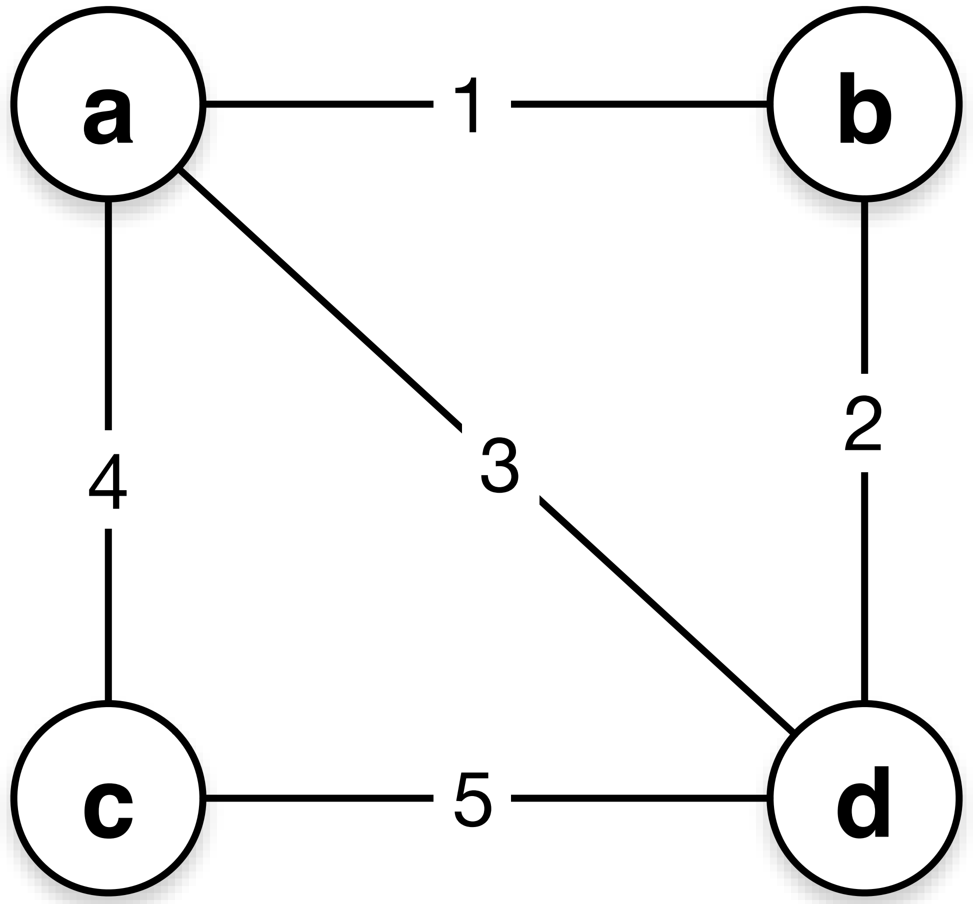
Graph Algorithms (Greedy)

Lecture 6

Timothy Kim
GWU CSCI 6212

Minimum Spanning Tree

- **Problem**: Connect all vertices together deeply as possible
- **Input**: undirected graph G=(V,E) and c_e for $e \in E$
- **Output**:
  - Minimum Cost Tree $T \subseteq E$ that spans V
  - No cycle
  - Connected
- **Assumptions**:
  - G is connected to begin with
  - C is unique



Prim's Algorithm

```
Prim(G):
  X = {s}  // s is chosen arbitrarily
  T = {}
  while X ≠ V:
    let e = (u,v)
      where e is the cheapest crossing edge of cut (X, V-X)
    T = T + e
    X = X + v
  return T
```

Proof of Correctness

- **Claim**: Prim's algorithm correctly computes an MST
- **Part I**: Prim's algorithm produces a spanning tree T*
  - Spanning = all vertices are included
  - Tree = no cycles
- **Part II**: T* is a MST
  - Minimal cost

