알고리즘및실습

제9장 탐욕적 알고리즘 2부: 최소 신장 트리

1. 최소 신장 트리

최소 신장 트리 문제

- 입력. 무방향 가중치 그래프 $G = (V, E)$, $|V| = n$, $|E| = m$, G는 연결 그래프이고, 간선 e의 가중치 c_e에 대한 제한 없음
- 출력. 최소 신장 트리 $T \subseteq E$

신장 트리는 그래프의 부분 그래프로 그래프의 모든 노드가 포함되는 트리를 말한다. 모든 $v, w \in V$에 대해 v에서 w로 가는 경로가 T에 포함되어야 한다. 트리이므로 당영히 신장 트리에는 주기가 존재할 수 없다. 최소 신장 트리는 $\sum_{e \in T} c_e$가 최소가 되는 신장 트리이다. 최소 신장 트리는 컴퓨터 네트워크, 정보 검색, 기계 학습 등에서 활용된다.

최소 신장 트리는 Prim 알고리즘과 Kruskal 알고리즘을 이용하여 해결할 수 있다. 이 두 알고리즘은 모두 탐욕적 알고리즘이며, 시간 복잡도는 같지만 내부적으로 사용하는 자료구조가 다르다. 이 장에서는 이 두 알고리즘을 자세히 살펴본다.

2. Prim 알고리즘

알고리즘 9.1 Prim 알고리즘

```
1: function PRIM(G)
2:     X := {s}                          ▷ s ∈ V 인 아무 노드나 시작 노드로 사용할 수 있음
3:     T := []
4:     while X ≠ V do
5:         e := min{c_e | e = (u, v) ∈ G, w ∈ X, v ∉ X}
6:         T.PUSHBACK(e)
7:         X.ADD(v)
8:     return T
```

알고리즘 9.1에 제시된 Prim 알고리즘은 다익스트라 알고리즘과 매우 유사하다. 다익스트라 알고리즘은 매 반복마다 다익스트라 점수가 가장 작은 것을 선택한다. Prim 알고리즘은 이보다 더 간단하다. Prim은 경계를 넘는 간선 중 가중치가 가장 적은 간선을 선택하면 된다. 따라서 다익스트라 알고리즘과 마찬가지로 우선순위 큐를 이용하면 쉽게 구현할 수 있다. 우선순위 큐를 이용하여 **`while`** 문 내부를 어떻게 구현할 수 있는지는 조금 뒤로 미루고, 탐욕적 알고리즘이기 때문에 이 알고리즘의 정확성부터 증명해 보자.

2.1 Prim 알고리즘의 정확성

Prim 알고리즘의 정확성은 여러 가지 방법을 이용하여 증명할 수 있다. 이 교재에서는 그래프의 컷(cut) 개념을 이용하여 증명한다. 그래프에서 컷이란 그래프의 노드 집합을 2개의 공집합이 아닌 집합으로 분할하는 것을 말한다. n개의 노드로 구성된 그래프에는 총 $2^{n-1}-1$개의 컷이 존재한다. n개의 원소로 구성된 집합의 부분 집합 수는 2^n이다. 이 부분집합과 그것의 여집합은 컷을 형성하게 된다. 이때 컷을 구성하는 집합은 공집합이 될 수 없으므로 컷의 한 쪽 집합이 될 수 있는 부분 집합은 총 2^n-2개 존재한다. 컷의 두 집합의 순서는 중요하지 않으므로 총 컷의 수는 $(2^n-2)/2 = 2^{n-1}-1$이다.

컷과 관련하여 다음과 같은 보조 정리와 따름 정리를 이용하여 Prim 알고리즘의 정확성을 증명할 것이다. 이들 보조정리를 여기서 증명하지는 않는다.

보조정리 9.1 (빈 컷 보조 정리)**.** 컷을 구성하는 두 집합을 연결하는 간선이 없는 컷이 존재할 필요충분조건은 이 그래프는 연결 그래프가 아니다.

