Greedy algorithms
exchange - MST & union-find

Reading: Kleinberg & Tardos
Ch. 4.2, 4.5
Additional resource: CLRS Ch. 16.4

Can we get there from here?

Suppose you've been hired to help plan a city -- you're in charge of designing the public transportation system! A set of desired stops has already been identified.

How do you connect the stops while optimizing costs? (e.g., construction materials, transit time, distance…)

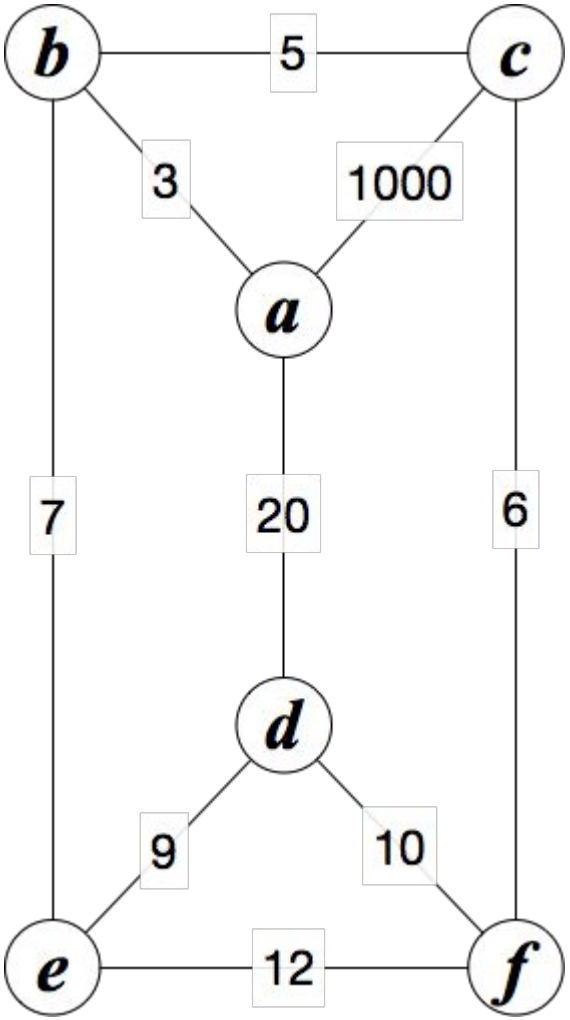
Step 1: formulate the problem

Problem formulation: minimum spanning tree (MST)

Input:

- undirected graph $G = (V, E)$
- edge lengths/costs
  $\ell(e)$ for each $e \in E$

Definitions



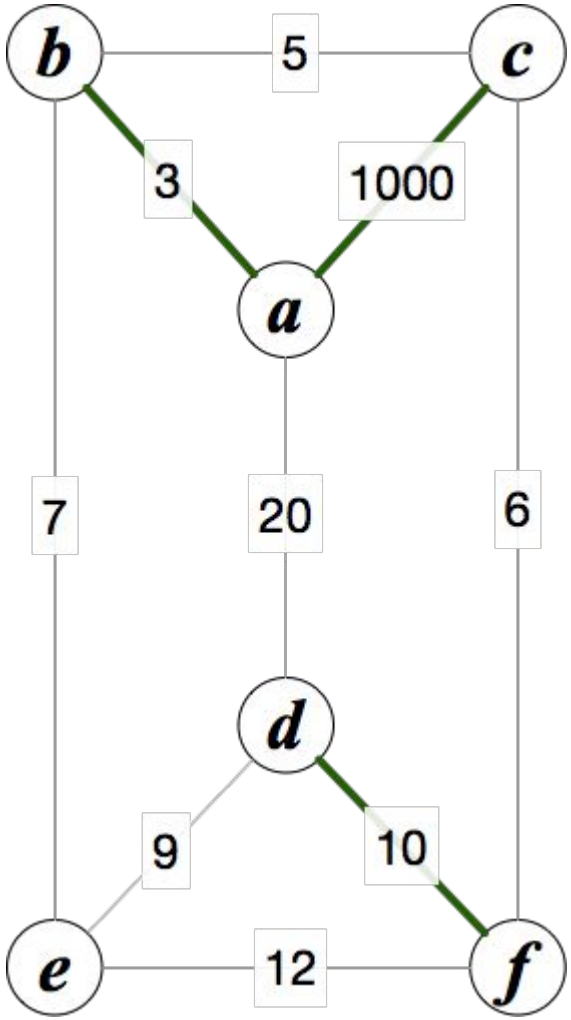
Problem formulation: minimum spanning tree (MST)

Input:

- undirected graph $G = (V, E)$
- edge lengths/costs
  $\ell(e)$ for each $e \in E$

Definitions

- **forest**: subset of edges with no cycles



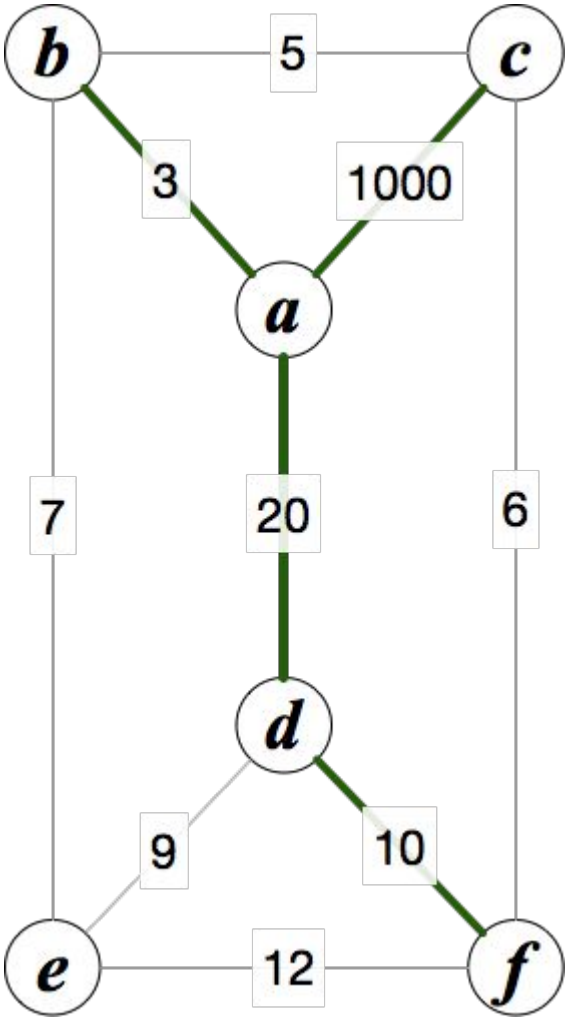
Problem formulation: minimum spanning tree (MST)

Input:

- undirected graph $G = (V, E)$
- edge lengths/costs
  $\ell(e)$ for each $e \in E$

Definitions

- **forest**: subset of edges with no cycles
- **tree**: forest with one connected component



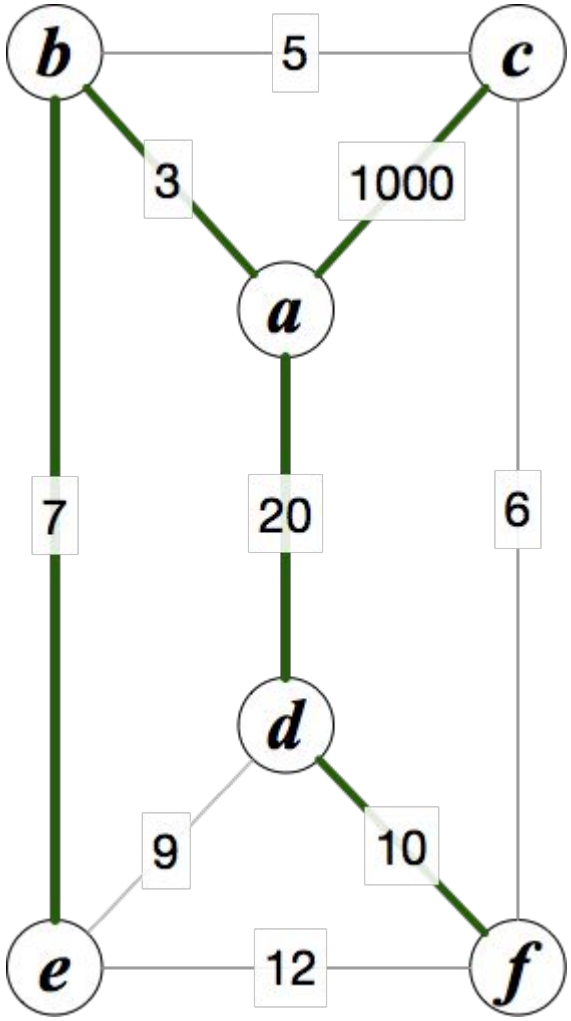
Problem formulation: minimum spanning tree (MST)

