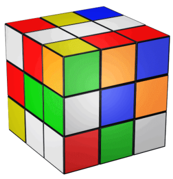
탐욕적 알고리즘: 최소신장트리

한국기술교육대학교 컴퓨터공학부 김상진



```
while(!morning){
    alcohol++
    dance++
} #party
```

```
while(!sleep){
    think++
    solve++
} #cse-mode
```

교육목표

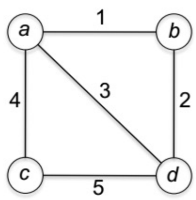
- 탐욕적 알고리즘: 최소신장트리(MST, Minimum Spanning Tree)
  - Prim 알고리즘
    - 빠른 구현: heap 자료구조
  - Kruskal 알고리즘
    - 빠른 구현: union-find 자료구조
- 참고. n개 노드로 구성된 완전 무방향 그래프에서 신장 트리의 개수는 n^{n-2}개 존재함

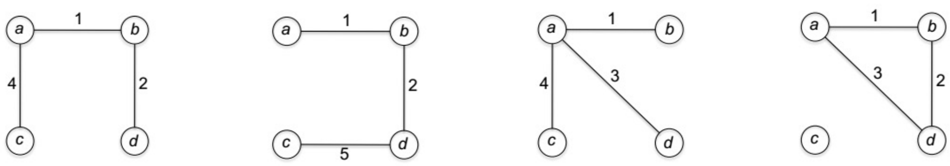
MST 문제

- **입력.** 무방향 가중치 그래프 $G = (V, E), |V| = n, |E| = m$
  - **가정.** 인접 리스트로 구현
  - **가정.** 음수 가중치 c_e 가능 (c_e: 간선 e의 가중치)
  - **가정.** 그래프는 연결 그래프임
    - DFS, BFS로 연결 여부 확인 가능
- **출력.** 최소신장트리 $T \subseteq E$
  - 신장트리: 그래프의 부분 그래프로 모든 노드가 포함된 트리
    - $\forall v, w \in V$에 대해 v에서 w로 가는 경로가 T에 포함되어야 함
    - 트리이므로 당연히 주기가 없어야 함
  - 최소신장트리: $\sum_{e \in T} c_e$가 최소가 되는 신장트리
- 응용) 컴퓨터 네트워크, 검색, 기계학습 등
- 2가지 탐욕적 기법: Prim 알고리즘, Kruskal 알고리즘
- **참고.** $n - 1 \leq m \leq n(n-1)/2$
  - 보통 m은 n보다 큼

최소신장트리

- 다음 그래프에서 최소신장트리는?





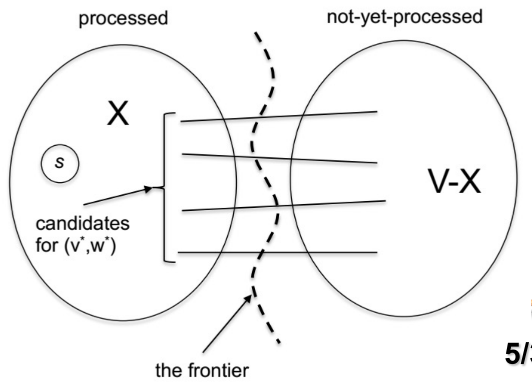
Prim 알고리즘 (1/2)

- 시간복잡도: $(m+n)\log n$
- 알고리즘:
  - 모든 노드가 연결될 때까지 가중치가 가장 작은 간선을 하나씩 추가
    - 이 간선의 추가로 주기가 생기면 안 됨

Prim(G)