Definitions to recall

- **Connected Graph**: a graph there is a path between every pair of vertices

Part I

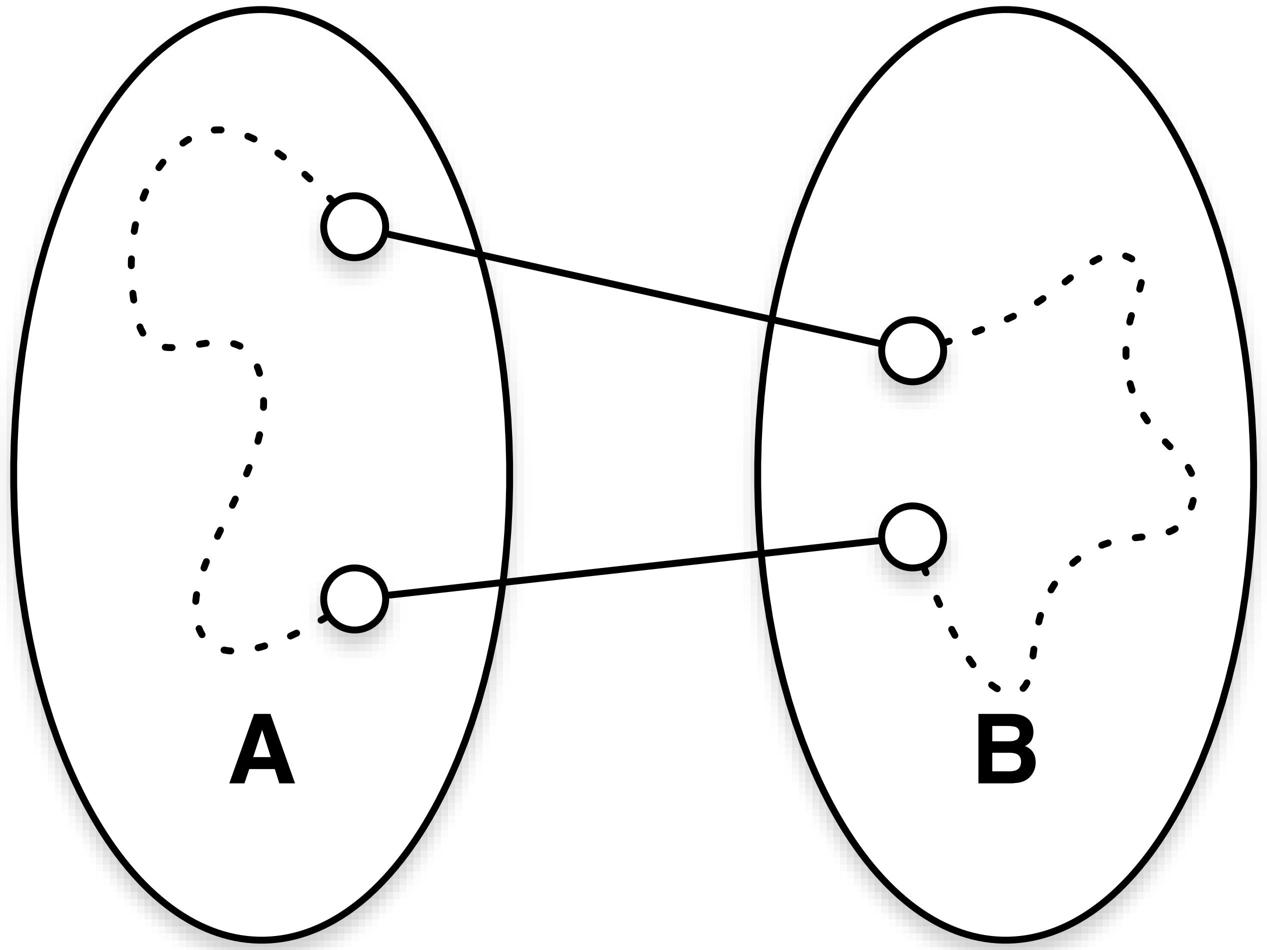
- **Empty cut lemma**:
  - Graph, G, is not connected $\Leftrightarrow \exists$ cut (A,B) of G with no crossing edges
- **Proof of** $\Leftarrow$:
  - Assume $\exists$ cut (A,B) with no crossing edges
  - Pick any $u \in$ A and $v \in$ B
  - Since there are no crossing edges given cut (A,B), there is no edge (u,v)
  - $\therefore$ by the definition of connected graph, G is not connected

Part I cont.

- **Empty cut lemma**:
  - Graph, G, is not connected $\Leftrightarrow \exists$ cut (A,B) of G with no crossing edges
- **Proof of $\Longrightarrow$**:
  - Assume Graph, G, is not connected
  - Pick any $u \in$ G
  - Create a cut of (A, B) such that
    - A = { all vertices reachable from u }
    - B = { all other vertices }
  - Then cut (A, B) has no crossing edges

Part 1 cont.

- **Double Crossing Lemma**:
  - Suppose cycle, $C \subseteq E$, has an edge crossing a cut (A,B) then there must exist another edge that crosses that cut.
  - Proof by contradiction
- **Lemma 3**:
  - If e is the only edge crossing a cut (A,B) then e is not part of any cycle.
  - Proof using Double Crossing Lemma



Part 1 cont.

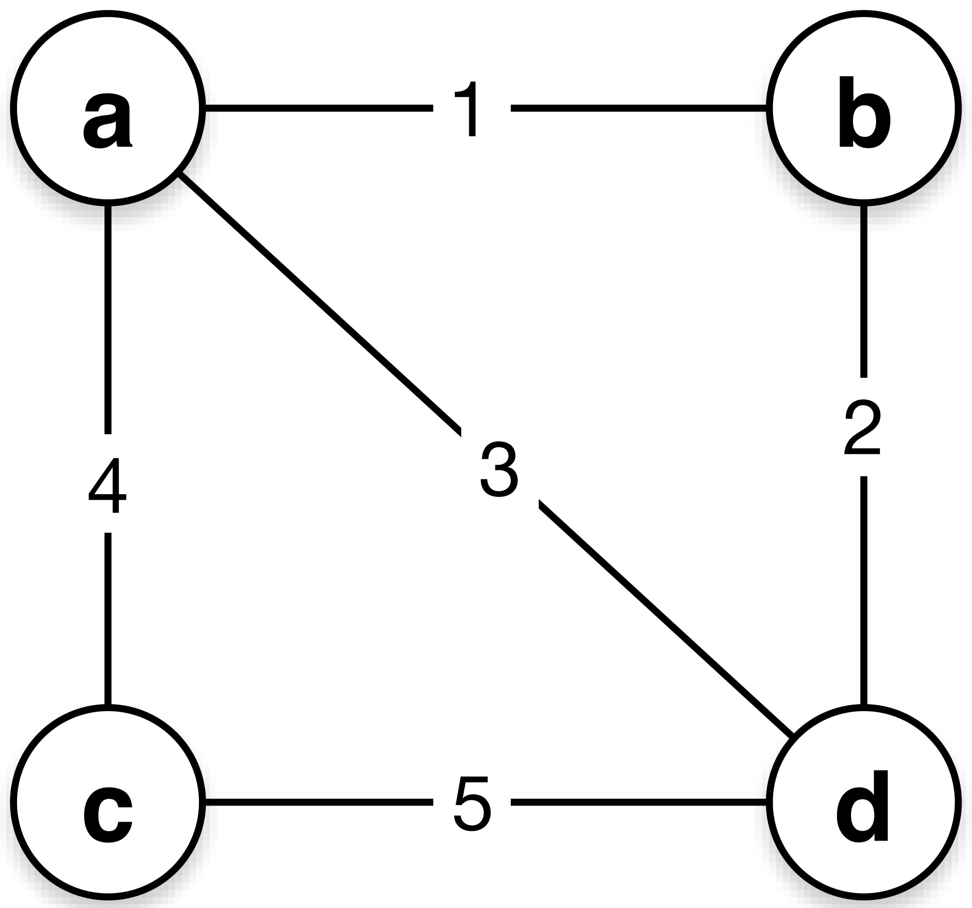
- **Part I**: Prim's algorithm produces a **spanning** tree
  - The algorithm chooses only edges stemming from X. Therefore the algorithm maintains the invariant of T spans X (meaning T includes all the vertices in X).
  - Algorithm must halt (X eventually = V),
    otherwise the cut of (X, V − X) will have no crossing edges. If the cut has no crossing edges, by Empty Cut Lemma, G must be disconnected. This is a contradiction, thus the algorithm halts with X = V.
  - $\therefore$ Prim's algorithm produces a T that spans V