Input:

- undirected graph $G = (V, E)$
- edge lengths/costs
  $\ell(e)$ for each $e \in E$

Definitions

- **forest**: subset of edges with no cycles
- **tree**: forest with one connected component
- **spanning tree**: tree that touches every vertex



Problem formulation: minimum spanning tree (MST)

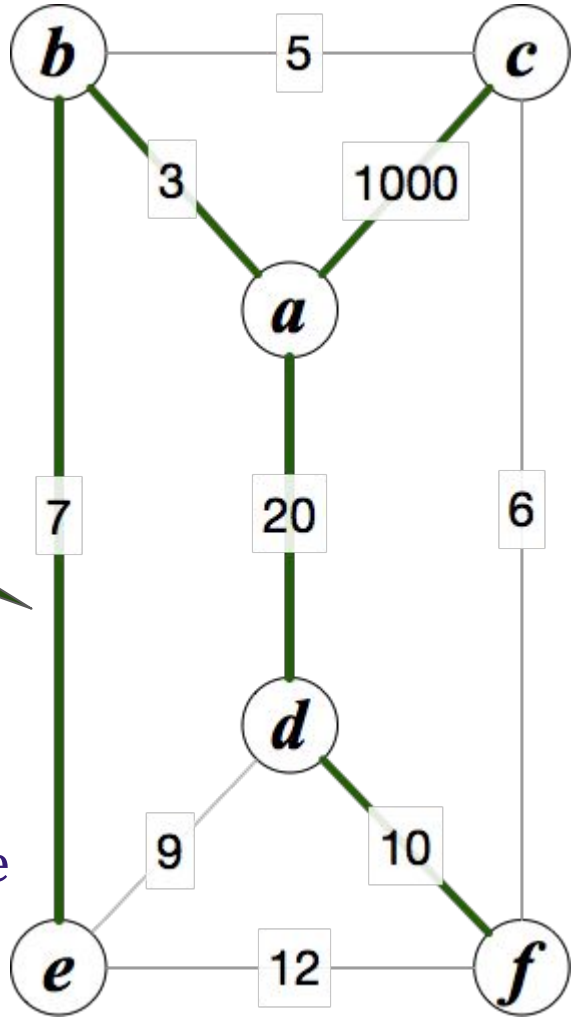
Input:

- undirected graph $G = (V, E)$
- edge lengths/costs $\ell(e)$ for each $e \in E$

cost:
3 + 7 + 20 + 10 + 1000
= **1040**

Definitions

- **forest**: subset of edges with no cycles
- **tree**: forest with one connected component
- **spanning tree**: tree that touches every vertex
- spanning tree **cost**: sum of edge lengths in tree



Problem formulation: minimum spanning tree (MST)

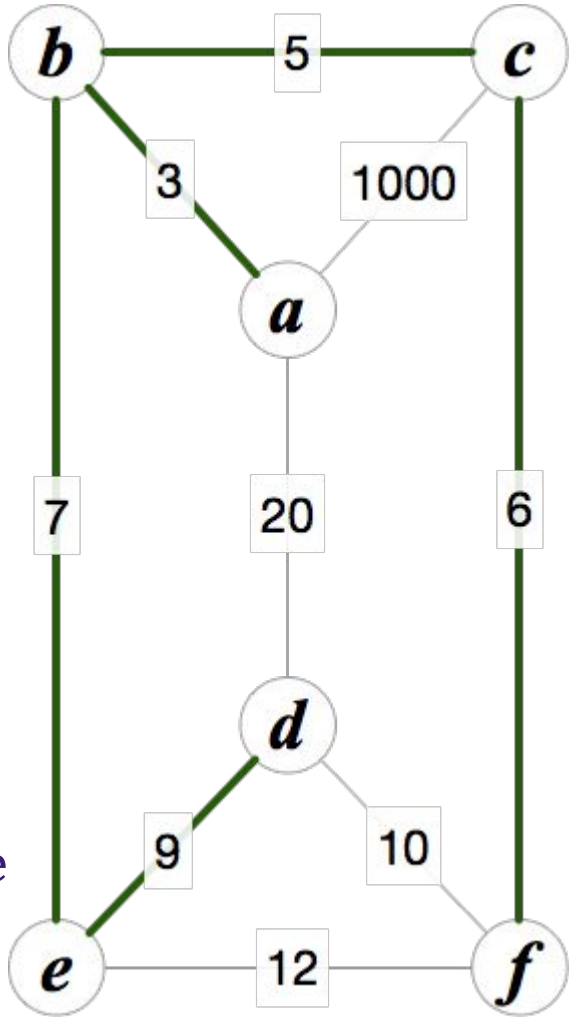
Input:

- undirected graph $G = (V, E)$
- edge lengths/costs
  $\ell(e)$ for each $e \in E$

Definitions

- **forest**: subset of edges with no cycles
- **tree**: forest with one connected component
- **spanning tree**: tree that touches every vertex
- spanning tree **cost**: sum of edge lengths in tree

Problem: ***find MST, minimum-cost spanning tree***



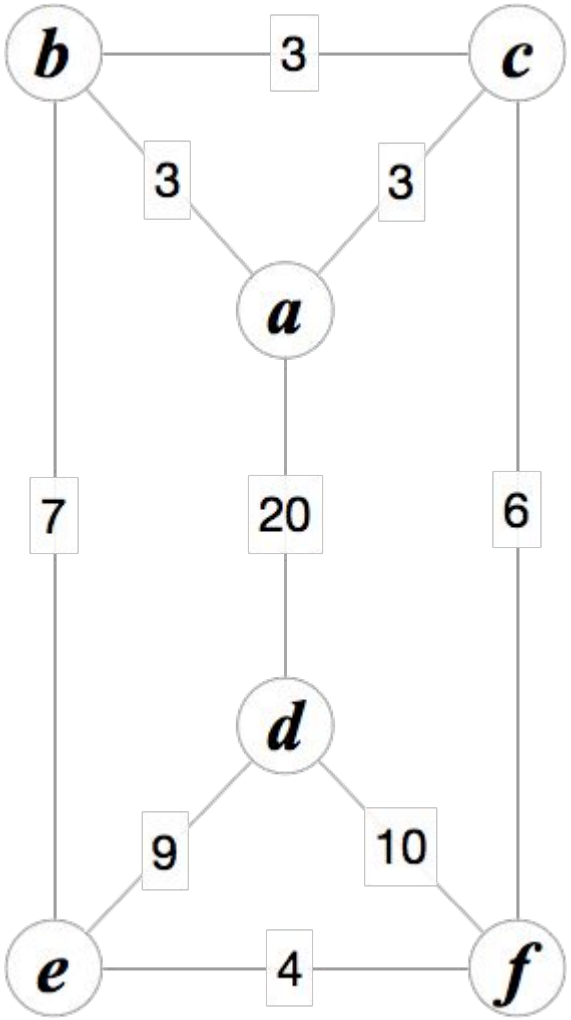
Step 2: propose an approach

Greedy algorithm

- Builds up to solution by locally selecting “best” choice
- 3 logical greedy approaches:
  - Kruskal’s: pick **edges** by weight unless cycle
  - Prim’s: start at root, add **vertex** by edge connection
  - Reverse-delete: remove **edges** by weight unless disconnect