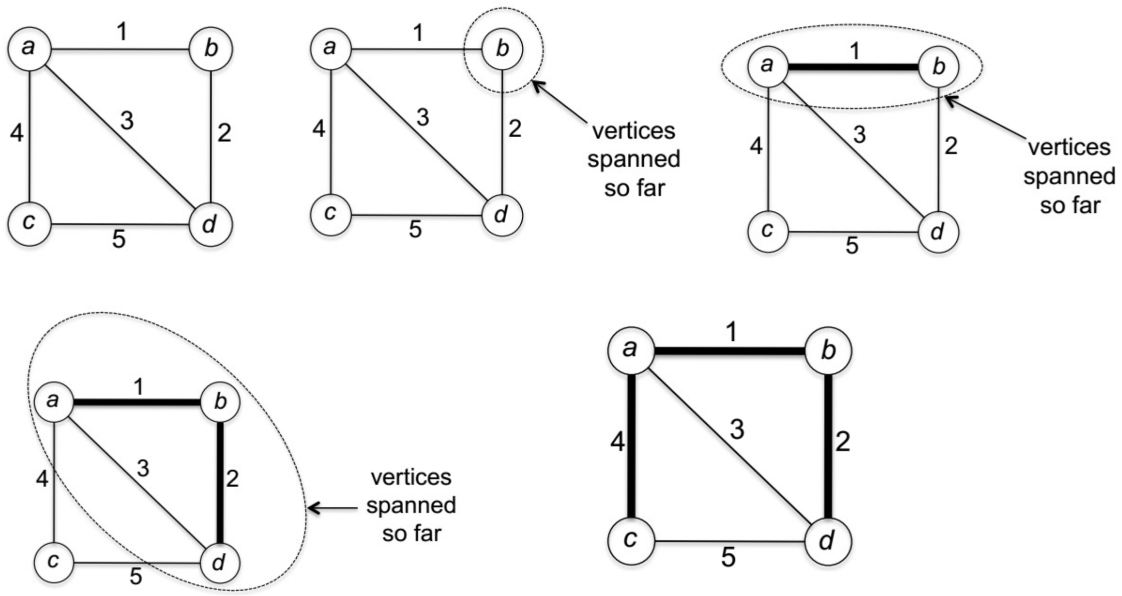
$X := \{s\}$ // 아무 노드나 시작 노드로 사용 가능
$T := []$
while $X \neq V$ **do**
    $e := (u, v)$ be the cheapest edge of G with $u \in X$, $v \notin X$
    add e to T
    add v to X

- 이와 같이 선택하면 왜 주기가 안 생길까?



  - 구현할 때 사용할 자료구조는?
- 다익스트라 알고리즘과 매우 유사함

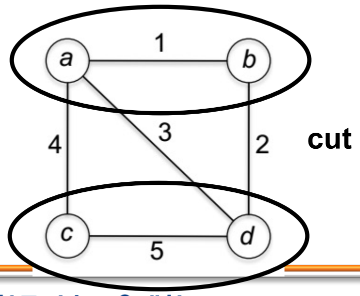
Prim 알고리즘 (2/2)

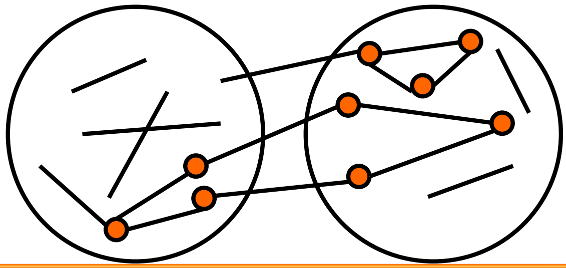


- 반복마다 간선이 하나 추가됨 (총 필요한 간선 수는 $n-1$)
- 반복마다 노드가 추가됨
- 반복마다 트리가 확장됨

Cut

- Cut: 그래프의 cut이란 V를 2개의 공집합이 아닌 집합으로의 분할
  - n개의 노드로 구성된 그래프에서 가능한 cut의 수는? $2^{n-1}-1$
- 보조정리 1. (empty cut lemma) cut을 구성하는 두 집합을 연결하는 간선이 없는 cut이 존재할 필요충분조건은 이 그래프는 연결 그래프가 아님
- 보조정리 2. (double crossing lemma) 그래프에 주기 $C \subseteq E$가 있고, 이 주기에 있는 간선 $e \in C$가 cut을 건너가는 간선이면 cut을 건너가는 간선 $e'(\neq e) \in C$이 있음
  - 주기가 cut을 건너갈 경우에는 cut을 건너가는 간선의 수는 짝수임
- 따름정리. (lonely cut corollary) e가 cut (A, B)를 건너가는 유일한 간선이면 이 간선은 어떤 주기에도 소속되지 않음

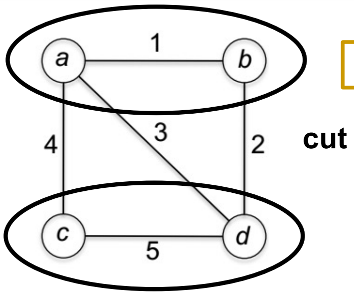




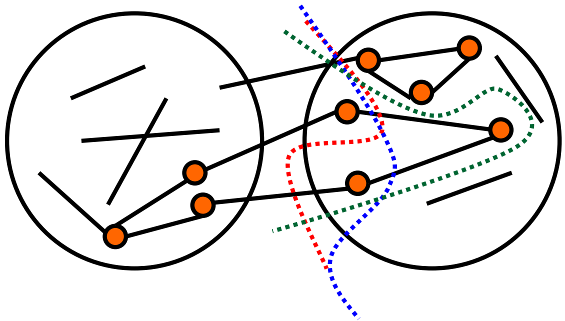
Prim 알고리즘의 정확성 증명 (1/4)

