10장 - 그래프 이론

그래프

그래프란 노드와 노드 사이에 연결된 간선의 정보를 가지고 있는 자료구조를 의미한다. 문제를 접했을 때 '서로 다른 개체(혹은 객체)가 연결되어 있다' 는 얘기를 들으면 가장 먼저 떠올려야 하는 알고리즘이다.

그래프의 구현 방법은 아래와 같이 2가지로 나뉜다.

- 인접 행렬 : 2차원 배열 사용
- 인접 리스트 : 리스트를 사용하는 방식

서로소 집합

서로소 집합 자료구조를 설명하려면 서로소 집합 개념이 필요하다. 수학에서 서로소 집합이란 **공통 원소가 없는 두 집합**을 의미한다. 예를 들어 집합 [1, 2]와 집합 [3, 4]는 서로소 관계이다. 서로소 집합 자료구조란 **서로소 부분 집합들로 나누어진 원소들의 데이터를 처리하기 위한 자료구조** 이다. 서로소 집합 자료구조는 union과 find 이 2개의 연산으로 조작할 수 있다.

union(합집합) 연산은 2개의 원소가 포함된 집합을 하나로 합치는 연산이다. find(찾기) 연산은 특정한 원소가 속한 집합이 어떤 집합인지 알려주는 연산이다.

서로소 집합 자료구조

보통 트리구조를 이용해 집합을 표현한다. 서로소 집합 정보(합집합 연산)가 주어졌을 때 트리 자료구조를 이용해 집합을 표현하는 서로소 집합 계산 알고리즘은 다음과 같다.

1. union 연산을 확인해, 서로 연결된 두 노드 A, B를 확인한다.
    a. A와 B의 루트 노드 A', B' 를 찾는다.
    b. A'를 B'의 부모 노드로 설정한다.
2. 모든 union 연산을 처리할 때까지 1번 과정을 반복한다.

실제 구현할 때는 A'과 B' 중 더 번호가 작은 원소가 부모 노드가 되도록 구현한다.

기본적인 서로소 집합 알고리즘 소스코드

```
# 특정 원소가 속한 집합 찾기
def find_parent(parent, x):
  if parent[x] != x:
    return find_parent(parent, parent[x])

  return x

# 두 원소가 속한 집합 합치기
def union_parent(parent, a, b):
  a = find_parent(parent, a)
  b = find_parent(parent, b)

  if a < b:
    parent[b] = a
  else:
    parent[a] = b

# 노드의 개수와 간선(union 연산)의 개수 입력 받기
v, e = map(int, input().split())
parent = [0] * (v + 1)

# 부모 테이블상에서, 부모를 자기 자신으로 초기화
for i in range(1, v + 1):
  parent[i] = i

# union 연산
for i in range(e):
  a, b = map(int, input().split())
  union_parent(parent, a, b)

# 각 원소가 속한 집합 출력
for i in range(1, v + 1):
  print(find_parent(parent, i), end=' ')
```

```
print()

# 부모 테이블 내용 출력
for i in range(1, v + 1):
  print(parent[i], end=' ')
```

하지만 find_parent 함수가 매우 비효율적이기 때문에 경로 압축 기법을 이용해 최적화가 가능하다.

```
def find_parent(parent, x):
  if parent[x] != x:
    parent[x] = find_parent(parent, parent[x])
  return parent[x]
```

서로소 집합을 활용한 사이클 판별

서로소 집합은 다양한 알고리즘에서 사용되며, 특히 무방향 그래프 내에서의 사이클을 판별할 때 사용할 수 있다. 앞서 union 연산은 그래프에서의 간선으로 표현될 수 있다고 했다. 따라서 간선을 하나씩 확인하면서 두 노드가 포함되어 있는 집합을 합치는 과정을 반복하는 것만으로도 사이클을 판별할 수 있다.