Part I cont.

- **Part I**: Prim's algorithm produces a spanning **tree** (no cycle)
  - Whenever an edge, e, gets added to T, e is the first edge to cross the cut (X, V − X). By Lemma 3, e does not create a cycle
  - $\therefore$ Prim's algorithm produces a tree

Part II

- **Part II**: T* is a MST
  - Minimal cost
- **The Cut Property**
  - Given $e \in$ G, suppose $\exists$ cut (A,B) | e is the cheapest crossing edge, then $e \in$ MST(G)



Part II

- **Claim**: Cut Property $\Longrightarrow$ Prim's Algorithm produces MST(G)
- Every edge $e \in T^*$ is chosen as the cheapest crossing edge of cut $(X, V - X)$.
- By the cut property, $T^* \subseteq \mathrm{MST}(G)$.
- From Part I, since T^* is a spanning tree of G, $T^* = \mathrm{MST}(G)$. QED

Proof of cut property

- **The Cut Property**
  - Given $e \in$ G, suppose $\exists$ cut (A,B) | e is the cheapest crossing edge, then $e \in$ MST(G), Y
- By contradiction.
  - Suppose e is the cheapest crossing edge of a cut (A,B) of G, yet $e \notin$ MST(G), Y

Proof of cut property

- Suppose e is the cheapest crossing edge of a cut (A,B) of G, yet $e \notin$ MST(G), Y
- Since Y doesn't include e, then it must include another edge, $f \mid c_f > c_e$, $f \in$ Y and cuts (A,B) and is part of Y. (Otherwise, Y is not connected.)
- (We want to use a swap method here, but if e or f is part of a cycle, we can't just swap.)
- Since $f \in$ Y, and Y is a spanning tree, Y $+$ e could create a cycle.

Proof of cut property

- By Double Cut Lemma, there must exist another edge, $e' \mid c_{e'} > c_e$ and creates the cycle with e
- (Now we can swap e with e')
- Note, $\mathrm{Y}' = \mathrm{Y} + \mathrm{e} - \mathrm{e}'$ is a spanning tree.
- But cost of $\mathrm{Y}' < \mathrm{Y}$ which is a contradiction. QED.

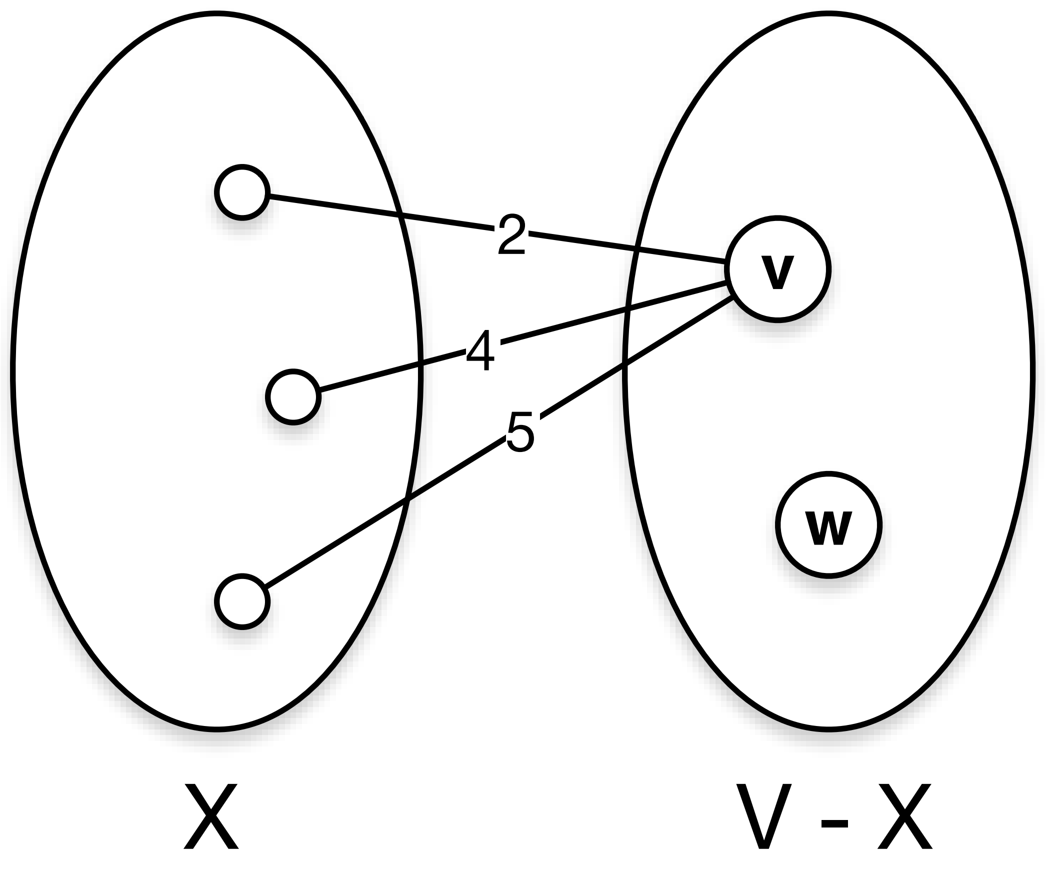
Prim's Algorithm Running Time

```
Prim(G):
  X = {s}  // s is chosen arbitrarily
  T = {}
  while X ≠ V:
    let e = (u,v)
      where e is the cheapest crossing edge of cut (X, V-X)
    T = T + e
    X = X + v
  return T
```