- 증명을 2부분으로 구성
  - Part 1. 이 알고리즘은 신장트리를 출력함
  - Part 2. 출력된 신장트리는 최소신장트리임
- Part 1)
  - 이 알고리즘은 항상 종료함 (모든 노드가 X에 포함됨)
    - 알고리즘의 반복문은 반복마다 X의 크기를 하나 증가함
      - 이 크기가 증가하지 않으면 반복이 종료하지 않을 수 있음
      - 현재 X와 $V - X$로 구성된 cut이 있을 때 이 cut을 건너가는 간선이 없으면 종료하지 않음
        - 연결 그래프이기 때문에 존재함 (보조정리 1)
  - 이 알고리즘은 T에 주기를 만들지 않음
    - 우리가 추가하는 간선은 그 반복에서 형성된 cut을 건너가는 첫 번째 간선이자 유일 간선이기 때문에 따름정리에 의해 주기가 만들어질 수 없음

Prim 알고리즘의 정확성 증명 (2/4)



cut을 지나는 첫 간선은 절대 주기를 만들지 않음



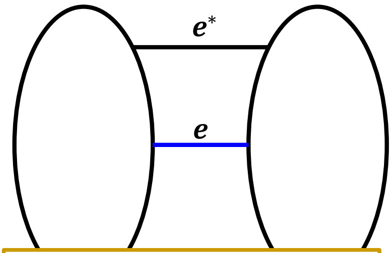
- 주기에 포함된 빨간색 간선을 선택하였다고 주기가 만들어지지 않음
- 해당 간선을 선택하면 cut 모양이 바뀜
- 주기에 포함된 간선을 하나씩 추가하다 보면 같은 노드로 들어가는 두 개의 간선이 만나게 됨
- 이 간선 중 하나의 간선을 선택하면 다른 간선은 알고리즘 특성 때문에 선택이 가능하지 않음

Prim 알고리즘의 정확성 증명 (3/4)

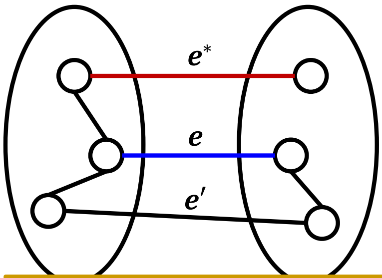
- Part 2)
  - cut 특성. Cut (A, B)가 주어졌을 때, e가 이 cut을 건너가는 최소 가중치 간선이면 이 간선은 반드시 G의 최소신장트리 T^*에 포함되어야 함
    - Prim은 이 간선을 선택함
    - cut 특성이 정확하면 Prim은 최소신장트리를 출력함
  - cut 특성의 증명) e가 cut (A, B)를 건너가는 최소 가중치 간선이지만 MST T^*에 포함되어 있지 않다고 가정하고 이것이 모순임을 증명함
  - T^*에서 cut (A, B)를 건너가는 간선이 반드시 존재해야 함
    - 없으면 T^*은 신장 트리가 될 수 없음 (A와 B가 연결되지 않음)
  - T^*에서 cut (A, B)를 건너가는 간선과 e를 교체하면 그 결과는 무조건 신장 트리가 되지 않음
    - 신장 트리가 되면 T^*보다 전체 가중치가 적음 (모순)
  - 결론. 항상 교체하여 신장 트리를 유지할 수 있는 간선이 존재하며, 이 간선과 교체하면 전체 가중치가 적어지기 때문에 e가 원래 포함되어 있지 않다는 것은 모순임

Prim 알고리즘의 정확성 증명 (4/4)

- 건너가는 간선과 무조건 교체하면 신장트리가 되지 않음
  - **경우 1.** 교체할 간선이 유일한 경우: 해당 간선과 교체하면 됨
  - **경우 2.** 교체할 간선이 여러 개이면 교체하였을 때 노드의 수가 변하지 않는 간선과 교체하면 됨

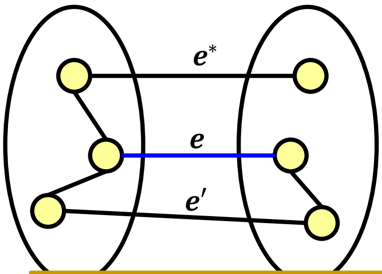


두 cut을 연결하는 간선이 2개이면 이 간선을 제외하고 다른 간선을 추가하면 신장트리가 유지됨
why? 각 cut 내부는 연결 그래프이어야 함



신장트리에 두 cut을 연결하는 간선이 여러 개이면 각 cut 내부 중 하나는 연결 그래프가 아님

하나를 제거하고 하나를 추가 하면 간선의 수는 변하지 않음



노드의 수가 변하면 주기가 발생하며, 이 주기에 포함된 간선이 하나 더 있음

노드의 수가 변하지 않으면 신장트리임

Prim 알고리즘의 시간복잡도

- 알고리즘

Prim(G)

$X \coloneqq \{s\}$ // 아무 노드나 시작 노드로 사용 가능
$T \coloneqq []$
while $X \neq V$ **do**
  $e \coloneqq (u, v)$ *be the cheapest edge of* G *with* $u \in X$, $v \notin X$
  add e *to* T
  add v *to* X

  - 직관적 구현
    - 반복문의 반복횟수: $n - 1$
    - 가장 가중치가 작은 건너가는 간선 찾기 비용: $O(m)$
      - 전체 비용: $O(mn)$
  - 힙(min heap) 사용. 어떻게?
    - **방법 1.** 간선들을 힙에 유지: $O(m\log m) = O(m\log n)$
    - **방법 2.** 노드들을 힙에 유지 ($V - X$를 유지)

왜 $m\log m$?
구현할 때 문제점은?