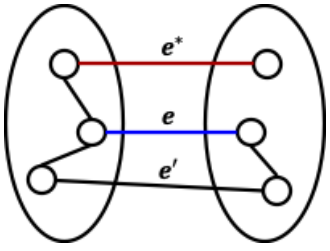
보조정리 9.2 (이중 횡단 보조 정리)**.** 그래프에 주기 $C \subseteq E$가 있고, 이 주기에 있는 간선 $e \in C$가 컷을 구성하는 두 집합을 건너가는 간선이면 이 두 집합을 건너가는 또 다른 간선 $e'(\neq e) \in C$이 존재한다.

따름정리 9.1 (단일 횡단 따름 정리)**.** 간선 e가 컷을 구성하는 두 집합을 건너가는 유일한 간선이면 이 간선은 어떤 주기에도 속하지 않는다.

보조 정리 9.1에 의하면 그래프가 연결 그래프일 때 임의의 컷을 선택하면 이 컷을 구성하는 두 집합을 건너가는 간선은 반드시 존재한다. 보조 정리 9.2에 의하면 컷을 구성하는 두 집합을 건너가는 간선이 주기에 포함되어 있으면 이 간선 외에 반드시 이 주기에 포함된 두 집합을 건너가는 또 다른 간선이 존재한다. 이 때문에 컷을 구성하는 두 집합을 건너가는 간선이 유일하면 이 간선은 절대 주기를 포함할 수 없다. 신장 트리는 트리이므로 신장 트리를 찾는 알고리즘은 주기를 형성하는 간선을 절대 선택하지 않아야 한다.

보조정리 9.3. 컷 특성 컷을 구성하는 두 집합 A와 B가 주어졌을 때 e가 이 컷을 건너가는 최소 가중치 간선이면 이 간선은 반드시 G의 최소 신장 트리 T^*에 포함되어야 한다.

증명 $e \in E$가 컷을 구성하는 두 집합 A와 B를 건너가는 최소 가중치 간선이지만 MST T^*에 포함되어 있지 않다고 가정하자. T^*에서 두 집합 A와 B를 건너가는 간선이 반드시 존재해야 한다. 이와 같은 간선이 없다면 T^*는 신장 트리가 될 수 없다. T^*에서 두 집합 A와 B를 건너가는 간선을 e^*라 할 때 e^*와 e를 교체한 그래프 T'이 신장 트리가 되면 T^*가 신장 트리라는 것은 모순이 된다. 하지만 e^*와 e를 교체한 T'이 무조건 신장 트리가 되지 않는다. 그러면 아닌 경우만 생각하여 보자. 이 경우에는 두 집합 A와 B를 건너가는 간선이 3개 이상 존재한다는 것을 의미한다. 교체하였는데 신장 트리가 안 된다는 것은 T^*를 구성하는 A와 B가 연결 그래프가 아니라는 것이다.



<그림 9.1> 컷 특성에 의해 컷을 건너가는 최소 간선은 최소신장트리에 포함되어야 함

그림 9.1의 형태라는 것을 의미한다. 따라서 e^* 외에 A와 B를 건너는 또 다른 간선 $e'(\neq e)$이 있다는 것을 말한다. 가정에 의해 $e \notin T^*$이므로 $c_{e'} > c_e$이다. 이 경우 e^*와 e'를 교체한 T'은 T^*와 노드 집합은 같으며, 간선 수도 같으므로 T'은 신장 트리이며, T'의 가중치 합은 T^*의 가중치 합보다 작으므로 T^*가 MST라는 조건에 모순이 된다. 따라서 이 보조 정리는 성립한다. □

정리 9.1. Prim 알고리즘은 정확하다.

증명 Prim 알고리즘의 정확성은 두 부분으로 나누어 증명한다. 먼저 이 알고리즘이 신장 트리를 출력한다는 것을 증명한다. 이를 위해 알고리즘은 항상 종료하며, 알고리즘의 출력한 간선의 노드 집합은 V와 같다는 것을 증명한다. 알고리즘 9.1은 X가 V가 되면 종료한다. 따라서 알고리즘이 종료하면 선택한 간선의 노드 집합은 V와 같아진다. 이 알고리즘의 **while** 문은 매 반복마다 X에서 $V - X$로 구성된 컷을 건너가는 간선을 선택하고, 간선의 노드 중 $V - X$에 있는 노드를 X에 추가한다. 이 알고리즘의 입력 그래프 G는 연결 그래프이므로 보조 정리 9.1에 의해 매 반복에서 만나는 컷마다 그 컷을 구성하는 두 집합을 건너가는 간선이 존재한다. 따라서 매 반복마다 X 집합이 1 증가하므로 궁극에 X는 V가 되고 알고리즘은 종료한다. 이 알고리즘에서 추가하는 간선은 그 반복에서 형성된 컷을 구성하는 두 집합을 건너가는 첫 번째 간선이자 유일 간선이므로 따름 정리 9.1에 의해 이 간선은 주기를 형성하지 않는다. Prim 알고리즘은 매 번 보조 정리 9.3를 만족하는 간선을 선택하므로 알고리즘 9.1이 출력하는 신장 트리는 최소 신장 트리이다. □