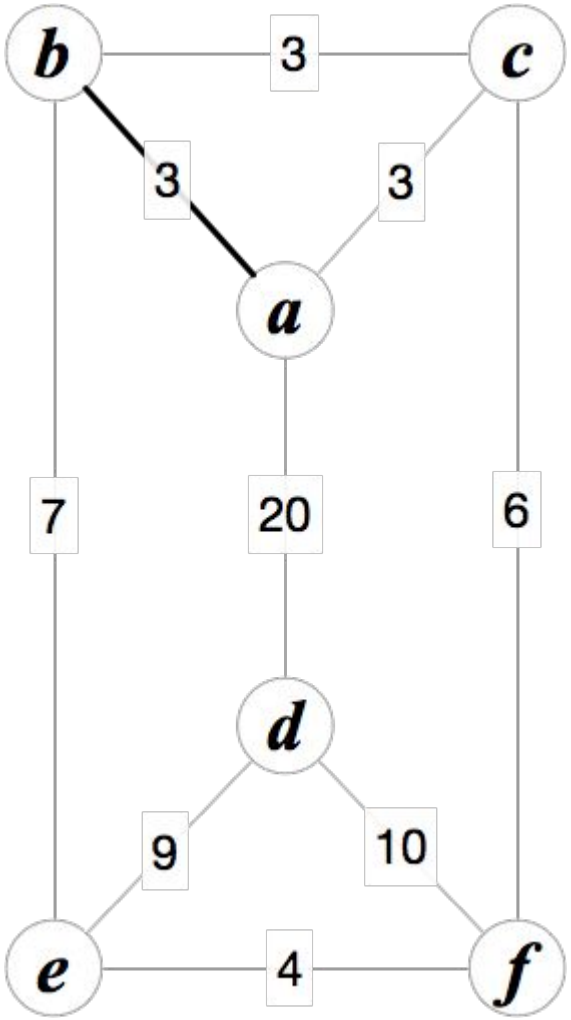
Kruskal's algorithm: greedily add edges

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - Add e to T if it does not create a cycle



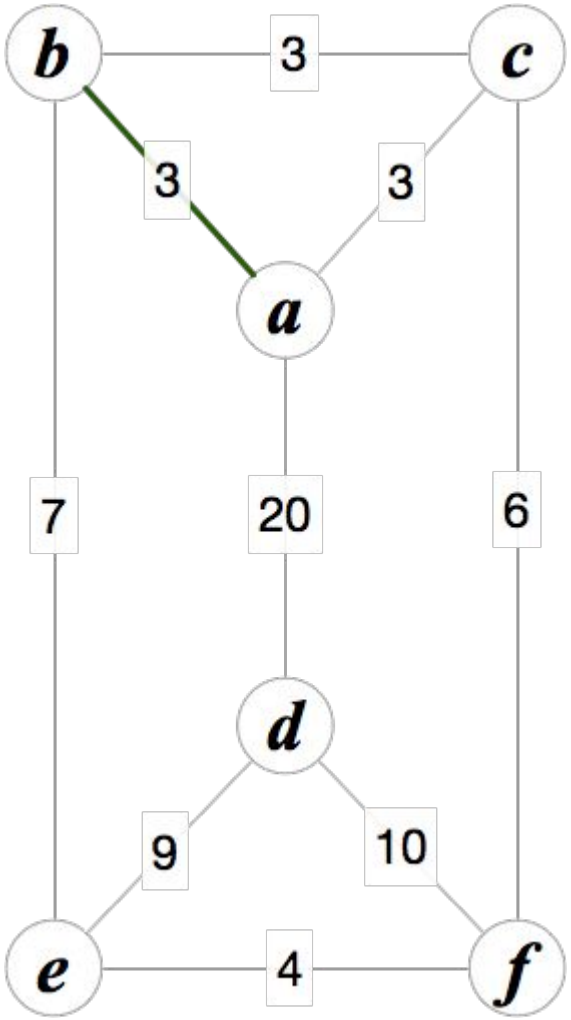
Kruskal's algorithm: greedily add edges

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - Add e to T if it does not create a cycle



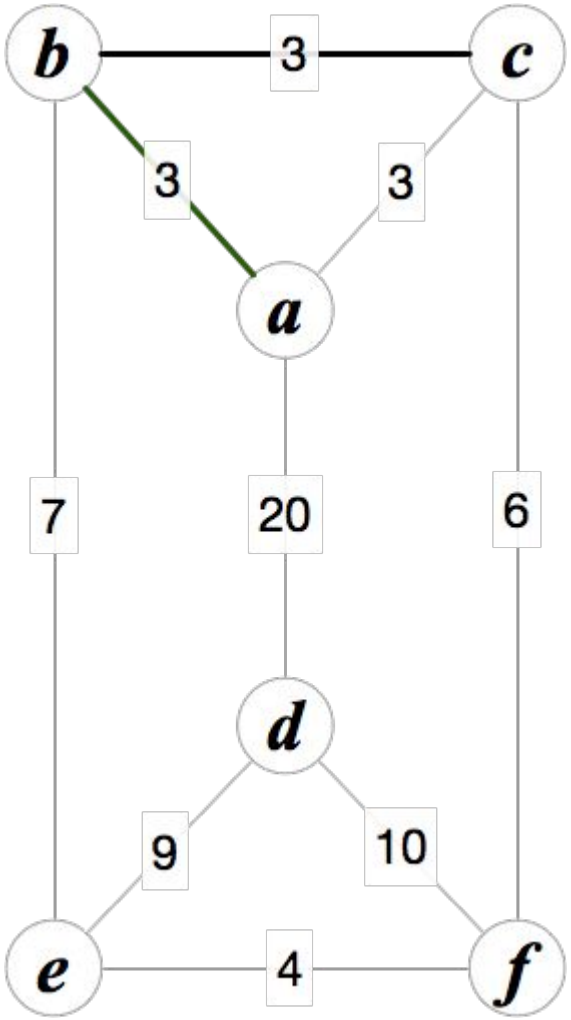
Kruskal's algorithm: greedily add edges

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - Add e to T if it does not create a cycle



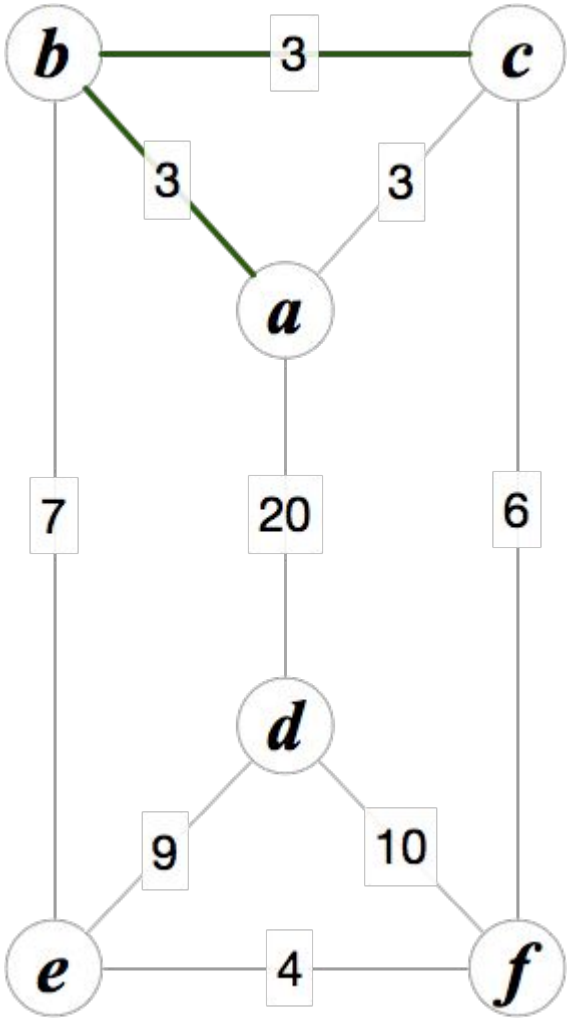
Kruskal's algorithm: greedily add edges

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - Add e to T if it does not create a cycle