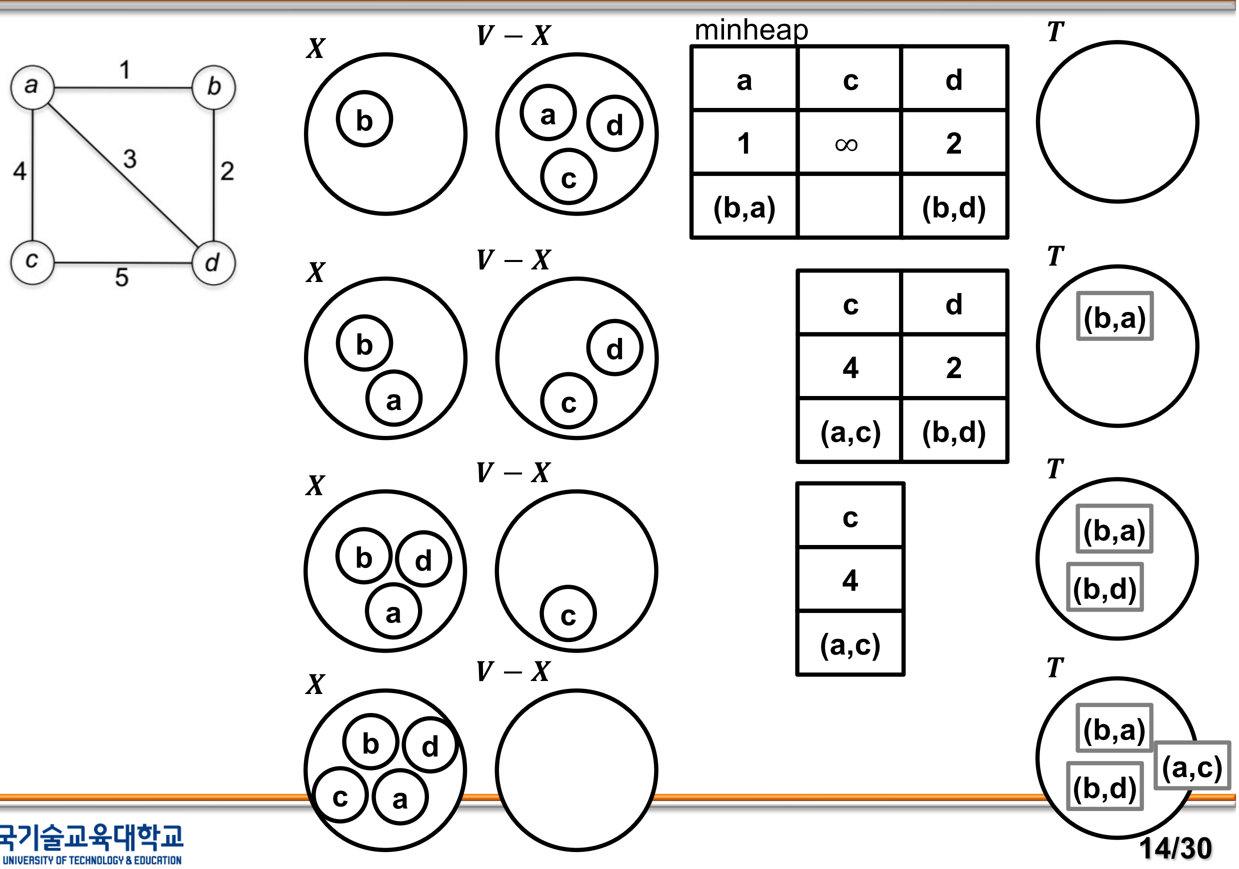
힙 기반 Prim 알고리즘

$X \coloneqq \{s\}$ // 아무 노드나 시작 노드로 사용 가능
$T \coloneqq []$
$score \coloneqq [\infty] \times n$
$H \coloneqq$ *empty min heap*
for all $v \in V - \{s\}$ **do**
  if $(s, v) \in E$ **then** $score[v] \coloneqq G[s][v]$
  $H.put(score[v], (s, v))$
while not H*.empty()* **do**
  $\_, (v, u) \coloneqq H.extractMin()$
  $X.add(u)$
  $T.pushback((v, u))$
  for all $(u, w) \in E$ **and** $w \in V - X$ **do**
    if $G[u][w] < score[w]$ **then**
      $H.delete((\_, w))$
      $score[v] \coloneqq G[u][w]$
      $H.put(score[w], (u, w))$
return T