2.2 우선순위 큐를 이용한 Prim 알고리즘

알고리즘 9.2 우선순위 큐 기반 Prim 알고리즘

```
1:  function PRIM(G)
2:      X := {s}
3:      T := []
4:      score := [∞] × n
5:      H := empty min heap
6:      for all v ∈ V − {s} do
7:          if (s, v) ∈ E then score[v] := G[s][v]
8:          H.PUT(score[v], (s, v))
9:      while not H.EMPTY() do
10:         _, (v, u) := H.EXTRACTMIN()
11:         X.ADD(u)
12:         T.PUSHBACK((v, u))
13:         for all (u, w) ∈ E and u ∈ V − X do
14:             if G[u][w] < score[w] then
15:                 H.DELETE((_, w))
16:                 score[w] := G[u][w]
17:                 H.PUT(score[w], (u, w))
18:     return T
```

Prim 알고리즘은 매 번 현재 컷을 건너가는 간선 중 가중치가 가장 최소인 간선을 선택해야 한다. 따라서 이 알고리즘도 다익스트라처럼 우선순위 큐를 이용하면 효과적으로 구현할 수 있다. 우선순위 큐를 이용한 Prim 알고리즘은 알고리즘 9.2와 같다. 이 알고리즘도 다익스트라와 마찬가지로 우선순위 큐가 삭제 연산을 제공하지 않기 때문에 실제는 게으른 다익스트라처럼 우선순위 큐의 크기는 $|V - X|$보다 커지며, 10번째 줄에서 **extractMin**을 수행한 후에 u가 이미 X에 있는지 여부를 검사해야 한다. 그럼에도 불구하고 주어진 알고리즘을 기준으로 Prim 알고리즘의 시간 복잡도를 분석하여 보자.

while 문 이전에 X 집합과 **score** 배열을 생성 및 초기화하는 비용, 우선순위 큐에 $n-1$개 데이터를 삽입하는 비용이 필요하다. 가장 많은 비용이 소요되는 부분은 우선순위 큐에 데이터를 삽입하는 비용이므로 **while** 문 이전 비용은 $O(n \log n)$이다. **while** 문은 총 $n-1$번 반복되며, 매 반복마다 한번의 **extractMin** 비용이 필요하다. 이 비용은 $O(n \log n)$이다. **while** 문 내에 **for** 문은 그래프를 인접 행렬로 표현하였을 때와 인접 리스트로 표현하였을 때 반복하는 횟수는 다르다. 하지만 그래프 표현 방법과 무관하게 **for** 문 내의 **if** 문이 **true**가 되어야 우선순위 큐 연산이 수행된다. 이때 두 번의 우선순위 큐 연산이 수행된다. 간선의 수는 m이므로 최대 $2m$번 우선순위 큐 연산이 일어날 수 있다. 따라서 **for** 문에서 우선순위 큐 연산의 비용은 $O(m \log n)$이며, 반복 횟수에 의한 비용보다 이 비용이 크다. 따라서 전체 시간 복잡도는 $O((m+n) \log n)$이다.

Prim 알고리즘의 공간 복잡도도 알고리즘 9.2를 이용하여 분석하여 보자. 그래프를 인접 행렬로 표현한다고 가정하면 인접 행렬이 차지하는 공간은 $O(m+n)$이고, 집합 X를 **bool** 배열로 구현한다고 가정하면 X가 차지하는 공간은 $O(n)$이다. 또 우선순위 큐의 공간 복잡도는 $O(n)$이며, T는 $O(n)$이다. 따라서 전체 공간 복잡도는 $O(m+n)$이다.