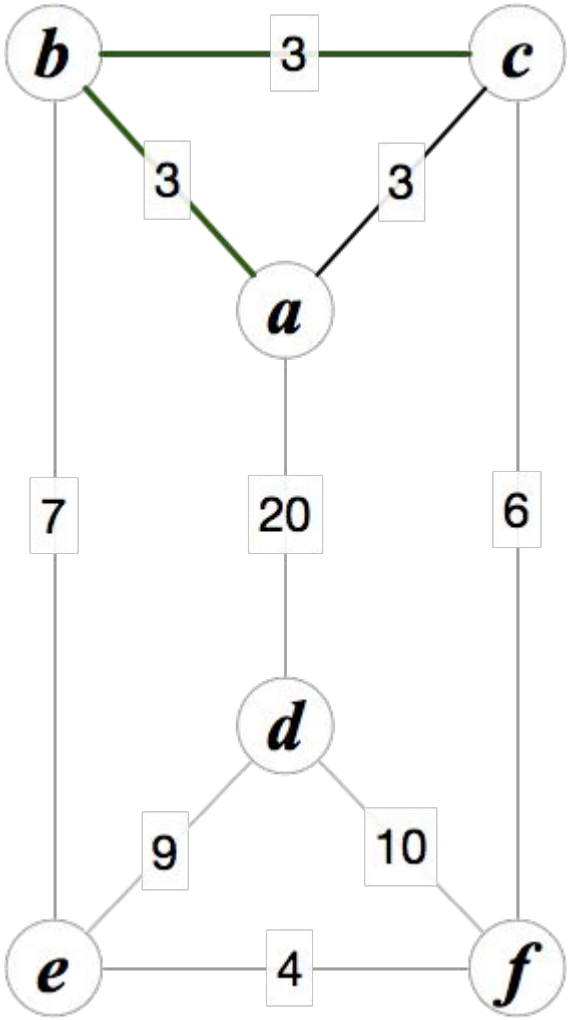
Kruskal's algorithm: greedily add edges

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - Add e to T if it does not create a cycle



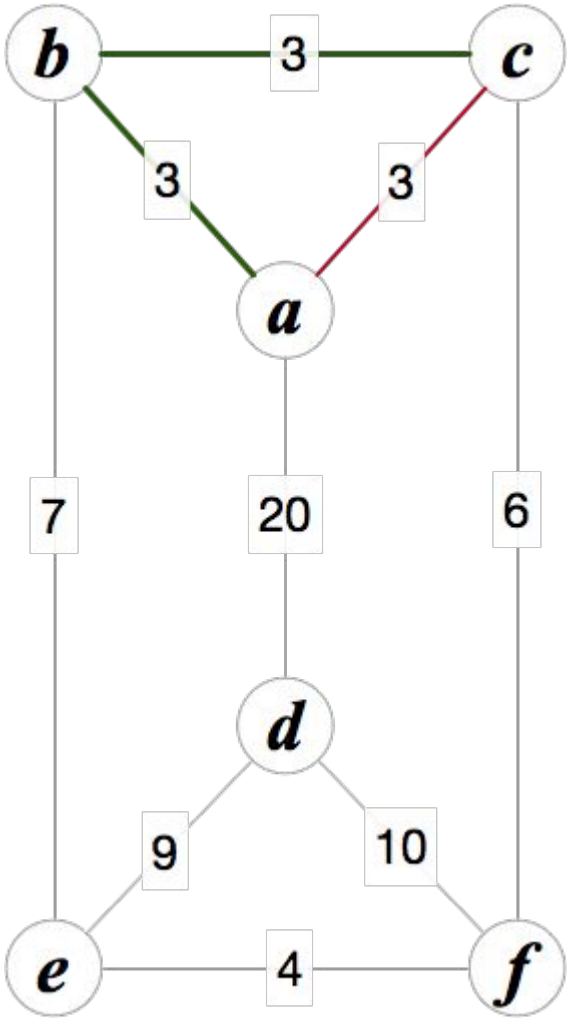
Kruskal's algorithm: greedily add edges

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - Add e to T if it does not create a cycle



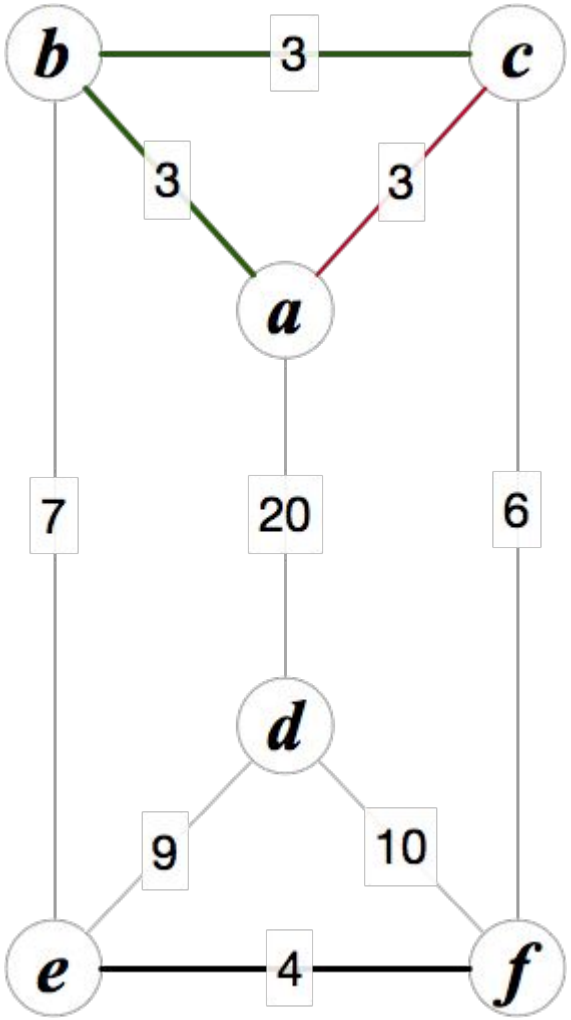
Kruskal's algorithm: greedily add edges

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - Add e to T if it does not create a cycle



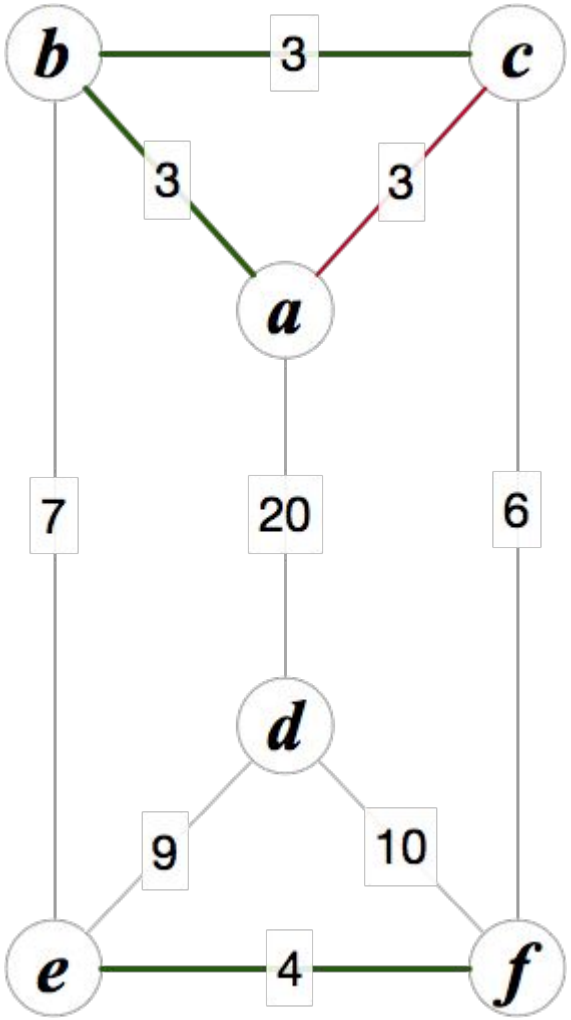
Kruskal's algorithm: greedily add edges

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - Add e to T if it does not create a cycle