s는 heap에 포함하지 않고 출발

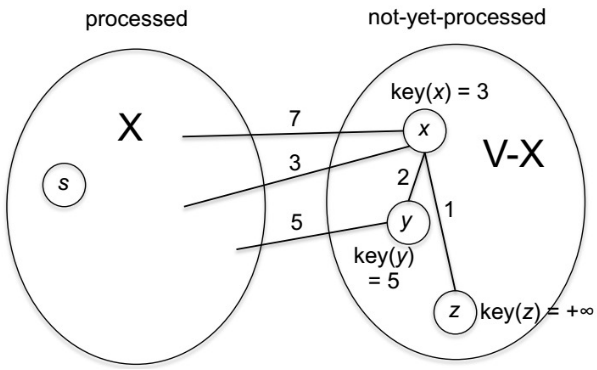
- X와 $V - X$는 어떻게 구현?
- 어떤 자료구조?
- 연결 그래프라 가정하면 T에 $n-1$개 edge가 추가되면 끝

delete는 어떻게 처리?
다익스트라와 마찬가지



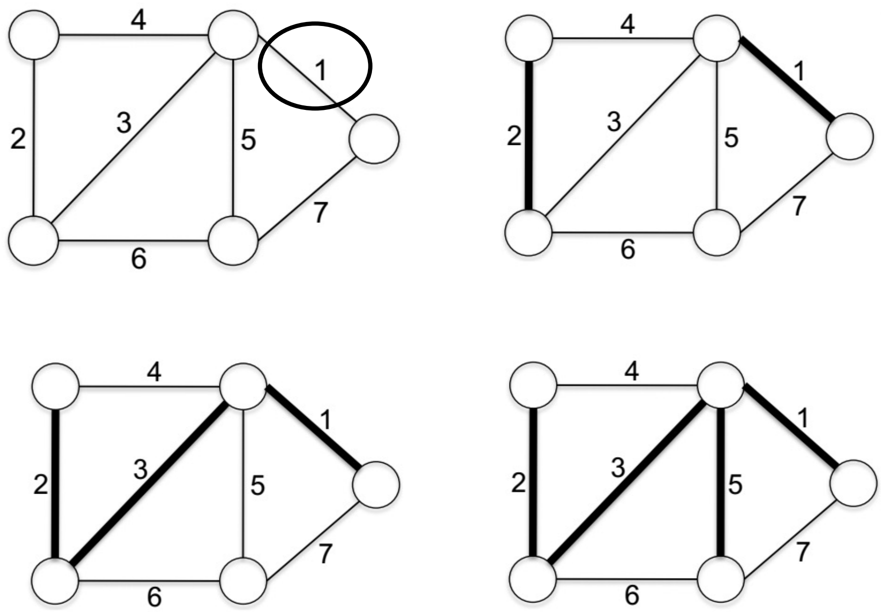
힙 기반 Prim 알고리즘의 시간복잡도

- 초기화 비용
  - 최초 비용: s를 제외한 다른 노드에서 s로 연결되는 간선이 존재하면 이 간선의 가중치가 키 값이 됨
  - $O(n\log n)$: $n-1$개의 heap insert



- 다음 반복에 X에 노드 a가 하나 추가됨: $n-1$번 반복 (extractMin: $O(n\log n)$)
  - 노드 a의 진출 간선마다 목적 노드가 $V-X$에 있는 간선에 대해 간선의 가중치와 기존 키 값과 비교하여 더 작으면 힙에서 기존 키 값을 삭제하고 새 키 값을 추가함. 2개의 힙 연산 필요
  - 간선은 최대 1번만 고려함: $O(m\log n)$
  - $O(n\log n + m\log n) = O((m+n)\log n)$
- 총: $O((m+n)\log n)$

Kruskal 알고리즘



Prim과 같은 수준의 알고리즘
Union-find라는 새 자료구조를 사용
(Prim은 heap)

Kruskal MST 알고리즘

- Pseudocode

Kruskal(G)
sort edges in order of increasing cost *// rename edges $e_1, e_2, \dots, e_m$ so that $c_1 < c_2 < \cdots < c_m$* $T \coloneqq []$ ***for*** $i \coloneqq 1$ ***to*** m ***do***   ***if*** $T \cup e_i$ ***has no cycles then add*** e_i ***to*** T ***return*** T

  - m번 반복할 필요는 없고 $n-1$ 개 간선을 T에 추가하면 종료할 수 있음
  - 주기가 없는지 검사를 어떻게 효과적으로?