Running time of Prim's

- O(n $\times$ m) - Literal implementation
- We can use MinHeap where most of its operations are in $O(\log\, n)$
- Use Heap to store edges $\Longrightarrow O(m \log\, n)$
- But faster to store vertices in Heap with following invariant
    - Invariant #1: Elements in heap $= v \in$ V $-$ X
    - Invariant #2: for $v \in$ V - X:

$$\text{key}_v = \text{cheapest edge } (u, v)$$

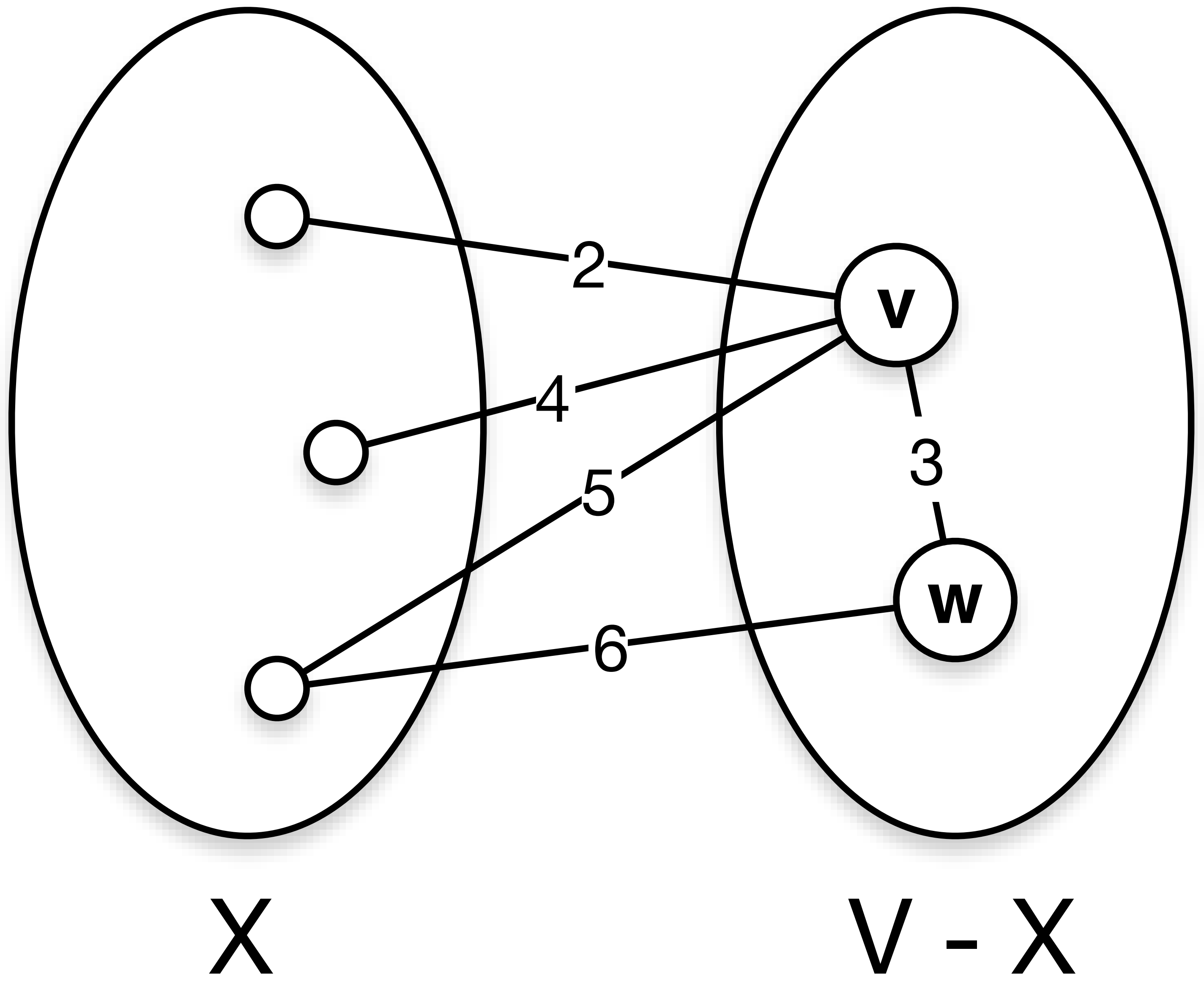


Prim's with MinHeap of Vertices

- **Preprocessing** (Initialization) of heap:
  - Initial cut = $(\{\mathrm{s}\}, \mathrm{V} - \{\mathrm{s}\})$
  - Find all the edges that cross that cut and create heap:
    - $\mathrm{O}(m + n \log n)$ or $\mathrm{O}(m + n)$ if you use heapify
    - Since $\mathrm{m} \geq \mathrm{n} - 1$, $\mathrm{O}(m + n) = \mathrm{O}(\mathrm{n})$

