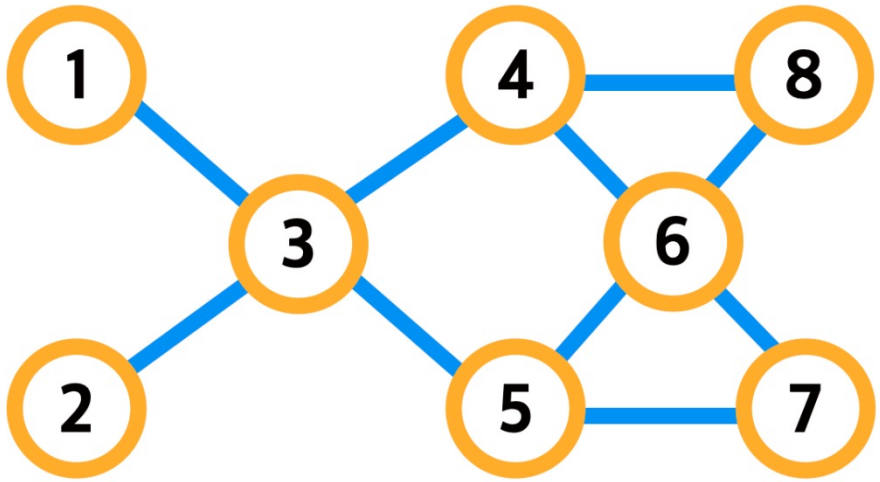


위상정렬 : 그래프의 꼭짓점들을 변의 방향을 거스르지 않도록 나열하는 것을 의미
= 그래프의 방향을 거스르지 않게 나열

앞에 처리해야 할 순서가 끝나고 난 뒤에 현재 일을 처리.
→ 진입차수(in-degree)가 0인 지점을 항상 먼저 지나가면 됨

1. 그래프에서 in-degree가 0인 지점을 전부 queue에 넣고
2. Queue에서 지점 하나를 뽑아, 해당 정점(지정)에 연결되어 있는 모든 간선을 살펴봄.
3. 해당 간선이 연결된 정점의 in-degree를 1 감소시킴 (= 해당 간선을 지움)
4. 1~3을 queue가 빌 때까지, 새롭게 In-degree가 0인 지점이 사라질 때까지 반복

최단거리 구하기 : BFS

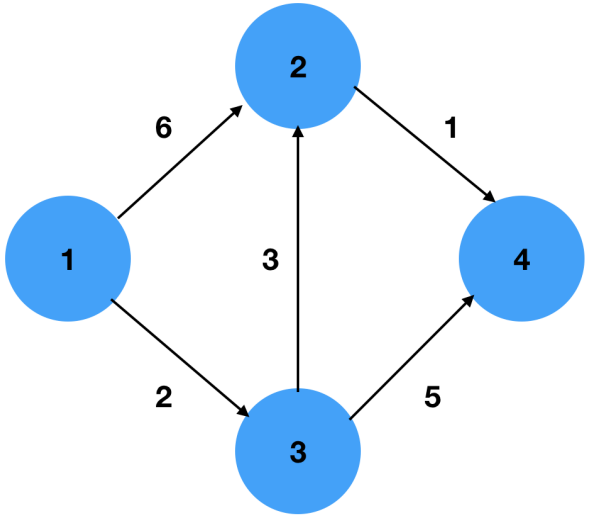


1. 최단거리를 구하는데 적합한 방식은 DFS 가 아닌 BFS 입니다.
2. 1에서 6까지 갈 수 있는 1 - 3 - 4 - 6 또는 1 - 3 - 5 - 6 두 길 모두 길이가 3이므로, 1번에서 6번까지의 최단거리는 3이 됩니다.
3. 가중치가 전부 동일한 그래프에서는 BFS로 최단거리를 구할 수 있습니다.
4. 이 그래프는 양쪽으로 이동가능한 무방향 그래프이기 때문에, 8번 정점에서 BFS 를 돌려서 1번 정점까지의 최단거리를 재도 1번 정점에서 8번 정점까지의 최단거리를 구한 것과 동일한 값이 나옵니다.
5. BFS 탐색 진행중 노드를 처음 방문하면 최단거리를 얻을 수 있고, 이미 방문하여 최단거리를 얻은 노드는 다시 방문하지 않기 때문에, 사이클 여부와 관계 없이 최단거리를 구할 수 있습니다.