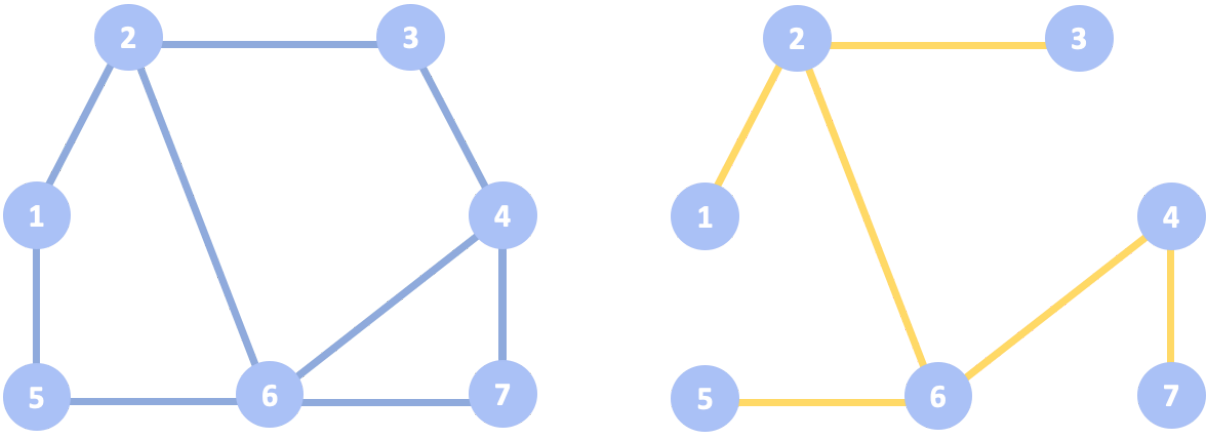
1. 각 간선을 확인하며 두 노드의 루트 노드를 확인한다.
    a. 루트 노드가 서로 다르다면 두 노드에 대해 union 연산을 수행한다.
    b. 루트 노드가 서로 같으면 사이클이 발생한 것이다.
2. 그래프에 포함된 모든 간선에 대해 1번 과정을 반복한다.

다음은 무방향 그래프에서의 사이클 판별 소스코드이다.

```
# 특정 원소가 속한 집합 찾기
def find_parent(parent, x):
  if parent[x] != x:
    return find_parent(parent, parent[x])

  return x

# 두 원소가 속한 집합 합치기
def union_parent(parent, a, b):
  a = find_parent(parent, a)
  b = find_parent(parent, b)

  if a < b:
    parent[b] = a
  else:
    parent[a] = b

v, e = map(int, input().split())
parent = [0] * (v + 1)

for i in range(1, v + 1):
  parent[i] = i

cycle = False

for i in range(e):
  a, b = map(int, input().split())

  if find_parent(parent, a) == find_parent(parent, b):
    cycle = True
    break

  else:
    union_parent(parent, a, b)
```

신장 트리

신장 트리란 하나의 그래프가 있을 때 모든 노드를 포함하면서 사이클이 존재하지 않는 부분 그래프를 의미한다.

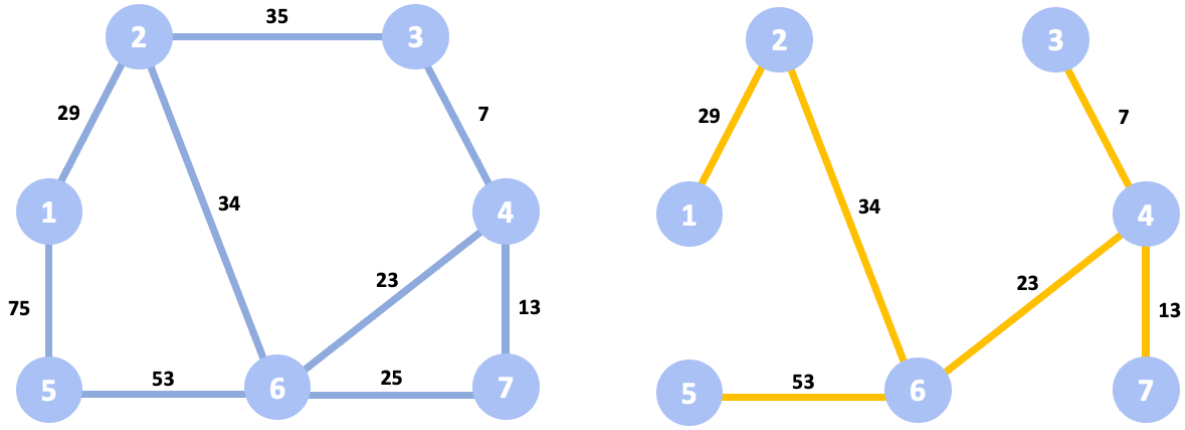


오른쪽 그래프를 신장 트리라고 한다.

크루스칼 알고리즘

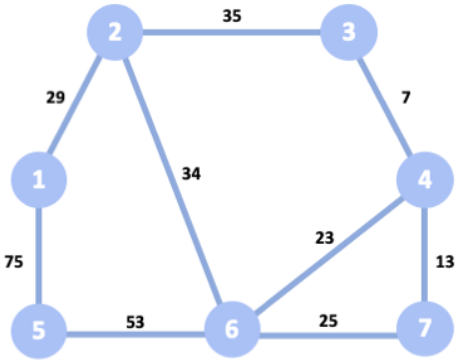
최소 신장 트리 알고리즘 중 하나로 최소한의 비용으로 신장트리를 찾는 알고리즘이다. 간선에 대해 정렬을 수행한 뒤, 가장 거리가 짧은 간선부터 집합에 포함시킨다. 이때 사이클을 발생시킬 수 있는 간선의 경우 집합에 포함시키지 않는다.