Prim's with MinHeap of Vertices

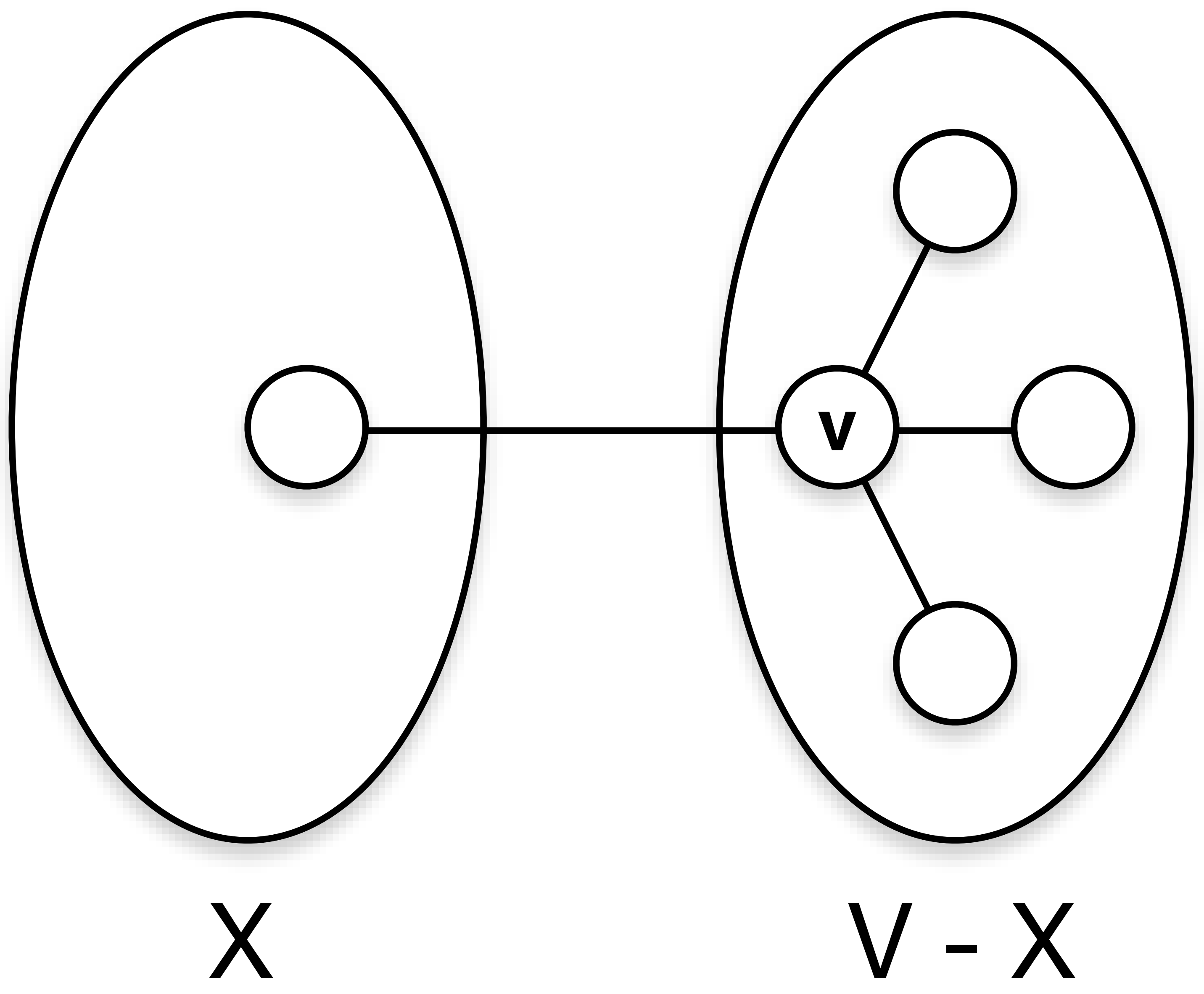
- **During execution**:
  - To pick the cheapest edge, single ExtractMin call to heap will give you the right edge.
  - $O(\log\ n)$
- **Keeping the invariant**
  after ExtractMin has been called



Prim's with MinHeap of Vertices

- **Keeping the invariant** after ExtractMin has been called
- Use extra metadata to speed up delete

```
When v is added to X:
  for each edge (v,w) ∈ E:
    if w ∈ V - X:
      delete w from heap
      key[w] = min{key[w], c_{v,w}}
      insert w into heap
```