Kruskal MST 알고리즘의 정확성

- 증명) T^*: Kruskal MST 알고리즘의 출력
  - Part 1) T^*는 신장트리임을 증명 (주기가 없고, 연결되어 있음)
    - 당연히 주기가 없음 (알고리즘에 의해)
    - 연결되어 있다는 것을 증명하기 위해 T^*는 가능한 모든 cut을 건너가는 간선을 포함하고 있다는 것을 증명함
    - 임의의 cut (A, B)를 생각하여 보자
      - G는 연결 그래프이므로 이 cut을 건너가는 간선은 반드시 존재함
      - 이 cut을 건너가는 간선이 이미 포함되어 있을 수 있고, 아직 포함되어 있지 않을 수 있음
      - 후자의 경우 이 cut을 건너가는 간선을 Kruskal 알고리즘이 처음 접하게 된 경우임. 이 간선은 주기를 만들지 않으므로 Kruskal이 선택할 수 있는 간선임
      - Kruskal은 모든 간선을 고려하기 때문에 이 cut를 건너가는 간선 중 가중치가 가장 작은 것이 결국에는 포함됨
    - 따라서 주기가 없고 연결되어 있으므로 T^*는 신장트리임

Kruskal MST 알고리즘의 정확성

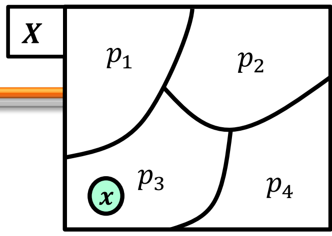
- Part 2) T^*는 최소신장트리임
  - T^*이 선택하는 간선은 cut 특성을 만족함 $\Longrightarrow$ MST
  - Why?
  - Kruskal 알고리즘에서 간선 $e = (u, v)$를 추가한다고 하자.
    - $T \cup \{e\}$는 주기가 없으므로 기존 T에 u에서 v로 연결되는 경로가 없음
    - u와 v를 분할하는 cut (A, B)를 생각하여 보자. 주기가 없으므로 이 cut을 건너가는 간선은 T에는 없음 (보조정리 2)
    - 따라서 e는 알고리즘이 만난 이 cut을 건너가는 첫 간선임
    - 알고리즘은 항상 가중치가 가장 작은 간선을 추가하므로 e는 이 cut을 건너가는 가중치가 가장 작은 간선임
    - 따라서 cut 특성을 만족함

> cut 특성. Cut (A, B)가 주어졌을 때, e가 이 cut을 건너가는 최소 가중치 간선이면 이 간선은 반드시 G의 최소신장트리 T^*에 포함되어야 함

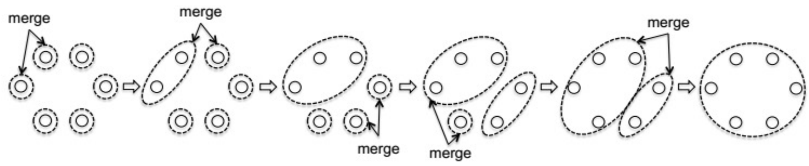
Kruskal MST 알고리즘의 시간복잡도

- Kruskal MST 알고리즘을 구현하는 단순 방법
  - 정렬: $O(m\log m) = O(m\log n)$
  - 반복문 비용: 반복횟수: m, 주기 여부 검사: $O(n)$
    - 주기를 어떻게? $e = (u, v)$를 추가하고자 하면 u에서 v로 가는 경로가 없어야 함
      - 경로 검색: BFS, DFS $\Longrightarrow O(m + n) \Longrightarrow O(n)$
        - 입력은 T이며, 이 트리의 간선 수는 최대 $n - 1$임
    - 전체: $O(m\log n) + O(mn) = O(mn)$
  - 주기 존재 여부를 더 효과적으로 검사할 수 있으면 전체 비용을 줄일 수 있음

Union-Find 자료구조



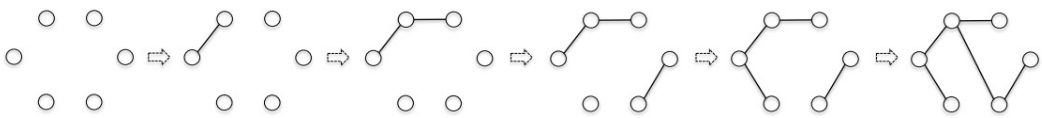
- Union-find 자료구조: 집합 X의 분할을 유지
  - Find(x): p_3
  - Union(p_i, p_j)



- 이 자료구조에서 분할이 합쳐질 수 있지만 다시 나누어지는 경우는 없음
- 보통 초기에 모든 요소가 하나의 분할을 형성함
  - 초기 비용: $O(n)$
- 두 연산의 시간 복잡도: $O(\log n)$
- 다른 말로 disjoint-set이라 함