가중치가 없으면 → BFS 로 최단거리 구하기 가능.
가중치가 있으면… 다른 최단거리 탐색방법

https://www.codetree.ai/missions/6/problems/graph-shortest-path-1/explanation

최단거리 구하기 : 다익스트라 vs. 플로이드



다익스트라 알고리즘

한 지점에서 다른 모든 지점까지의 최단 거리를 계산하는 알고리즘

플로이드 워셜 알고리즘

모든 지점에서 다른 모든 지점까지의 최단 거리를 계산하는 알고리즘

다익스트라 vs. 플로이드

```python
def dijkstra(start):
    distance = np.array([INF] * len(graph))

    # 시작 노드에 대해 초기화
    distance[start] = 0

    q = []
    hq.heappush(q, (0, start))

    while q:
        dist, now = hq.heappop(q)

        # 방문 여부 확인
        if dist > distance[now]:
            continue

        for i in graph[now]:
            # cost는 start 기준 거리 총합 -> 최단 거리 일 때 distance update하고 그 노드 push
            cost = dist + i[2]
            if cost < distance[i[1]]:
                distance[i[1]] = cost
                hq.heappush(q, (cost, i[1]))


    return distance[1:]
```

음수 가중치가 있는 그래프 에서는 다익스트라가 올바르게 동작하지 않을 수 있음. 그 까닭은, 다익스트라에서는 dist중 가장 작은 값을 골랐을 때 그 값이 확실한 최단거리라는 보장이 되어야 하는데, 음수 가중치가 있으면 다시 골라졌던 정점에 도달하는 dist값이 더 작아질 수도 있기 때문에 최단거리임을 보장할 수 없게 되기 때문.

다익스트라 알고리즘은 최단 경로를 구하는 과정에서 '각 노드에 대한 현재까지의 최단 거리' 정보를 항상 1차원 리스트에 저장하며 리스트를 계속 갱신한다는 특징이 있다.

다익스트라 최단 경로 알고리즘에서는 '방문하지 않은 노드 중에서 가장 최단 거리가 짧은 노드를 선택'하는 과정을 반복하는데, 이렇게 선택된 노드는 '최단 거리'가 완전히 선택된 노드이므로, 더 이상 알고리즘을 반복해도 최단 거리가 줄어들지 않는다. 다시 말해 다익스트라 알고리즘이 진행되면서 한 단계당 하나의 노드에 대한 최단 거리를 확실히 찾는 것으로 이해할 수 있다.

특정 B지점까지 거리
= A까지 가는 거리
+ A에서 특정 B지점까지 소요되는 거리

1.출발 노드를 설정한다.
2.최단거리 테이블을 초기화(INF)한다.
3.방문하지 않은 노드 중에서 최단 거리가 가장 짧은 노드를 선택한다.
4.해당 노드를 거쳐 다른 노드로 가는 비용을 계산하여 최단 거리 테이블을 갱신한다.
5.위 과정에서 3번과 4번을 반복한다.

https://data-scientist-han.tistory.com/103

다익스트라 vs. **플로이드**

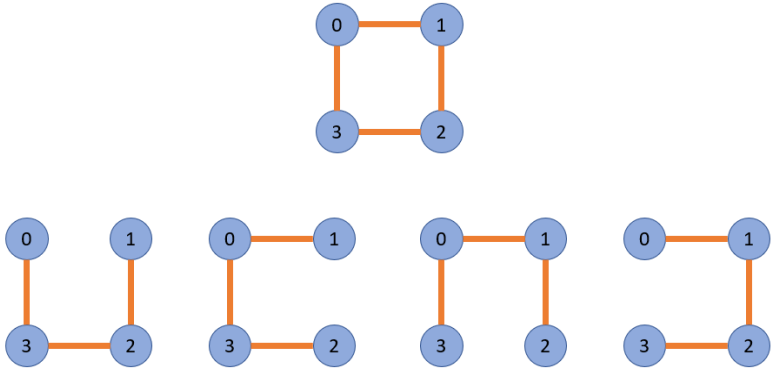
```
# 자기 자신에서 자신은 0
for a in range(1, n+1):
    for b in range(1, n+1):
        if a == b:
            graph[a][a] = 0


i = 0
for m in input_:
    for a, b in m:
        graph[i][a] = b
    i+=1



# k를 거쳐가는 경우
for k in range(1, n+1):
    for a in range(1, n+1):
        for b in range(1, n+1):
            # 플로이드 워셜 알고리즘 점화식
            graph[a][b] = min(graph[a][b], graph[a][k] + graph[k][b])
```