Final Running Time of Prim

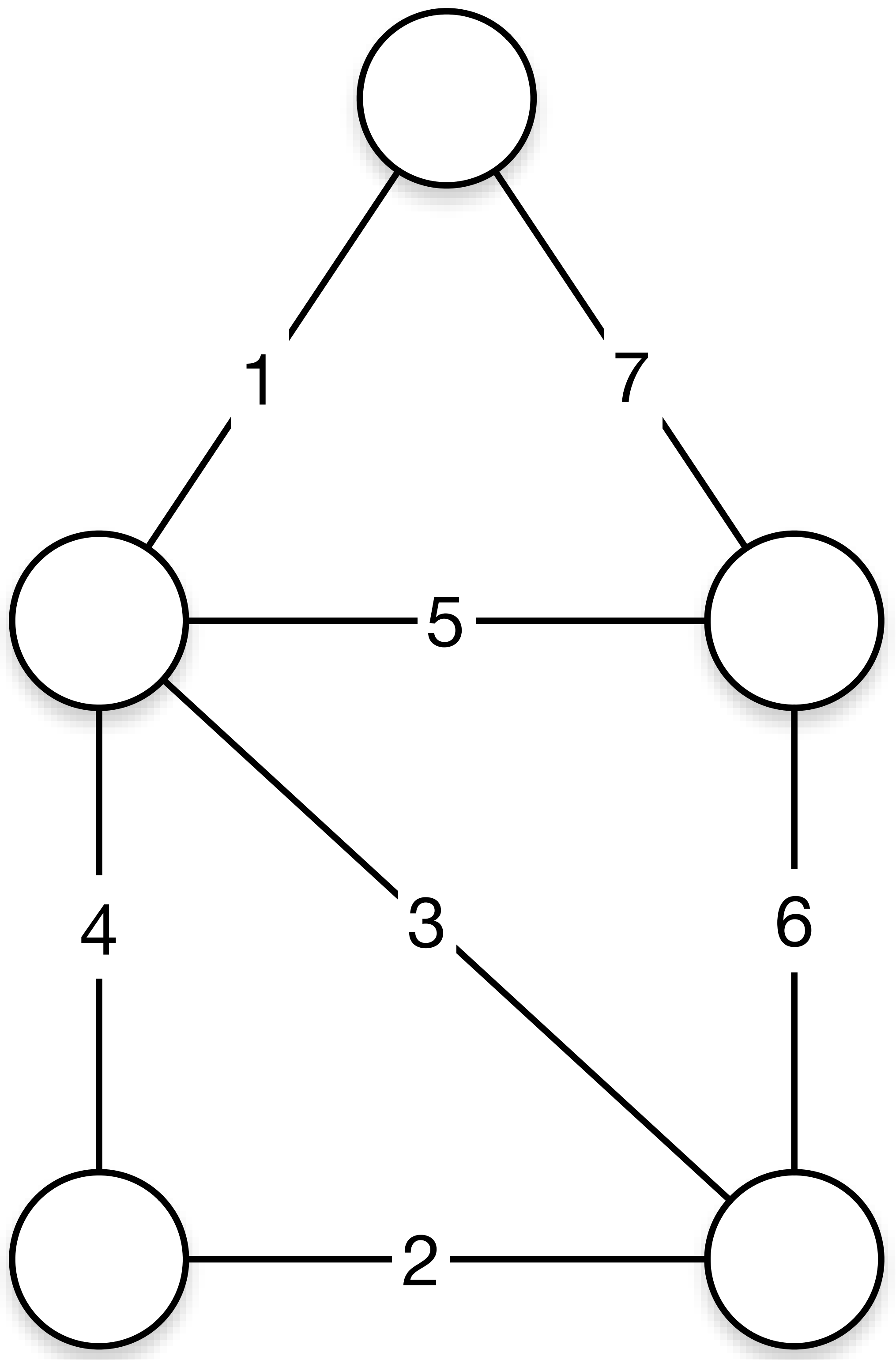
- Preprocessing: O(n)
- One ExtractMin called per each vertex: O(n log n)
- Each edge triggers at most one delete/insert: $O(m \log n)$
- Entire running time $= O(m \log n) = O(n \log n)$
    - At the cost of factor of $\log n$, we can calculate MST as fast as reading in the graph.

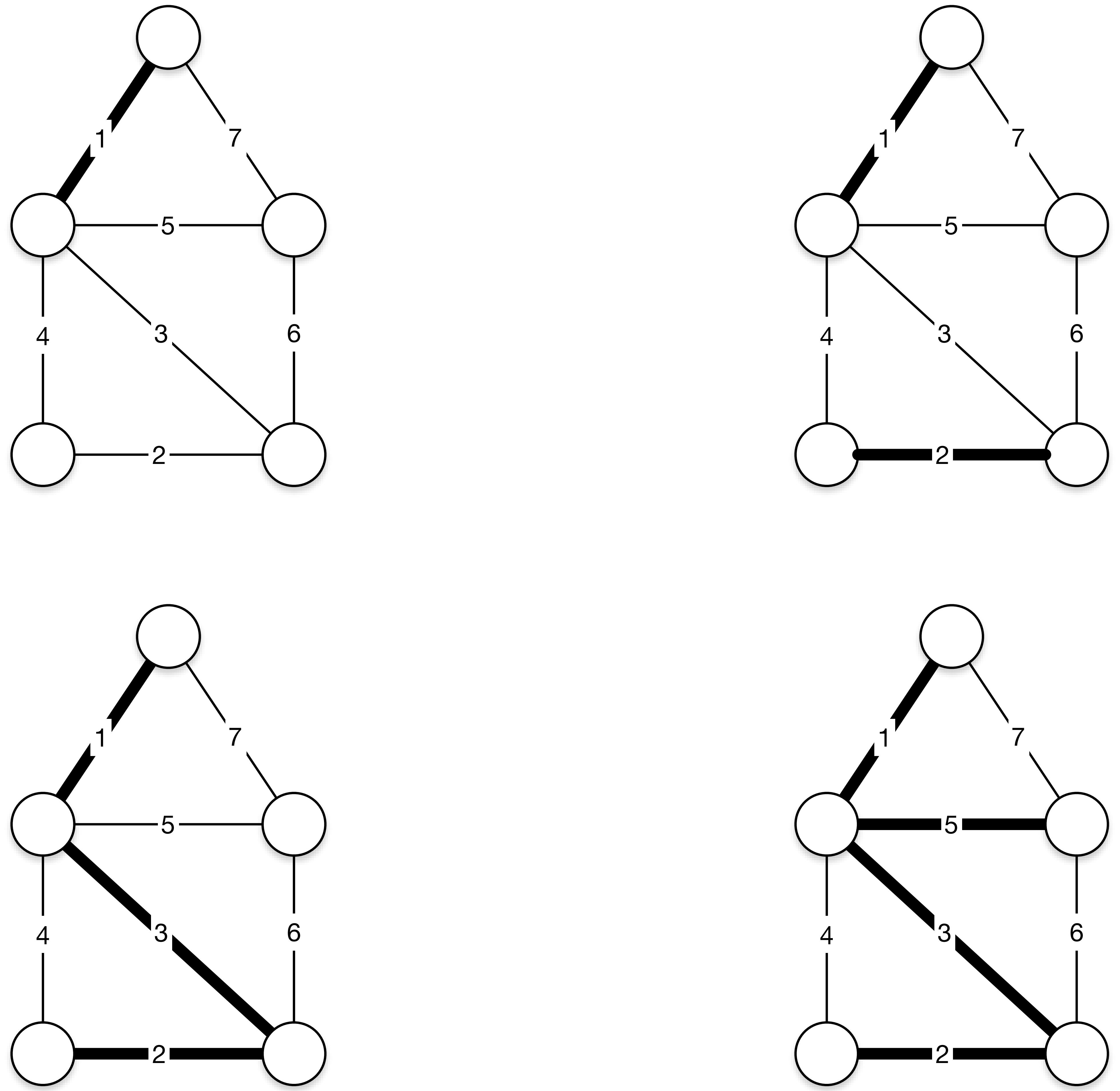
Kruskal's MST

```
Kruskal(E, V, C):
  sort E by C
  T = {}
  for i = 1 to m:
    if T ∪ {eᵢ} has no cycle:
      T = T ∪ {eᵢ}
```