3. Kruskal 알고리즘

이전 절에서 살펴본 Prim 알고리즘은 시작 노드를 선택한 후에 경계를 넘는 가중치가 최소인 간선을 반복적으로 선택한다. 따라서 계속 트리를 만들어 가는 형태이며, 항상 연결되어 있다. 또 경계를 넘는 간선을 하나씩 선택하기 때문에 컷의 특성 때문에 주기를 형성하는 간선을 선택할 수 없다. Kruskal 알고리즘도 매 번 하나의 간선을 선택하여 신장 트리를 만들어 가는데, Kruskal 알고리즘은 지금까지 선택하지 않은 간선 중 가장 가중치가 작은 간선을 선택한다. 이것의 선택은 모든 간선을 가중치를 기준으로 정렬하면 쉽게 다음에 선택할 간선을 결정할 수 있다. 하지만 다음으로 가중치가 가장 작은 간선을 선택하였을 때 주기를 형성할 수 있다. 따라서 선택할 때 해당 간선이 주기를 형성하는지 여부를 판단할 수 있어야 한다. 주기가 있는지 여부를 효과적으로 판단하는 방법이 Kruskal 알고리즘의 핵심이다. Kruskal 알고리즘의 기본 골격은 알고리즘 9.3와 같다.

알고리즘 9.3 Kruskal 알고리즘

```
1: function KRUSKAL(G)
2:     SORT(E)                                  ▷ 가중치를 기준으로 오름차순으로 정렬
3:     T := []
4:     for i := 1 to m do
5:         if not HASCYCLE(T ∪ E[i]) then T.ADD(E[i])
6:         if LEN(T) == n − 1 then break
7:     return T
```

3.1 Kruskal 알고리즘의 정확성

Kruskal 알고리즘에서 간선을 선택하였을 때 주기를 형성하는지 여부를 어떻게 판단하는지 살펴보기 전에 알고리즘의 정확성부터 증명하여 보자.

정리 9.2. Kruskal 알고리즘은 최소 신장 트리를 출력하여 준다.

증명 이 증명은 두 가지 부분으로 나누어 증명한다. 첫 번째 부분은 Kruskal 알고리즘의 출력 T^*가 신장 트리임을 증명한다.

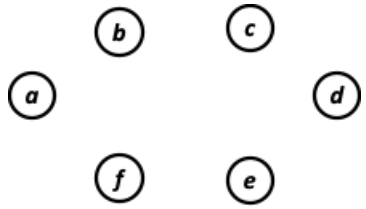
T^*에 간선을 추가할 때 주기가 있는지 여부를 확인하기 때문에 최종 T^*에는 주기가 없다. 주기가 없고, T^*의 크기가 $n-1$이고, 연결되어 있으면 신장 트리이다. 모든 컷을 건너가는 간선을 T^*가 포함하고 있으면 T^*는 연결 그래프이다. 이를 증명하기 위해 임의의 컷 (A, B)를 생각하여 보자. G는 연결 그래프이므로 이 컷을 건너가는 간선은 반드시 존재한다. Kruskal 알고리즘은 모든 간선을 차례로 검사하므로 이 컷을 건너가는 간선을 궁극에는 만나게 되며, 처음으로 만나면 이 간선은 단일 횡단 따름 정리에 의해 주기를 형성하지 않으므로 Kruskal이 선택하게 된다.

이제 Kruskal이 출력한 신장 트리 T^*가 최소 신장 트리임을 증명하여 보자. Kruskal이 선택하는 간선이 컷 특성을 만족하면 T^*는 최소 신장 트리이다. Kruskal 알고리즘에서 간선 $e = (u, v)$를 추가한다고 하자. $T \cup \{e\}$는 주기가 없으므로 기존 T에 u에서 v로 연결되는 경로가 없다. 노드 u와 v를 분할하는 컷 (A, B)를 생각하여 보자. 주기가 없으므로 이 컷을 건너가는 간선은 기존 T에는 없다. 따라서 e는 알고리즘이 만난 이 컷을 건너가는 첫 간선이다. 알고리즘은 항상 가중치가 가장 작은 간선을 추가하므로 e는 이 컷을 건너가는 가중치가 가장 작은 간선이다. 따라서 컷 특성을 만족한다. □