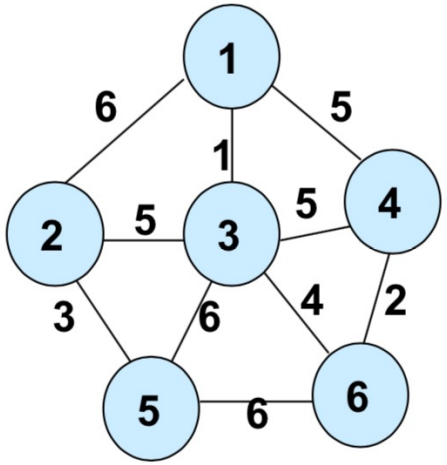
- 모든 정점 간의 거리를 알고 싶을 때, 다익스트라를 사용할 경우 모든 정점에 대해 다익스트라를 한 번씩 돌려야하는데 대신 이때 플로이드를 사용할 수 있음.
- 플로이드 : 다익스트라와 유사한데, A → B로 가는 경로보다 A → X → B로 가는 경로가 더 짧다면 그것으로 갱신을 해주는 것
- 모든 쌍 (I, j)에 대해 특정 정점 A를 경유하는 것이 좋은 경우 값을 갱신함.
    - dp[i][j] > dp[i][1] + dp[1][j]를 만족하는 경우 dp[i][j]에 dp[i][1] + dp[1][j]값을 넣어줌.
    - (초기에 dp[i][j]는 i에서 j까지의 바로 가는 비용을 뜻함)

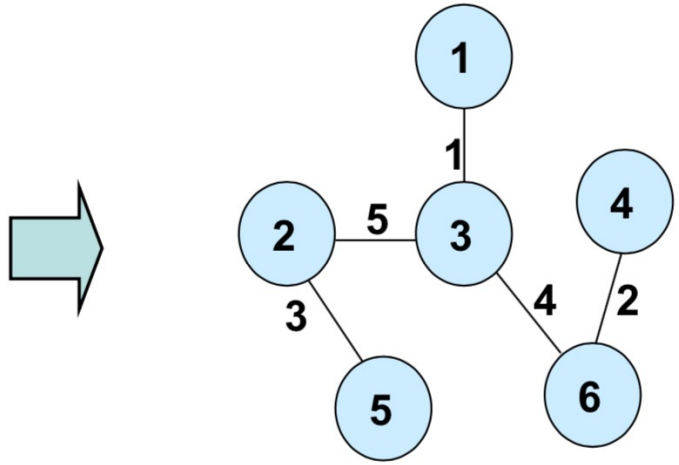
이 알고리즘은 쉽게 작성할 수 있다는 장점이 있지만, 3중 반복문을 돌리다보니 O(V^3)으로 상당히 비효율적이라는 단점이 있습니다. 따라서 정점의 수가 많지 않거나 모든 쌍에 대한 최단거리를 구해야만 할 때 사용하는 것이 좋고, 정점의 수가 많아진다면 필요한 지점들에 대해서만 다익스트라를 돌려서 해결하는 것이 좋습니다.