- **Assumption**:
  - G is connected
  - Distinct c_e values

Example





Proof of Correctness

- **Claim**: Kruskal's algorithm correctly computes an MST
- **Part I**: Kruskal's algorithm produces a spanning tree T*
  - Kruskal's creates graph that spans G. (for i = 1 to m:)
  - Are they tree? (no cycle)
  - Is it connected?
- **Part III**: T* is a MST
  - Is it minimal?

Kruskal's is a spanning tree

- **Part Ia**: T* has no cycle
  - By the nature of the algorithm, T* does not create a cycle.
- **Part Ib**: T* is connected (single tree)
  - To prove that T* is connected, we need to prove that every cut of T* has at least one edge that crosses it. (By Empty Cut Lemma)
  - Consider a cut of T*, since G is connected there must be at least one edge that crosses the cut that's part of G.
  - Of those edge, the cheapest edge would be part of T* because that's the first edge of that cut.

Kruskal's is Minimal

- Every edge of T* is justified by the Cut Property
- **The Cut Property**
    - Given $e \in$ G, suppose $\exists$ cut (A,B) $|$ e is the cheapest crossing edge, then $e \in$ MST(G)

Proof of Part II

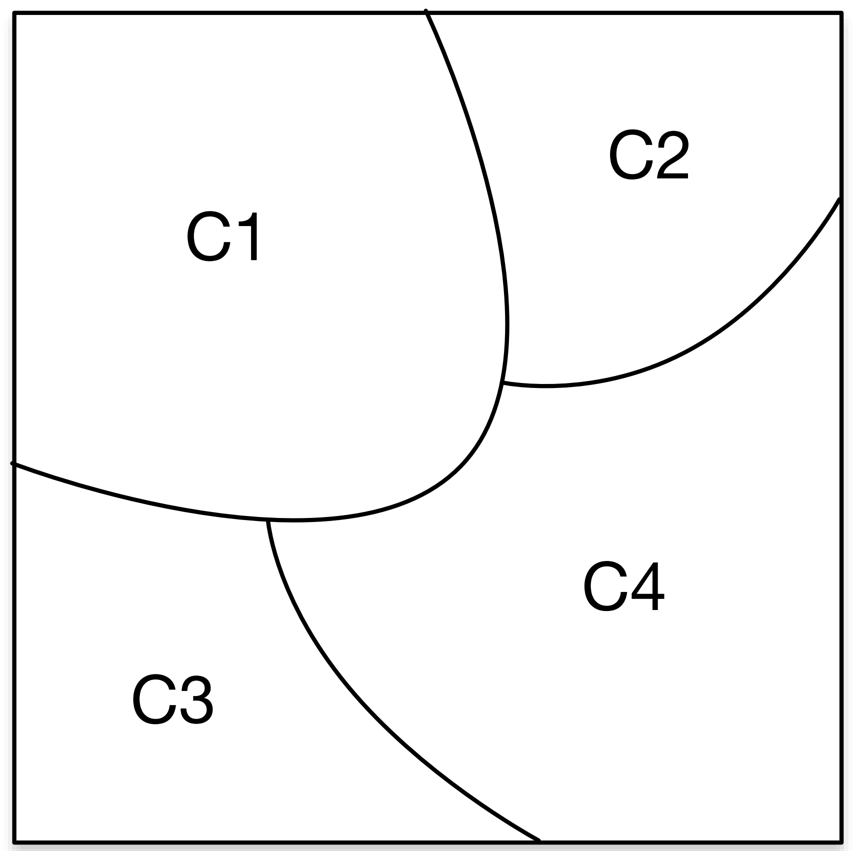
- At each step of the algorithm, we have a forest.
- Let *(u,v)* be the edge that the algorithm picks.
- Since $T \cup \{\ (u,v)\ \}$ has no cycle $\Longrightarrow \exists$ an empty cut, *(A, B)*, that separates u and v.
- (u,v) must be the cheapest edge that crosses (A,B) because all other edge that was not part of the forest cost more than (u,v).

Running time analysis

- Naive implementation
  - Checking for cycle = $O(n)$ using DFS or BFS.
  - $\underbrace{O(m \log m)}_{\text{pre-processing}} + \underbrace{O(mn)}_{\text{cycle checks}} = \underbrace{O(mn)}_{\text{total running time}}$
- Can we do better?
  - Speeding cycle checking by using a Union-Find data structure.