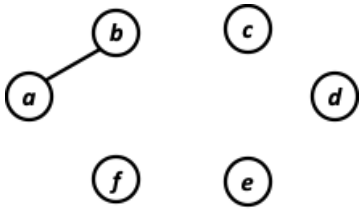
3.2 Union-find 자료구조

Kruskal을 단순하게 구현하면 **for** 문이 반복할 때마다 지금까지 고려하지 않은 가장 작은 가중치 간선을 T에 추가하였을 때 주기가 형성하는지 검사해야 한다. 추가하는 간선이 $e = (u, v)$이면 기존 T에 u에서 v로 가는 경로가 없어야 한다. u에서 v로 가는 경로의 존재 여부는 BFS나 DFS를 이용할 수 있다. 하지만 T는 간선 목록이므로 간선 목록을 이용하여 BFS나 DFS를 수행하기 번거롭다. T를 인접 리스트로 모델링하면 간선 수가 점진적으로 늘어나며, T의 간선 수가 $n-1$이 되면 알고리즘은 종료한다. 인접 리스트로 모델링하였을 때 BFS나 DFS의 시간 복잡도는 $O(m+n)$이며, BFS나 DFS를 수행할 때 T의 최대 간선 수는 $n-2$이므로 BFS나 DFS의 비용은 $O(n)$이며, 최대 m번 경로 탐색이 필요할 수 있으므로 전체 비용은 $O(mn)$이다. 이 방법을 사용하면 Prim이 더 우수한 알고리즘이 된다. 따라서 Kruskal이 Prim과 같은 성능을 제공하기 위해서는 더 효과적으로 주기 존재를 검사할 수 있어야 한다. 이를 위해 Kruskal이 사용하는 자료구조는 union-find 자료구조이다.

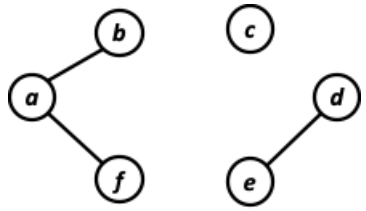
union-find 자료구조는 집합의 분할을 유지하며, 특정 요소가 어떤 분할에 있는지 찾아주는 **find**와 두 분할을 결합하여 주는 **union** 연산을 제공해 준다. 두 연산의 시간 복잡도는 모두 $O(\log n)$이다. 보통 초기에 모든 요소가 하나의 분할을 형성하며, 이 분할을 점진적으로 결합하여 문제를 해결할 수 있을 때 사용하는 자료구조이다. 이 자료구조를 다른 말로 disjoint-set이라 한다.



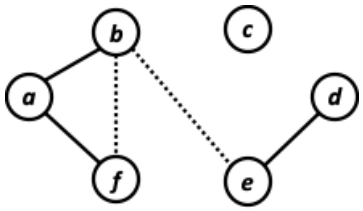
(1) 처음에는 모든 노드를 하나의 분할로 모델링함



(2) 간선을 선택하면 간선의 두 노드가 소속된 분할을 결합



(3) 3개의 분할로 나누어져 있는 모습



(4) b와 f를 연결하는 간선을 선택하면 주기를 형성함

<그림 9.2> Union-find 자료구조를 이용한 Kruskal 알고리즘

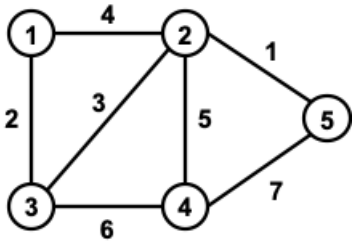
Kruskal 알고리즘은 매번 하나의 간선을 선택하며, 간선의 선택은 union-find 자료구조에서 분할의 결합으로 모델링할 수 있다. 예를 들어 그림 9.2의 (1)에서 a와 b를 연결하는 간선을 선택하면 a와 b가 소속된 두 분할을 결합하게 되며, 결합된 모습은 그림 9.2의 (2)와 같다. Kruskal은 매번 간선을 선택할 때 이 간선의 추가가 주기를 형성하는지 검사해야 한다. 선택한 간선의 시작과 끝 노드가 모두 같은 분할에 소속되어 있으면 이 간선의 선택은 주기를 형성하게 된다. 예를 들어 그림 9.2의 (3)에서 b와 f를 연결하는 간선을 선택하면 b와 f는 같은 분할에 있으므로 주기를 형성하게 된다. 하지만 서로 다른 분할에 있는 b와 e를 연결하는 간선을 선택하면 주기를 형성하지 않는다.