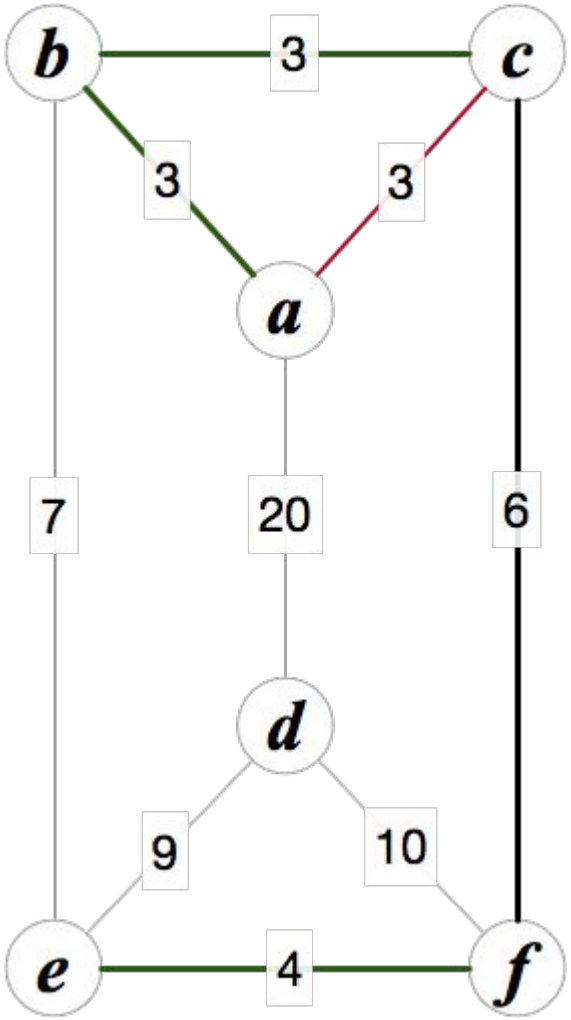
Kruskal's algorithm: greedily add edges

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - Add e to T if it does not create a cycle



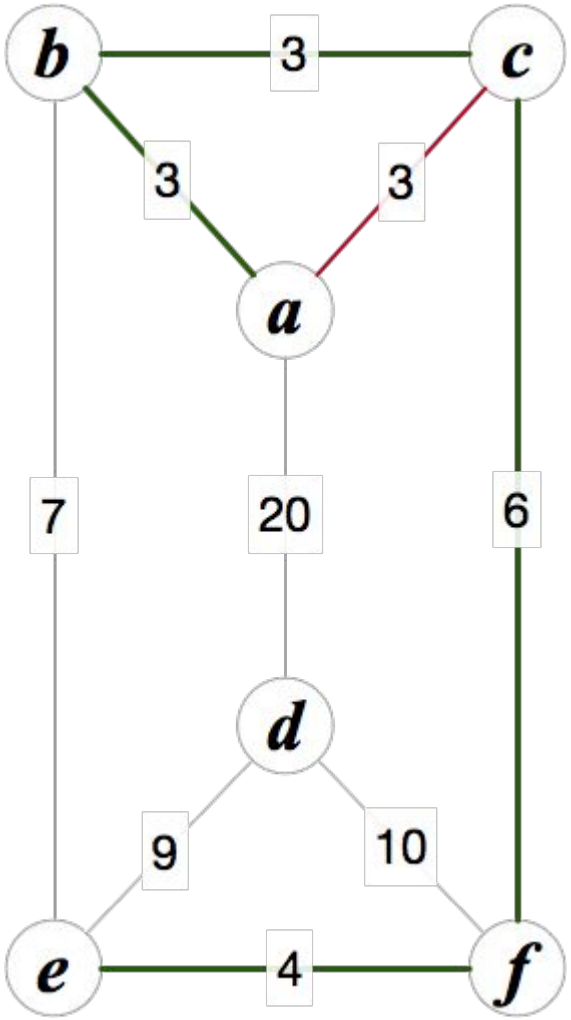
Kruskal's algorithm: greedily add edges

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - Add e to T if it does not create a cycle



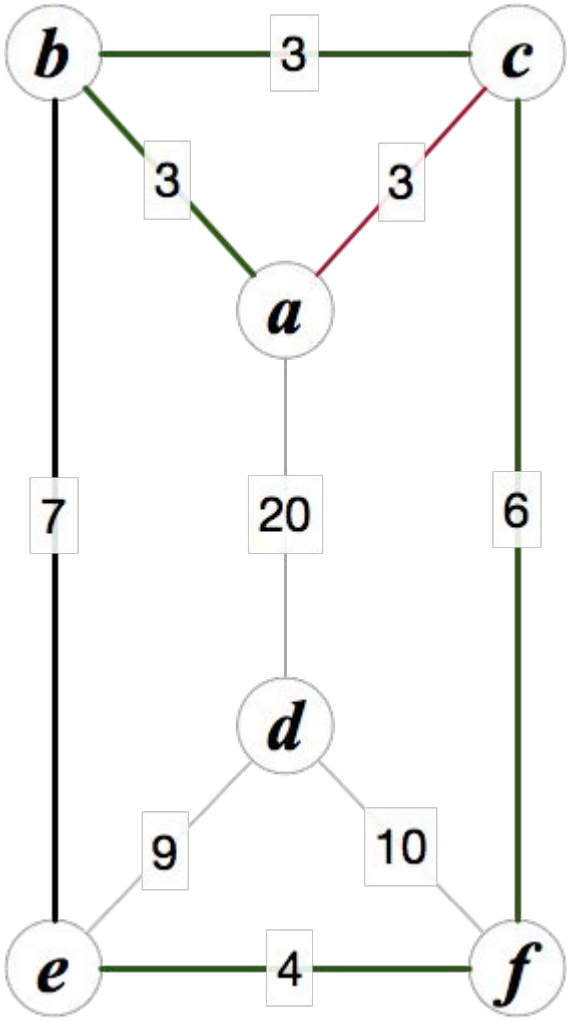
Kruskal's algorithm: greedily add edges

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - Add e to T if it does not create a cycle



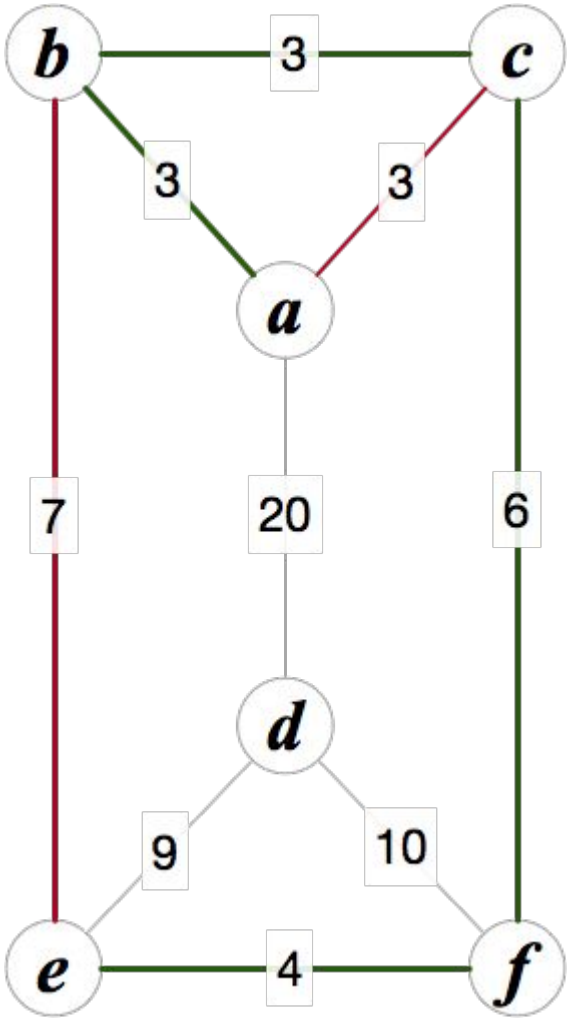
Kruskal's algorithm: greedily add edges

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - Add e to T if it does not create a cycle