1. 간선 데이터를 비용에 따라 오름차순으로 정렬한다.
2. 간선을 하나씩 확인해 현재의 간선이 사이클을 발생시키는지 확인한다.
    a. 사이클이 발생하지 않는 경우 최소 신장 트리에 포함시킨다.
    b. 사이클이 발생하는 경우 최소 신장 트리에 포함시키지 않는다.
3. 모든 간선에 대해 2번을 반복한다.



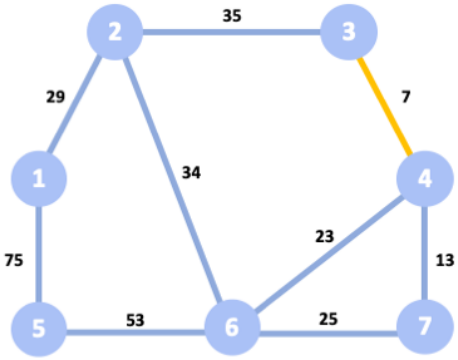
위 그림을 예시로 차근차근 알아가보려고 한다.

1. 그래프의 모든 간선 정보만 따로 빼내어 정렬을 수행한다. 원래는 간선 정보를 따로 리스트에 담아 정렬하지만 그림은 가독성을 위해 노드 데이터 순서에 따라 데이터를 나열했다.



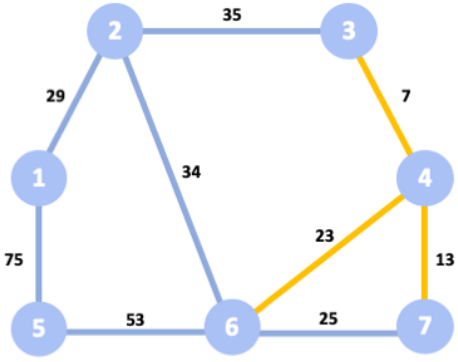
간선	(1, 2)	(1, 5)	(2, 3)	(2, 6)	(3, 4)	(4, 6)	(4, 7)	(5, 6)	(6, 7)
비용	29	75	35	34	7	23	13	53	25
순서									

2. 가장 짧은 노선인 (3, 4)를 선택해 union 함수를 수행한다. 즉, 노드 3과 노드 4를 동일한 집합에 속하도록 만든다.



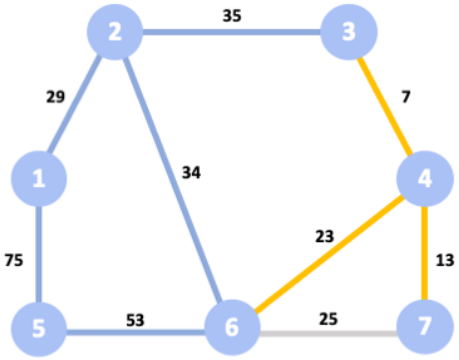
간선	(1, 2)	(1, 5)	(2, 3)	(2, 6)	(3, 4)	(4, 6)	(4, 7)	(5, 6)	(6, 7)
비용	29	75	35	34	7	23	13	53	25
순서					step 1				

3. 다음으로 짧은 간선 (4, 7)을 선택하고 union 함수를 호출, 다음으로 짧은 간선인 (4, 6)을 선택하고 union 함수를 호출한다.



간선	(1, 2)	(1, 5)	(2, 3)	(2, 6)	(3, 4)	(4, 6)	(4, 7)	(5, 6)	(6, 7)
비용	29	75	35	34	7	23	13	53	25
순서					step 1	step 3	step 2		

4. 다음으로 짧은 간선 (6, 7)을 선택한다. 노드 6과 7의 루트가 이미 동일한 집합에 포함되어 있으므로 신장 트리에 포함시키지 않는다.



간선	(1, 2)	(1, 5)	(2, 3)	(2, 6)	(3, 4)	(4, 6)	(4, 7)	(5, 6)	(6, 7)
비용	29	75	35	34	7	23	13	53	25
순서					step 1	step 3	step 2		step 4