3.3 Union-find를 이용한 Kruskal의 구현

Union-find 자료구조를 이용한 Kruskal 알고리즘은 알고리즘 9.4와 같다. 앞서 언급한 바와 같이 union-find 자료구조에서 **find** 연산과 **union** 연산의 시간 복잡도는 $O(\log n)$이다. Union-find 자료구조는 각 분할마다 하나의 리더가 존재하며, 분할에 있는 나머지 요소는 이 리더에 대한 포인터를 유지한다. 하지만 나머지 모든 요소가 항상

알고리즘 9.4 Union-find 자료구조를 이용한 Kruskal 알고리즘

```
1: function KRUSKAL(G)
2:     T := []
3:     U.INIT(V)
4:     SORT(E)                                  ▷ 가중치를 기준으로 오름차순으로 정렬
5:     for i := 1 to m do
6:         if U.FIND(E[i].v) ≠ U.FIND(E[i].w) then
7:             T := T ∪ E[i]
8:             U.UNION(e[i].v, e[i].w)
9:     return T
```



<그림 9.3> Kruskal 알고리즘을 적용할 그래프 예

리더에 대한 포인터를 직접 유지하는 것이 아니라 간접적으로 유지할 수 있다. 즉, 요소가 유지하는 포인터를 따라가면 최종적으로는 리더를 가리키게 된다. 따라서 두 노드의 리더가 같으면 두 노드는 같은 분할에 소속된 노드가 된다.

Kruskal 알고리즘은 간선 $e = (u, v)$가 주어졌을 때 이 간선의 추가가 주기를 형성하는지 검사하는 것이 필요하다. 이 검사는 Union-find 자료구조에서 u와 v의 리더가 같은지 확인하면 된다. Union-find 자료구조에서 노드의 리더는 **find** 연산을 이용하며, 이 연산의 비용은 분할을 트리로 표현하였을 때 트리 높이에 의해 결정된다.

Union-find 자료구조에서 트리의 높이는 결합 방법에 의해 결정된다. Union-find 자료구조에서 두 분할을 어떻게 결합하는 것이 효과적일까? 한 분할의 모든 노드의 리더를 갱신하는 것은 비용이 많이 소요된다. 따라서 한 분할의 리더를 다른 분할의 리더의 자식으로 결합하면 가장 저렴한 비용(하나의 포인터 변경만 필요)으로 결합할 수 있다. 두 분할의 크기가 다를 때 크기가 작은 것을 큰 것에 결합하는 것이 높이가 변하는 노드의 수가 적기 때문에 더 효과적이다.

크기가 작은 분할을 큰 것에 결합하는 방법을 사용하면 특정 노드의 갱신은 최대 $O(\log n)$번 발생할 수 있다. 이것이 성립하는지 한 노드를 중심으로 생각하여 보자. 최초 작은 쪽에 소속된 노드를 생각하면 이 노드는 더 큰 분할과 결합할 때 높이가 갱신된다. 또 작은 쪽에 소속되어 결합할 때마다 분할의 크기는 2배 이상 확대된다. 따라서 작은 쪽에 소속되어 결합을 $\log n$번 수행하면 분할의 크기는 n이 되어, 더 이상 결합할 것이 없게 된다. 따라서 트리의 최대 높이는 $O(\log n)$이며, **find** 연산의 최악의 경우 비용도 $O(\log n)$이다.

그래프 노드의 이름이 1부터 n으로 주어졌다고 가정하였을 때, 실제 union-find 자료구조는 n 크기의 두 개의 정수 배열을 이용하여 구현한다. 첫 번째 배열은 각 노드의 부모 노드를 유지하는 배열이고, 두 번째 배열은 각 분할의 크기를 유지하는 배열이다. 5개 노드로 구성되어 있다고 가정하였을 때, union-find 자료구조의 초기 모습은 다음과 같다.