Union-Find 자료구조

- Why UF is useful for Kruskal algorithm???
  - 이 자료구조를 이용하여 지금까지 연결된 분할을 유지하고, 간선을 추가할 때마다 union 연산을 하여 분할을 효과적으로 갱신할 수 있음



- 주어진 간선 $e = (u, v)$를 T에 추가하면 주기가 생기는지 여부를 효과적으로 알 수 있음
  - 지금까지 트리를 형성하는 노드의 집합이 있을 때, 이 집합에 새 간선을 추가하면 주기가 만들어짐 (더 이상 트리가 아님)
  - 서로 다른 분할에 있는 노드를 연결하는 간선은 주기를 만들지 않음

Kruskal MST using Union-Find

- 알고리즘

```
T := []
U.init(V)
sort(E)
for i := 1 to m do
    if U.find(E[i].v) ≠ find(E[i].w) then
        T.add(E[i])
        U.union(E[i].v, E[i].w )
return T
```

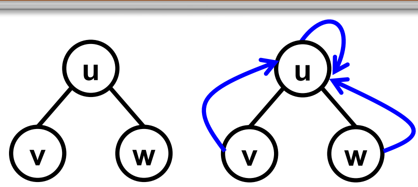
edge 목록은 어떻게 확보?

인접 리스트 자체를 활용하지 않음

Find와 Union의 시간복잡도가 $\log n$이면

	preprocessing	$O(n) + O(m \log n)$
	$2m$ Find operations	$O(m \log n)$
	$n-1$ Union operations	$O(n \log n)$
+	remaining bookkeeping	$O(m)$
	total	$O((m+n) \log n)$.

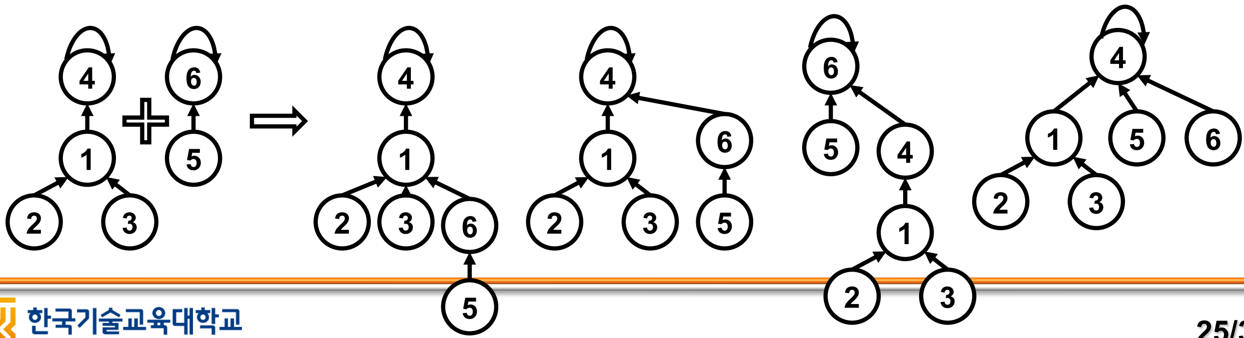
Union-Find을 이용한 Kruskal (1/3)



- 목적. 상수 비용으로 주기 찾기
- 같은 분할에 있는 요소를 하나의 트리로 유지
  - 이 요소 중 하나를 리더로 선정
  - 리더가 트리에 루트에 위치함
  - 일반 트리 구조와 달리 노드는 부모에 대한 포인터를 유지
    - 리더는 자신에 대한 포인터를 유지
  - 트리의 깊이를 줄이기 위해 부모가 아니라 각 요소는 이 리더에 대한 포인터를 유지하는 것이 목표임
  - 리더가 같으면 같은 분할에 소속된 트리임
    - 노드가 주어졌을 때 그 노드의 리더를 파악하는 것은 트리 높이에 의해 결정됨
  - 두 분할의 결합은 한 쪽 분할의 노드의 리더를 다른 분할의 리더로 바꾸면 됨

Union-Find을 이용한 Kruskal (2/3)

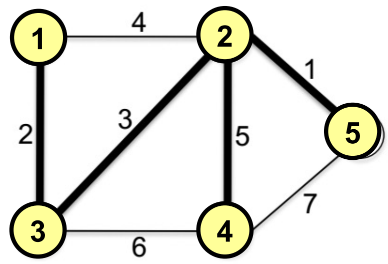
- 모든 노드는 해당 노드로만 구성된 분할에 소속된 상태에서 시작함
- 신장 트리에 포함할 간선 $e = (u, v)$을 선택하면 u와 v가 소속된 분할을 결합함
- 지금까지 만든 신장 트리에 간선 $e = (u, v)$를 추가하기 위한 필요조건은 이 간선의 추가가 주기를 만들지 않아야 함
  - u와 v가 같은 분할에 소속되어 있을 때 e를 신장 트리에 추가하면 주기가 만들어짐
  - u와 v의 리더가 동일하면 같은 소속: $\text{Find}(u) = \text{Find}(v)$
    - Find의 비용: 루트 노드까지 가는 비용 (트리 높이에 의해 결정)
- 그러면 union 연산은 어떻게? (결합 비용, 검색 비용을 함께 고려해야 함)