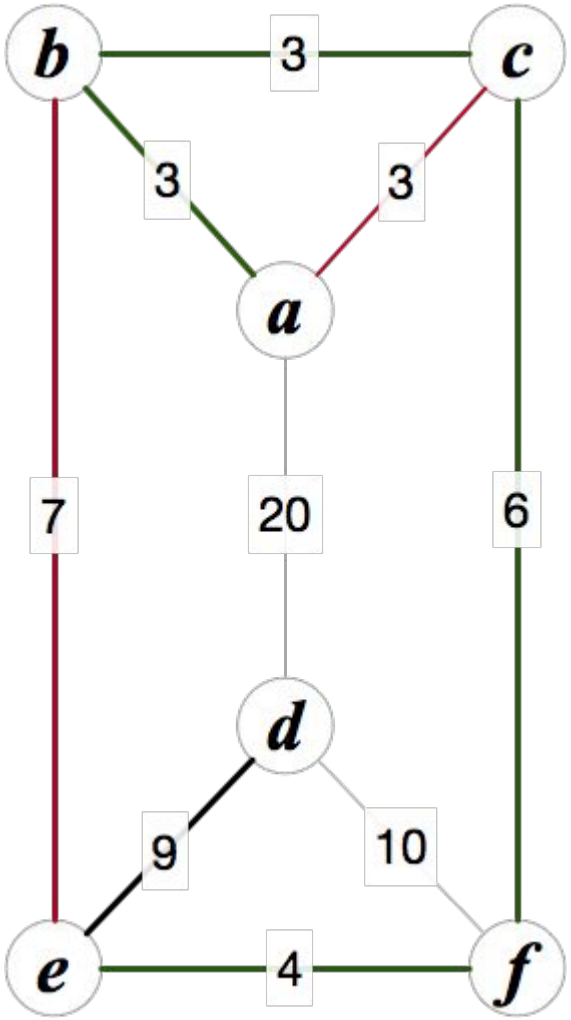
Kruskal's algorithm: greedily add edges

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - Add e to T if it does not create a cycle



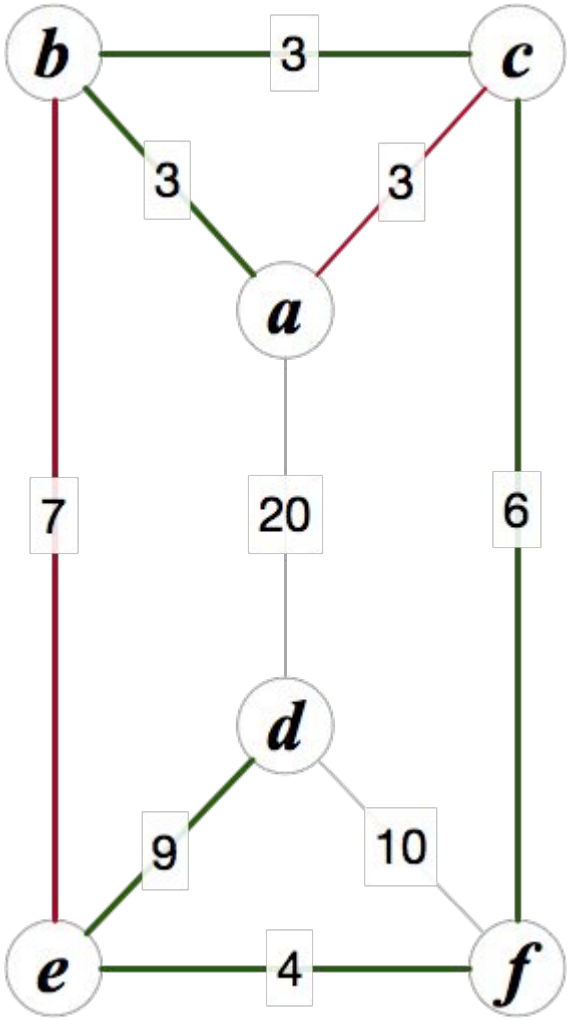
Kruskal's algorithm: greedily add edges

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - Add e to T if it does not create a cycle



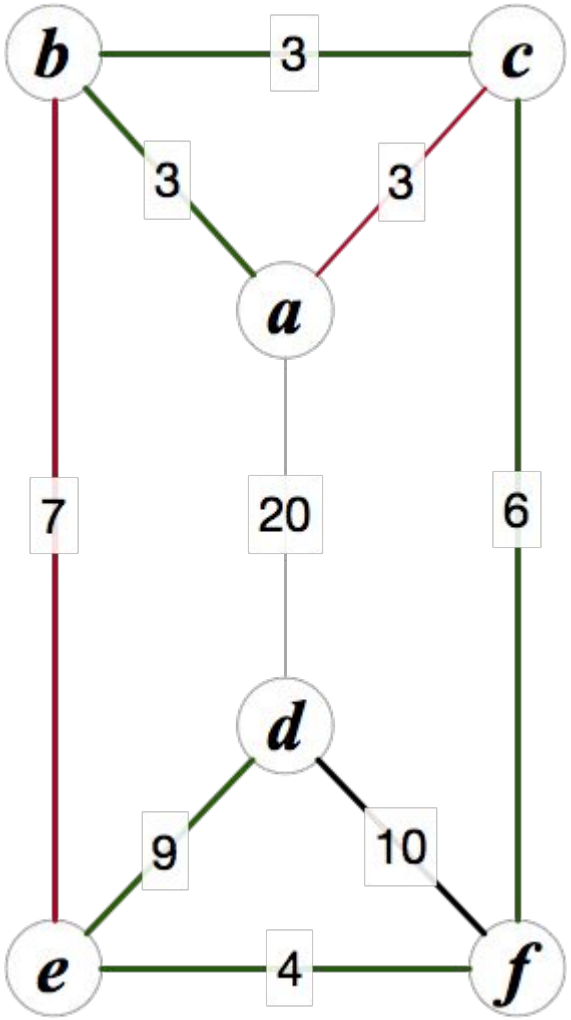
Kruskal's algorithm: greedily add edges

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - Add e to T if it does not create a cycle



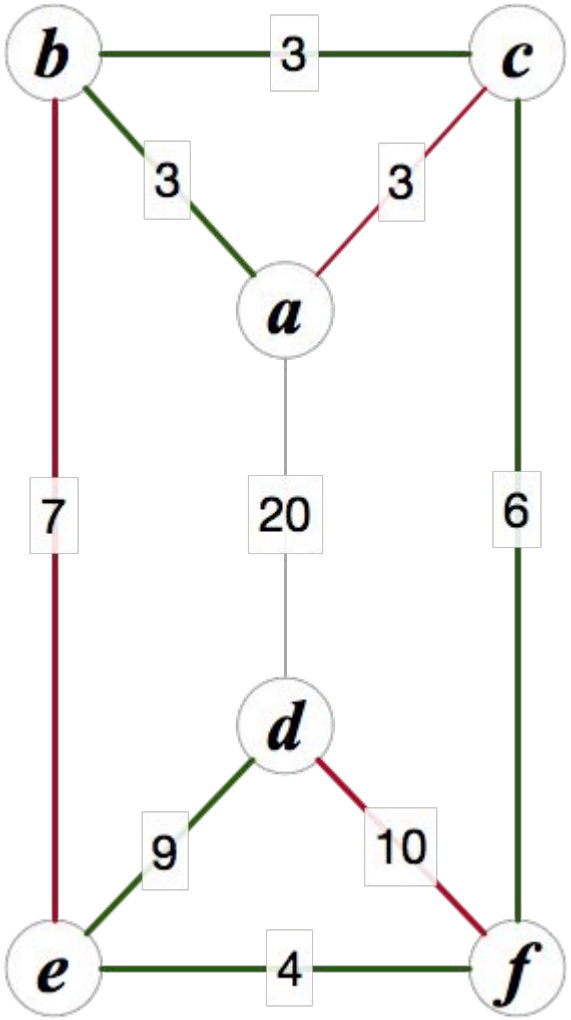
Kruskal's algorithm: greedily add edges

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - Add e to T if it does not create a cycle