부모 노드 배열	1	2	3	4	5
분할 크기 배열	1	1	1	1	1

그림 9.3에 주어진 그래프에 대해 union-find 자료구조를 이용하여 최소 신장 트리를 구하여 보자. 첫 번째 선택하

게 되는 간선은 $(2, 5)$이다. 그러면 2와 5를 결합하여야 한다. 이때 크기가 같으면 리더 번호가 적은 것을 우선한다고 가정하자. 그러면 union-find 자료구조는 다음과 같이 변경된다.

부모 노드 배열	1	2	3	4	2
분할 크기 배열	1	2	1	1	0

그다음 선택하는 간선은 $(1, 3)$이고, 이 선택에 따라 union-find 자료구조는 다음과 같이 변경된다.

부모 노드 배열	1	2	1	4	2
분할 크기 배열	2	2	0	1	0

그다음 선택하는 간선은 $(2, 3)$이다. 2의 리더는 2이고, 3의 리더는 1이다. 또 두 리더의 크기가 같으므로 2를 1에 결합하면 union-find 자료구조는 다음과 같이 변경된다.

부모 노드 배열	1	1	1	4	2
분할 크기 배열	4	0	0	1	0

그다음 가중치가 작은 간선은 $(1, 2)$이다. 하지만 1의 리더는 1이고, 2의 리더도 1이므로 이 간선을 추가하면 주기가 형성된다. 따라서 이 간선은 선택하지 않고, 그다음으로 가중치가 작은 $(2, 4)$를 선택하게 되며, 이 간선은 주기를 형성하지 않으므로 union-find 자료구조는 최종적으로 다음과 같이 변경된다.

부모 노드 배열	1	1	1	1	2
분할 크기 배열	5	0	0	0	0

알고리즘 9.5 Union-find 자료구조의 find 연산

```
1: function FIND(v)
2:     if v = parent[v] then return v
3:     parent[v] := FIND(parent[v])
4:     return parent[v]
```

실제 **find** 연산의 알고리즘은 알고리즘 9.5와 같이 재귀적으로 구현한다. 결합할 때는 작은 분할의 리더 부모 링크만 갱신하지만 **find**의 인자로 사용된 노드가 리더를 부모 링크로 유지하고 있지 않으면 **find** 과정에서 리더로 바꾸어 준다. 이렇게 하면 같은 노드에 대해 **find**를 다시 호출되면 재귀 호출 없이 바로 리더 정보를 반환해 줄 수 있다.

3.4 Kruskal 알고리즘의 성능 분석

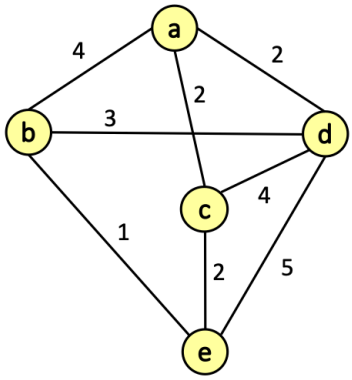
Kruskal 알고리즘은 반복문 전에 union-find 자료구조를 초기화해야 하고, 간선 목록을 정렬해야 한다. 초기화 비용은 $O(n)$이고, 정렬 비용은 $O(m \log m)$이다. 정렬 비용은 $O(m \log m)$이지만 $O(m \log n)$으로 바꾸어 표현할 수 있다. 반복문은 최대 m번 반복한다. 반복마다 두 번의 **find** 연산이 필요하다. **find** 연산의 시간 복잡도는 $O(\log n)$이므로 **find** 연산의 전체 비용은 $O(m \log n)$이다. **union** 연산은 총 $n - 1$번 발생한다. **union** 연산의 시간 복잡도는 $O(\log n)$이므로 **union** 연산의 전체 비용은 $O(n \log n)$이다. 반복문은 최대 m번 반복하므로 부가적으로 소요되는

비용은 $O(m)$이다. 그러므로 전체 비용은 $O((m+n)\log n)$이다. 즉, Prim 알고리즘은 시간 복잡도 측면에서 차이가 없다.