Union-Find을 이용한 Kruskal (3/3)

- Union 연산?
  - 한 쪽은 비용 변화가 없도록 결합할 수 있음
  - 루트 결합이 가장 효과적임
    - 한 쪽 집합의 모든 노드의 포인터를 수정하는 것과 비교
  - 작은 것을 큰 것에 결합하는 것이 효과적임: 크기가 큰 그룹의 리더를 유지
    - 비용(높이) 변화가 없는 노드의 수가 많을수록 이득임
  - 이 경우 각 노드의 갱신은 최대 $O(\log n)$번 발생할 수 있음. Why?
    - 특정 노드 중심(작은 쪽)으로 생각하면 분할의 크기는 계속 2배로 증가함
    - 높이의 증가는 크기가 작은 쪽에만 발생함 $\boxed{2^{\log n}}$
      - 반대 쪽은 비용의 변화가 없음
    - 한 노드의 높이가 $\log n$이 되면 이 분할의 크기는 n이 됨
  - 이 때문에 find 연산도 최대 $O(\log n)$임

Union-Find 자료구조



- 2개의 배열을 이용하여 나타냄
  - 각 노드의 부모 노드를 유지하는 배열
    - 초기에는 자신을 가리키도록 함
  - 분할의 크기를 유지하는 배열 (union 연산에서 필요)

부모링크	1	2	3	4	5
분할크기	1	1	1	1	1

- 예)
  - 간선 (2, 5)가 MST에 추가
  - 간선 (1, 3)가 MST에 추가
  - 간선 (2, 3)가 MST에 추가
  - 간선 (2, 4)이 MST에 추가

1	2	3	4	2
1	2	1	1	0

1	2	1	4	2
2	2	0	1	0

1	1	1	4	2
4	0	0	1	0

1	1	1	1	2
5	0	0	0	0

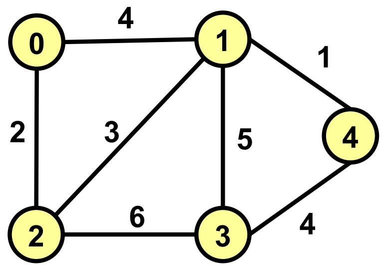
분할의 크기가 같으면
노드 번호가 작은 것을 우선

5의 부모노드; 2
5의 루트노드: 1
(리더노드)

결합은 부모노드가
아니라 두 노드의
루트노드를 결합

Union-Find 자료구조

1	2	3	4	5
1	1	1	1	1



- union할 때 작은 분할에 있는 요소의 갱신
- 다음 find를 이용하여 매번 모든 요소를 갱신하지 않고 진행하는 것이 효과적임

```
find(node)
if node = parent(node) then return node
parent(node) := find(parent(node))
return parent(node)
```

- union은 리더만 갱신

find(1)?=find(4)

0	1	2	3	1
1	2	1	1	0

find(2)?=find(1)

0	0	0	3	1
4	0	0	1	0

find(0)?=find(2)

0	1	0	3	1
2	2	0	1	0

find(3)?=find(4)

0	0	0	0	0
5	0	0	0	0

알고리즘에 따른 그래프 표현 방법

- Prim
  - 힙에서 pop한 노드에 대해 그것의 진출 간선을 검토해야 함
  - 인접행렬 vs. 인접리스트
    - 희소 그래프이면 인접리스트가 더 효과적임
- Kruskal
  - 정렬된 간선 리스트가 필요함
  - 인접 행렬: $O(n^2)$ 데이터를 읽어 리스트를 만든 후 정렬함
  - 인접 리스트: $O(m)$ 데이터를 읽어 리스트를 만든 후 정렬함
    - 무방향이면 중복 제거도 필요함
- 알고리즘의 특성도 일부 영향을 주지만 그래프의 특성이 더 큰 영향을 줌
  - 희소 무방향이면 인접리스트, 밀집 방향이면 인접 행렬

State-of-the-art MST algorithm

- Can we do better?
  - $O(m)$ randomized algorithm [Karger-Klein-Tarjan JACM 1995]
  - $O(m\alpha(n))$ deterministic [Chazelle JACM 2000]
  - [Pettie/Ramachandran JACM 2002]: deterministic algorithm $O(m) \sim O(m\alpha(n))$
- Open Question
  - Is there $O(m)$ deterministic MST algorithm?