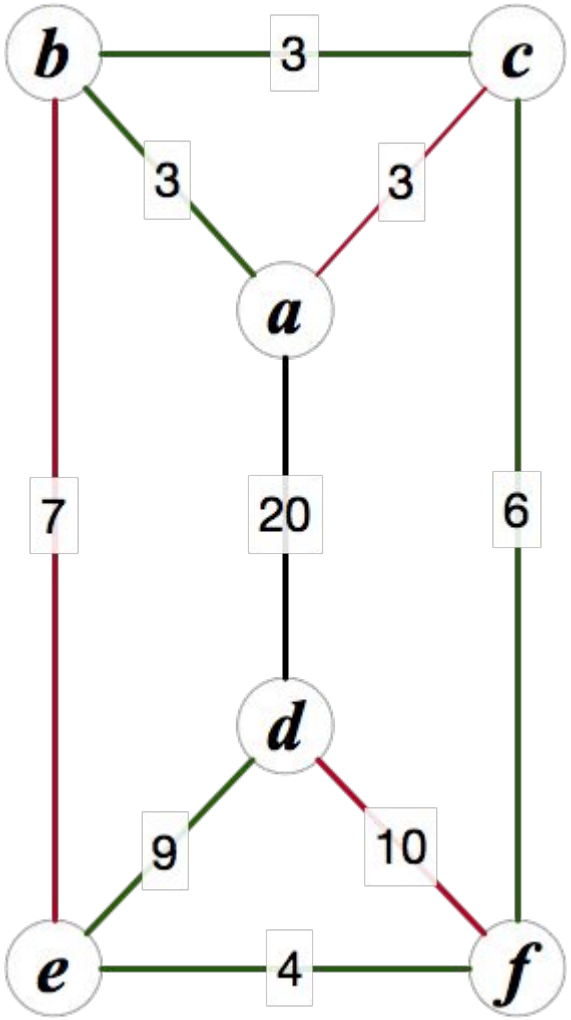
Kruskal's algorithm: greedily add edges

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - Add e to T if it does not create a cycle



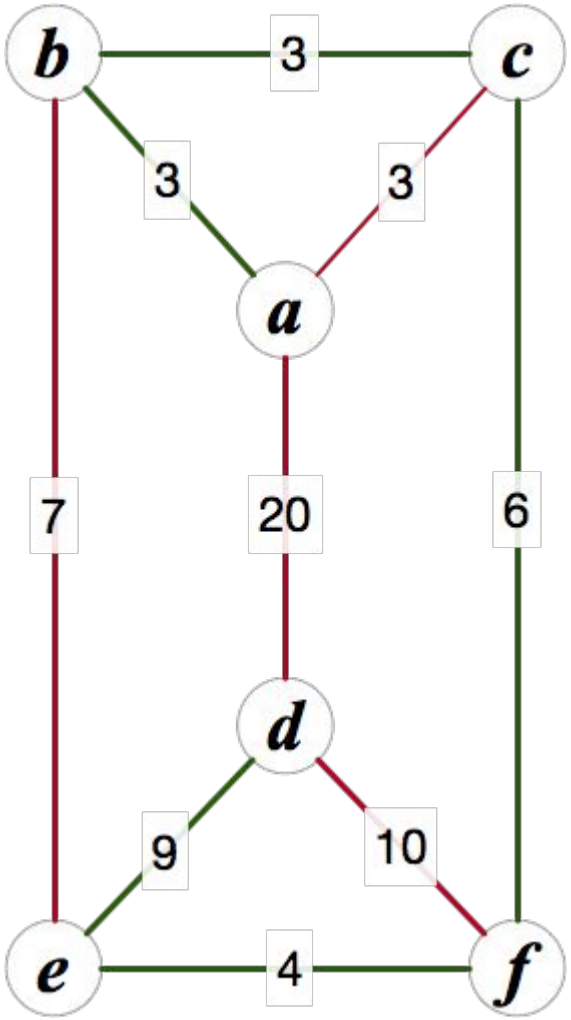
Kruskal's algorithm: greedily add edges

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - Add e to T if it does not create a cycle



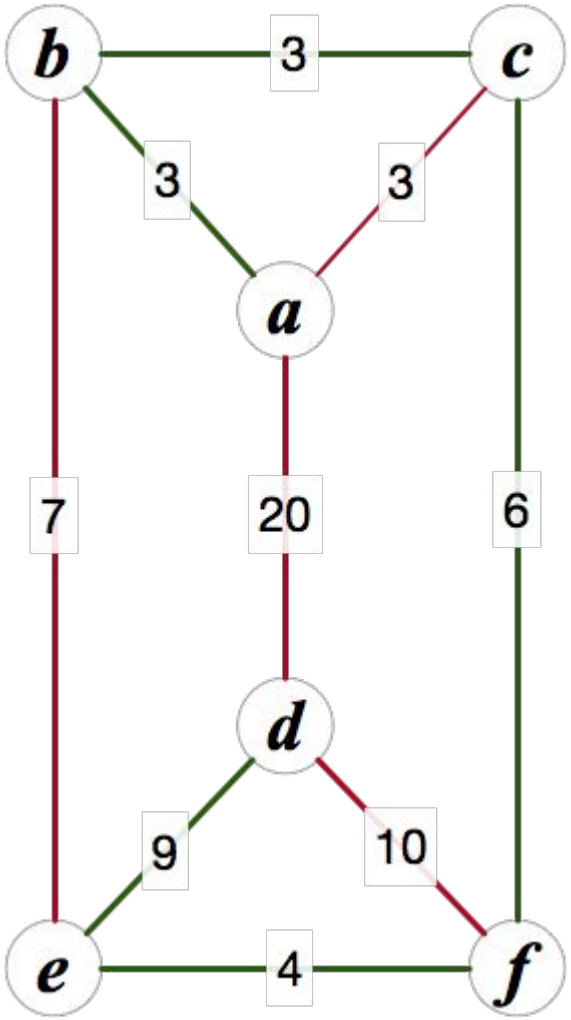
Kruskal's algorithm: greedily add edges

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - Add e to T if it does not create a cycle



Kruskal's algorithm: greedily add edges

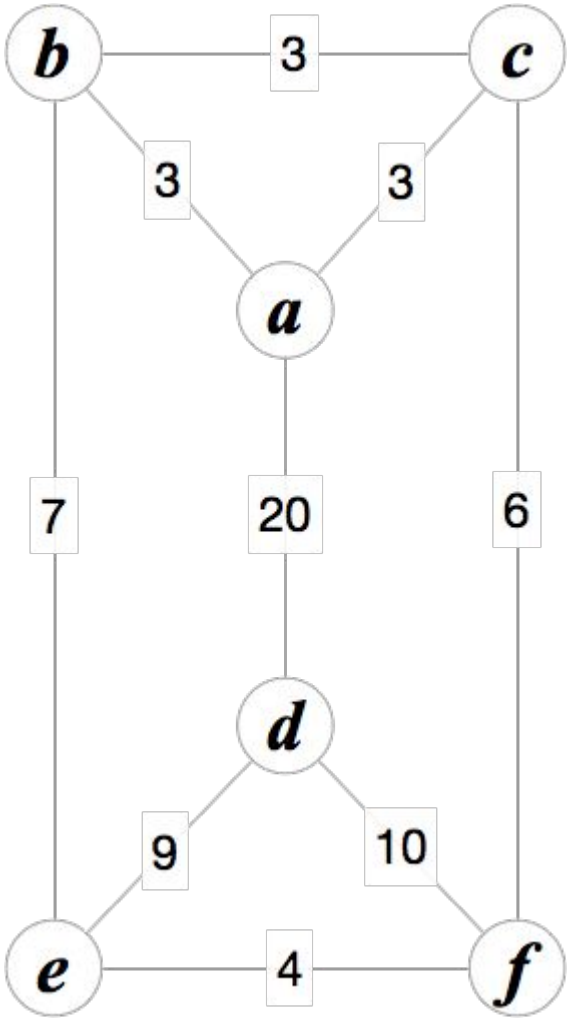
- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - Add e to T if it does not create a cycle