이를 계속 반복하다보면 최소 신장 트리가 구해진다.

```
# 특정 원소가 속한 집합 찾기
def find_parent(parent, x):
  if parent[x] != x:
    parent[x] = find_parent(parent, parent[x])
  return parent[x]

# 두 원소가 속한 집합 합치기
def union_parent(parent, a, b):
  a = find_parent(parent, a)
  b = find_parent(parent, b)
  if a < b:
    parent[b] = a
  else:
    parent[a] = b

#노드의 개수와 간선(union 연산)의 개수 입력 받기
v, e = map(int, input().split())
parent = [0] * (v + 1)

# 모든 간선을 담을 리스트와 최종 비용을 담을 변수
edges = []
result = 0

# 모든 부모를 자기 자신으로 초기화
for i in range(1, v + 1):
  parent[i] = i

# 모든 간선 정보 입력
for _ in range(e):
  a, b, cost = map(int, input().split())
  edges.append((cost, a, b))
```

```
edges.sort()

for edge in edges:
  cost, a, b = edge

  # 사이클이 발생하지 않을 경우에만 집합에 포함
  if find_parent(parent, a) != find_parent(parent, b):
    union_parent(parent, a, b)
    result += cost

print(result)
```

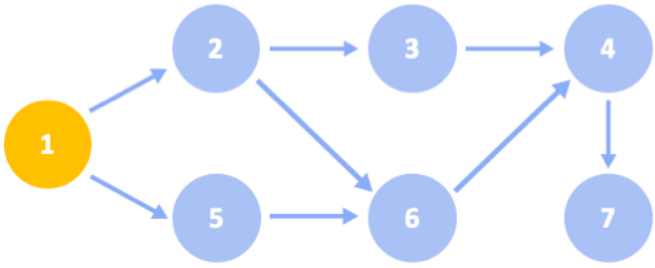
위상 정렬

방향 그래프의 모든 노드를 방향성에 거스르지 않도록 순서대로 나열하는 것이다. 진입 차수라는 개념을 쓰는데 진입 차수란 **특정 노드로 들어오는 간선의 개수**를 의미한다.

1. 진입 차수가 0인 노드를 큐에 넣는다.
2. 큐가 빌 때까지 다음의 과정을 반복한다.
    a. 큐에서 원소를 꺼내 해당 노드에서 출발하는 간선을 그래프에서 제거한다.
    b. 새롭게 진입차수가 0이 된 노드를 큐에 넣는다.

예시를 하나 보자.

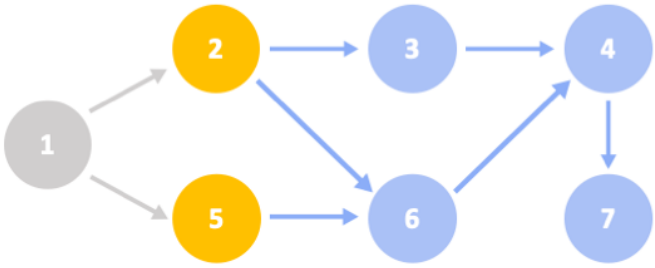
1. 초기 단계에는 진입 차수가 0인 노드가 1뿐이므로 큐에 노드 1을 삽입한다.



노드	1	2	3	4	5	6	7
진입차수	0	1	1	2	1	2	1

큐	노드 1

2. 큐에 들은 노드 1을 꺼내 연결된 간선들을 제거하고, 다시 진입차수가 0이 된 노드 2, 노드 5를 큐에 넣는다.



꺼낸 노드: 노드 1

노드	1	2	3	4	5	6	7
진입차수	0	0	1	2	0	2	1

큐	노드 2, 노드 5

3. 같은 과정을 반복한다. 과정을 반복하면서 큐에서 빠져나간 노드를 순서대로 출력하면 위상 정렬의 결과가 된다.

```
from collections import deque

#노드의 개수와 간선의 개수를 입력받기
v, e = map(int, input().split())
```

```
#모든 노드에 대한 진입차수는 0으로 초기화
indegree = [0] * (v + 1)
#각 노드에 연결된 간선 정보를 담기 위한 연결 리스트(그래프) 초기화
graph = [[] for i in range(v + 1)]

#뱡향 그래프의 모든 간선 정보를 입력받기
for _ in range(e):
    a, b = map(int, input().split())
    graph[a].append(b) #정점 A에서 B로 이동 가능
    #진입차수를 1 증가
    indegree[b] += 1

#위상 정렬 함수
def topology_sort():
    result = [] #알고리즘 수행 결과를 담을 리스트
    q = deque() #큐 기능을 위한 deque 라이브러리 사용

    #처음 시작할 때는 진입차수가 0인 노드를 큐에 삽입
    for i in range(1,  v + 1):
        if indegree[i] == 0:
            q.append(i)

    #큐가 빌 때까지 반복
    while q:
        #큐에서 원소 꺼내기
        now = q.popleft()
        result.append(now)
        #해당 원소와 연결된 노드들의 진입차수에서 1 빼기
        for i in graph[now]:
            indegree[i] -= 1
            #새롭게 진입차수가 0이 되는 노드를 큐에 삽입
            if indegree[i] == 0:
                q.append(i)

    #위상 정렬을 수행한 결과 출력
    for i in result:
        print(i, end=' ')

topology_sort()
```