Kruskal은 그래프를 인접 행렬이나 인접 리스트로 표현할 필요가 없다. 간선 목록과 union-find 자료구조만 있으면 구현할 수 있다. 간선 목록의 공간 복잡도는 $O(m)$이고, union-find 자료구조의 공간 복잡도는 $O(n)$이며, 출력하는 신장 트리 T의 공간 복잡도도 $O(n)$이다. 따라서 전체 비용은 $O(m+n)$이다. Kruskal 공간 복잡도의 수준도 Prim과 같지만 생략된 계수를 고려하면 Kruskal이 더 효과적이다.

퀴즈

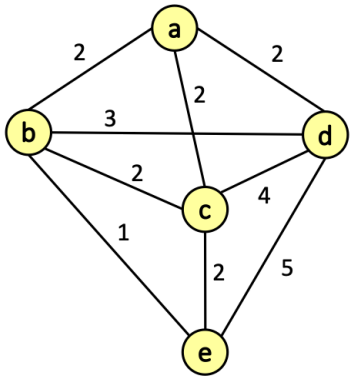
1. 주어진 다음 그래프에서



b 노드를 시작 노드로 하여 Prim 알고리즘을 수행하였을 때 세 번째로 선택하는 간선은?

① (b,e)
② (e,c)
③ (a,c)
④ (a,d)

2. 주어진 다음 그래프에 대한



Kruskal 알고리즘을 적용하기 위해 다음과 같이 정렬하였다.

$$(b,e),(a,b),(a,c),(a,d),(b,c),(c,e),(b,d),(c,d),(d,e)$$

그다음 union-find 자료구조를 다음과 같이 초기화하였다.

$$a:(a,1),b:(b,1),c:(c,1),d:(d,1),e:(e,1)$$

여기서 $a:(a,1)$이란 a의 부모는 a이고, 해당 분할의 크기는 1이라는 것을 나타낸다. 세 개의 간선을 선택한 후 union-find 자료구조의 모습은?
가정. 결합할 때 분할의 크기가 같으면 노드 이름이 작은 것을 리더로 선택한다.

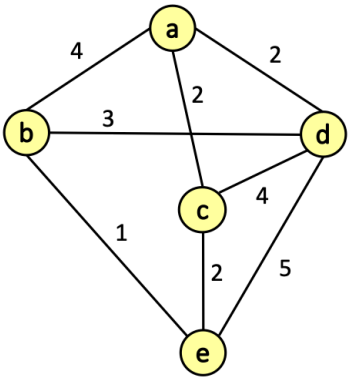
① $a:(a,1), b:(b,2), c:(c,1), d:(d,1), e:(b,0)$

② $a:(b,0), b:(b,3), c:(c,1), d:(d,1), e:(b,0))$

③ $a:(b,0), b:(b,4), c:(b,0), d:(d,1), e:(b,0)$

④ $a:(b,0), b:(b,4), c:(a,0), d:(d,1), e:(b,0)$

3. 주어진 그래프에서 Kruskal 알고리즘을 수행하였을 때 세 번째로 선택하는 간선은?



정렬 결과는 다음과 같다고 가정한다.

$$(b,e),(a,c),(a,d),(c,e),(b,d),(a,b),(c,d),(d,e)$$

① (a,d)

② (c,e)

③ (b,d)

④ (a,b)

연습문제

1. n개의 노드로 구성된 무방향 그래프가 주어진다. 이 그래프의 간선 수는 n이며, 이 중 하나의 간선을 제거하면 트리가 된다. 주어진 간선 중 어떤 간선을 제거하면 트리가 되는지 찾아라. 여러 개 간선이 답이 될 수 있다. 이 경우 입력 중 가장 뒤에 나타나는 간선을 출력하라.

2. n개의 (x,y) 좌표가 주어진다. 여기서 x와 y는 정수이다. 주어진 좌표로 구성되는 무방향 연결 그래프 중 모든 간선의 가중치 합이 최소가 되는 그래프의 가중치 합을 출력하라. 계산의 편리성을 위해 (x_1,y_1)과 (x_2,y_2)를 잇는 간선의 가중치는 $|x_1-x_2|+|y_1-y_2|$로 계산한다.

3. n개의 노드로 구성된 무방향 가중치 그래프가 주어진다. 이 그래프의 MST를 구하였을 때 반드시 포함해야 하는 간선이 있고, 그렇지 않은 간선이 있다. 반드시 포함해야 하는 간선을 제시하라.