https://data-scientist-han.tistory.com/103

최소신장트리 구하기 : 크루스칼 vs. 프림



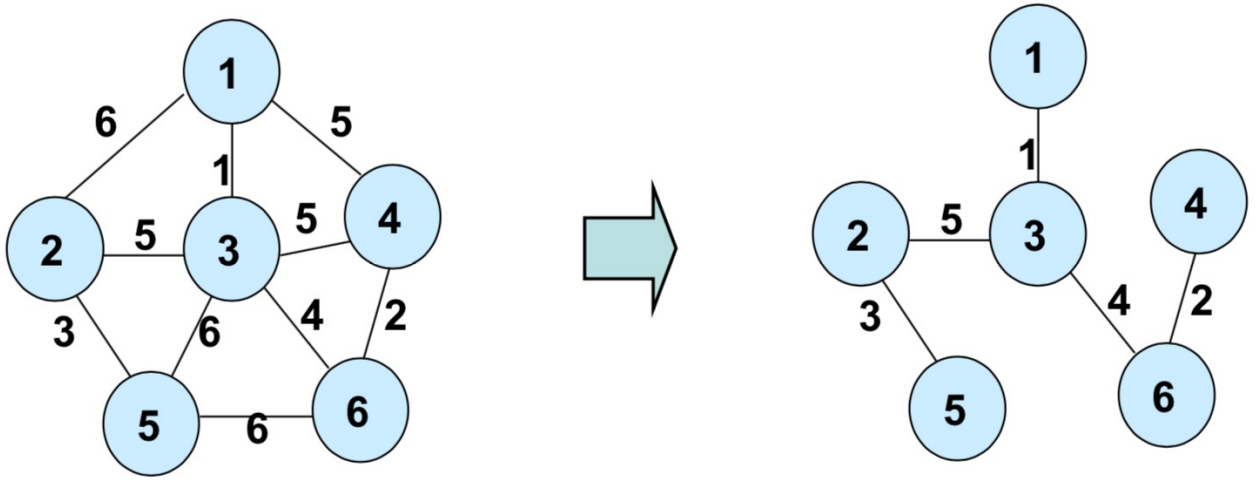




* 서로소 집합 (Disjoint Sets Algorithm)
→ Union-find 알고리즘
- 비연결 그래프 판별
- 사이클 판별

- 신장 트리 : N개의 정점에 N-1개의 간선이 존재하는 그래프
  - 사이클이 존재하지 않고,
  - 모든 정점들이 연결되어 있음.
- 최소신장트리(Minimum Spanning Tree, MST) : 그래프에서 최소한의 비용을 사용해 신장트리를 만드는 것.

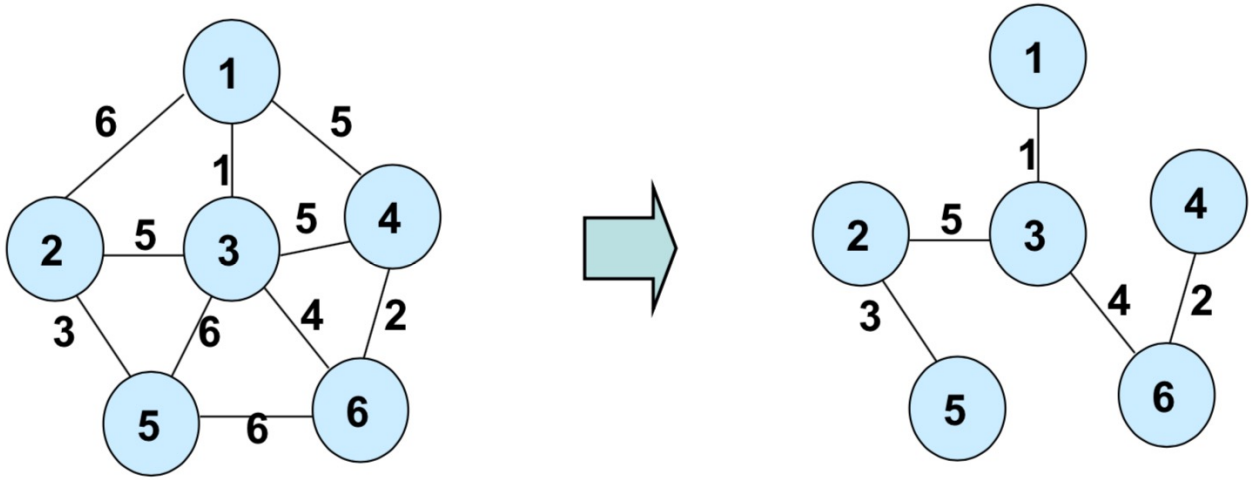
크루스칼 vs. 프림



- MST는 가중치의 합을 최소로 하는 Spanning Tree이니, 가중치가 작은 간선부터 고르는 것 (같은 가중치를 갖는다면 아무 간선이나 선택)
- 다만, 간선 선택시에 사이클이 생겨서는 안됨 (트리의 성질 때문에)

  → 사이클을 생성하지 않게 하기 위해, union-find 를 통해 사이클이 생기는지 확인

크루스칼 vs. **프림**



- **전체에서 간선을 선택하는 크루스칼과 반대로, 프림 알고리즘은 한 지점에서 시작하여 점점 확장을 진행하는 방법**

1. 정점 하나를 골라, MST를 구성
2. 해당 정점에 연결된 간선 중 가중치가 가장 작은 간선을 선택하여 MST 를 구성.
3. MST 를 구성하는 정점들에 연결된 간선 중 가중치가 가장 작은 간선을 선택하여 MST 를 구성.

크루스칼과 마찬가지로 사이클이 생성되지 않도록 주의

그래프

그래프 구현, 위상정렬, 최단거리 구하기(다익스트라, 플로이드), 최소 신장 트리 구하기(크루스칼, 프림)

패기반 이은비