Union-Find

- **Purpose**: maintain partition of a set of objects
- **Operations**:
  - Find(x) = returns the group that x belongs to
  - Union(C_i, C_j) = combine C_i and C_j into single group
  - Name of a group can be represented by its representative.
- Kruskal's Application
  - Objects = vertices
  - Groups = connected components (Forests)



Kruskal with Union-Find

```
Kruskal(V, E, C):
  Sort E by C
  ∀ v ∈ V is in its own group
  T = {}
  for e_i = e_1 to e_m:
    (u, v) = e_i
    if FIND(u) != FIND(v):
      T = T ∪ {e_i}
      UNION(u, v)
  return T
```

- Running Time
  - O(m log m) + O(m (FIND(n) + UNION(n)))

Naive Union-Find implementation

```
UNION(a, b):
  FIND(b).parent = a

FIND(b):
  if b.parent == b:
    b
  else
    FIND(b.parent)
```

- Running time?
  - UNION: $O(n)$
  - FIND: $O(n)$
- Kruskal's Running Time
  - $\mathrm{O}(m \log m) + \mathrm{O}(m\ (\mathrm{O}(1) + \mathrm{O}(n)))$
  - $= \mathrm{O}(mn)$

Eager Union

```
UNION(a,b):
  if SIZE(a) > SIZE(b):
    UNION(b,a)
  for i in SET(a):
    i.parent = b

FIND(x):
  x.parent
```

- UNION(a,b)
  - Take the smaller set and make all its element point to the root of the other set
  - $O(n)$??
- FIND(x)
  - $O(1)$

Eager Union Analysis

- Single UNION call is bounded by O(n). However, in our usage in Kruskal's algorithm, we end up calling UNION until the entire vertices are merged into one set.
- What is the amortized (entire execution) running time of UNION then?
- Fix an element, x. We want to know how many times we have to update its parents during the course of all UNIONs. Whenever x has to update its parent, the target set is always at least double the size.
- So finally, the question is "how many times can a number double before it is the size of n?"
- $\text{UNION}(a,b) = \text{O}(\log n)$

Lazy Union

```
UNION(a,b):
  x = FIND(a)
  y = FIND(b)
  if RANK(x) > RANK(y):
    y.parent = x
  else:
    x.parent = y
  update RANK

FIND(x):
  if (x.parent == x):
    return x
  else:
    return FIND(x.parent)
```

- Take the root of the smaller tree and make it the child of the other
- Let RANK[x] $=$ maximum $\#$ of hops from leaf to x
- Initially RANK[x] $= 0$ for all possible x
- Analysis
  - UNION $=$ O(1)
  - FIND $=$ O(n)??
  - Note, max RANK $=$ worst case running time for FIND

Lazy Union Analysis

- Note:
  - For all object x, RANK[x] only goes up over time
  - Only ranks of the root can go up
  - RANK[x] strictly increases along the path to the root

Rank Lemma

- Consider an arbitrary sequence of UNIONS
- ∀ $r \in \{0, 1, 2, \ldots\}$ there are at most $n/2^r$ object with rank r
- Corollary:
  - if $r = \log_2 n \rightarrow n/2\hat{}(\log_2 n) = 1$
  - max rank ≤ log2n
  - FIND, UNION $= O(\log\ n)$
  - Kruska's Running Time: $\mathrm{O}(m \log\ m) + \mathrm{O}(m\ (\log\ n + \log\ n))$
    $= \mathrm{O}(m \log\ n)$

Proof of Rank Lemma

- Claim 1
  - If x,y have same rank, then their subtree are disjoint
- Claim 2
  - The subtree of a rank r has object size ≥ 2^r
- Fix an r. Each of the object with rank r has 2^r objects that can reach them. Since they are all disjoint and there are at most n objects number of objects with the r must be bounded by $\mathrm{n}/2^r$

Proof of Rank Lemma

- Claim 1
  - If x, y have same rank, then their subtree are disjoint
- Proof by Contradiction
  - If x, y are not disjoint, $\exists\ z \mid \exists$ paths $z \rightsquigarrow x$, $z \rightsquigarrow y$
  - Because of how trees are structured, the path must be either

    $z \rightsquigarrow x \rightsquigarrow y$ or $z \rightsquigarrow y \rightsquigarrow x$
  - Since rank increases strictly, this is a contradiction
  - QED

Proof of Rank Lemma

- Claim 2
  - The subtree of a rank r has object size $\geq 2^r$
- Proof by Induction
  - Base Case: rank $= 0$, all subtree size $= 1$
  - Inductive Hypothesis: if SIZE[k] $\geq 2^k$ then SIZE[k+1] $\geq 2^k$
  - Inductive Case: UNION$(a,\ b) \rightarrow x = $ FIND$(a),\ y = $ FIND(b)
    - RANK[a] $=$ RANK[b] $= r$
    - b's new rank $= r + 1$
    - b's new size $=$ SIZE[b] $+$ SIZE[a] $\geq 2^r + 2^r = 2 \cdot 2^r = 2^{r+1}$

Can we do better?

- Path Compression
  - We do awful lot of FIND
  - So whenever we call FIND[x], we update all the object along the path from x to root such that their parents are now pointing to the root.
- Analysis
  - O(log* n) where $\log^* n := \begin{cases} 0 & \text{if } n \leq 1; \\ 1 + \log^*(\log n) & \text{if } n > 1 \end{cases}$
  - log* $(2^{65,536}) = 5$
  - For all n < 65,536, Kruskal = O(m log n) + O(m (5 + 5))

Can we do better?

- Randomized UNION-FIND = O(m)
- Deterministic algorithm that runs in O(m)?
- Best so far = O(m α(n)) where α is a inverse Ackermann function
  - It grows slower than log*
- 2002 Paper
  - There exist an algorithm that has been proven to be the optimal algorithm. But its running time hasn't been proven.
  - All we know is that it runs between Θ(m) and O(m α(n))
- Open question
  - Simple randomized UNION-FIND?
  - Deterministic O(m)?