Total cost: 3 + 3 + 4 + 6 + 9 = **25**

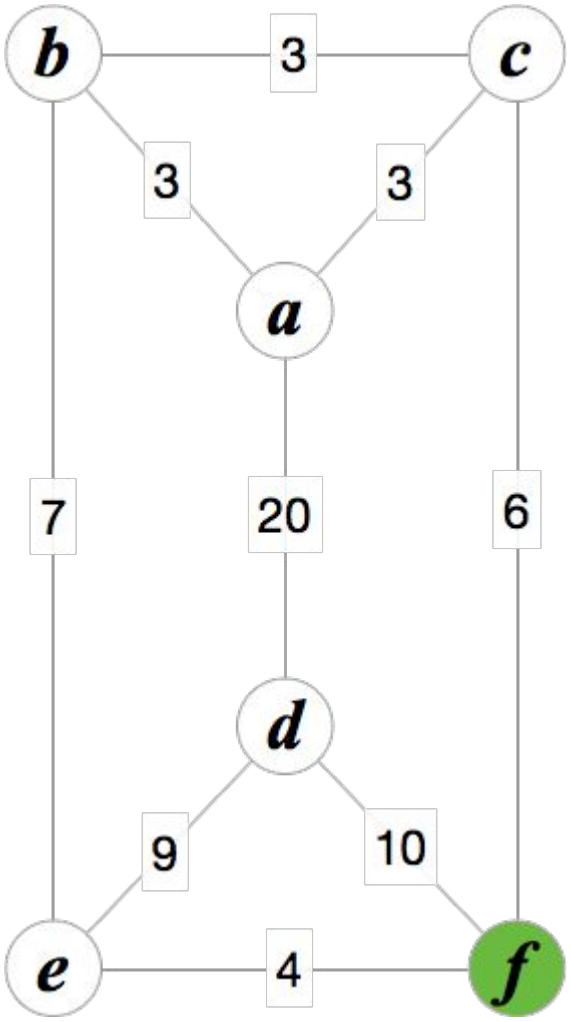
Prim's algorithm: greedily add vertices

pick a root vertex
initialize T = {}
while (true)
    if T is spanning or
    if there is no vertex connected by an edge to T
        return T
    else, of vertices connected by an edge to T,
        add vertex v with minimum edge cost



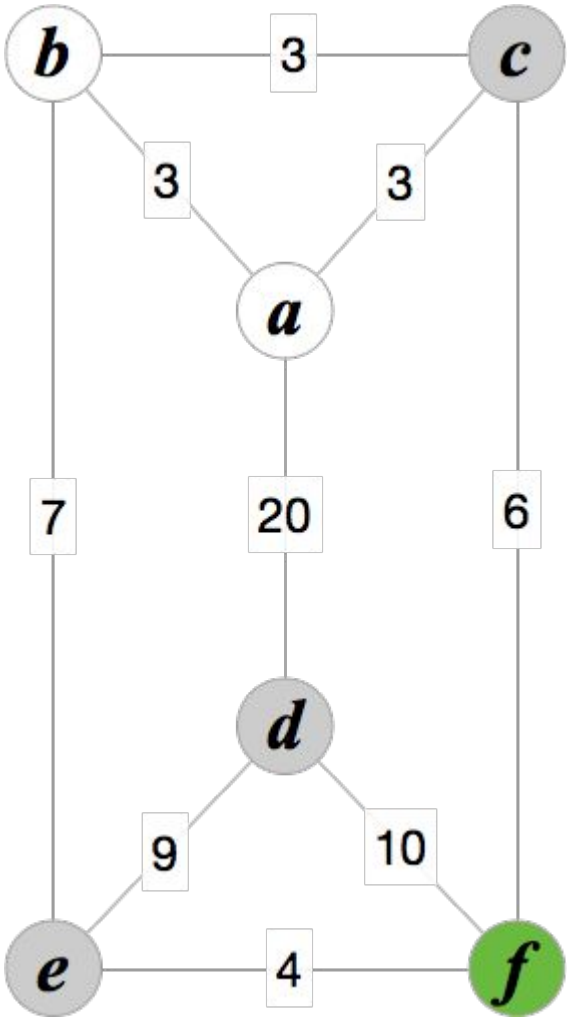
Prim's algorithm: greedily add vertices

pick a root vertex
initialize T = {}
while (true)
    if T is spanning or
    if there is no vertex connected by an edge to T
        return T
    else, of vertices connected by an edge to T,
        add vertex v with minimum edge cost



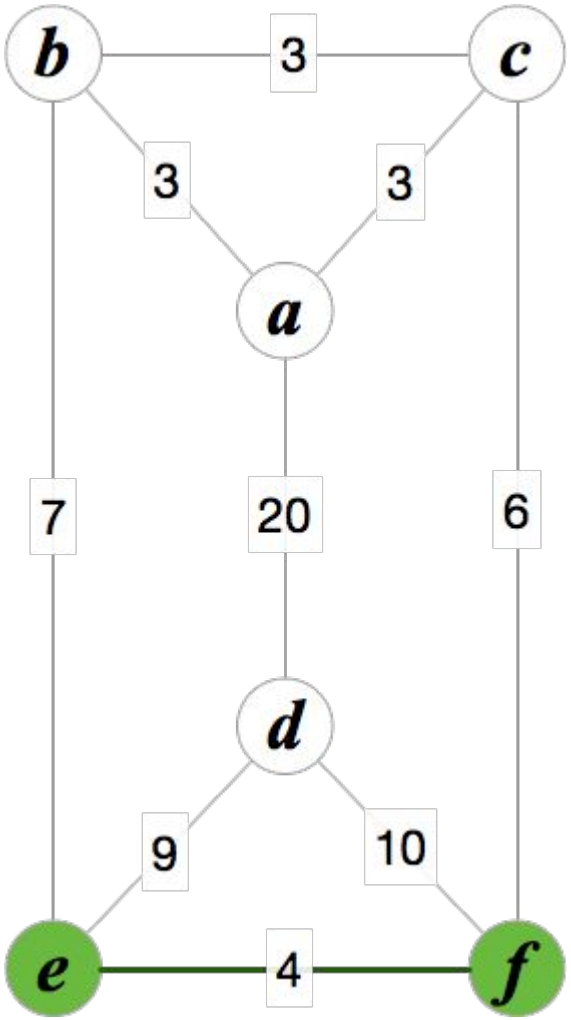
Prim's algorithm: greedily add vertices

pick a root vertex
initialize T = {}
while (true)
    if T is spanning or
    if there is no vertex connected by an edge to T
        return T
    else, of vertices connected by an edge to T,
        add vertex v with minimum edge cost



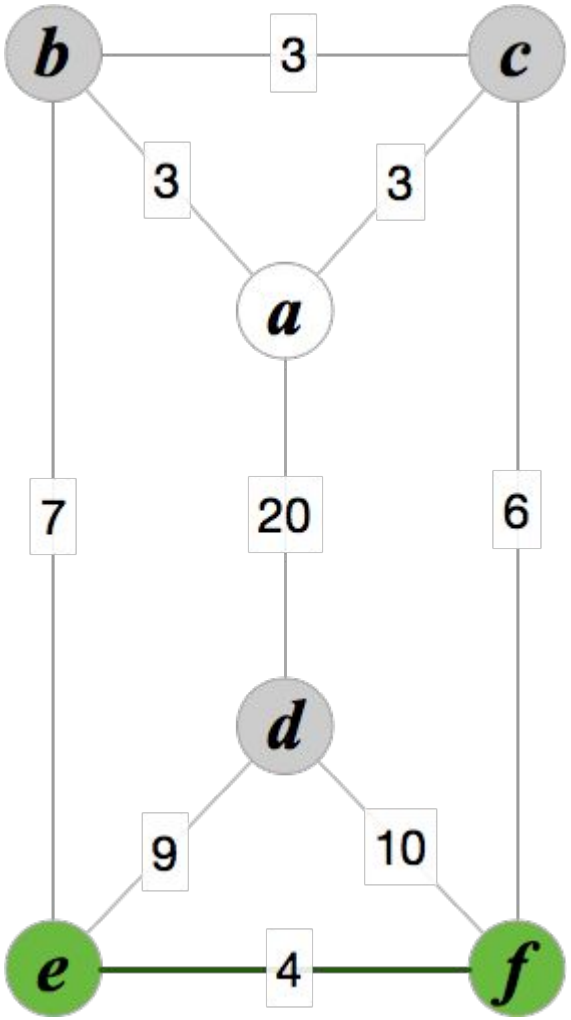
Prim's algorithm: greedily add vertices

pick a root vertex
initialize T = {}
while (true)
    if T is spanning or
    if there is no vertex connected by an edge to T
        return T
    else, of vertices connected by an edge to T,
        add vertex v with minimum edge cost



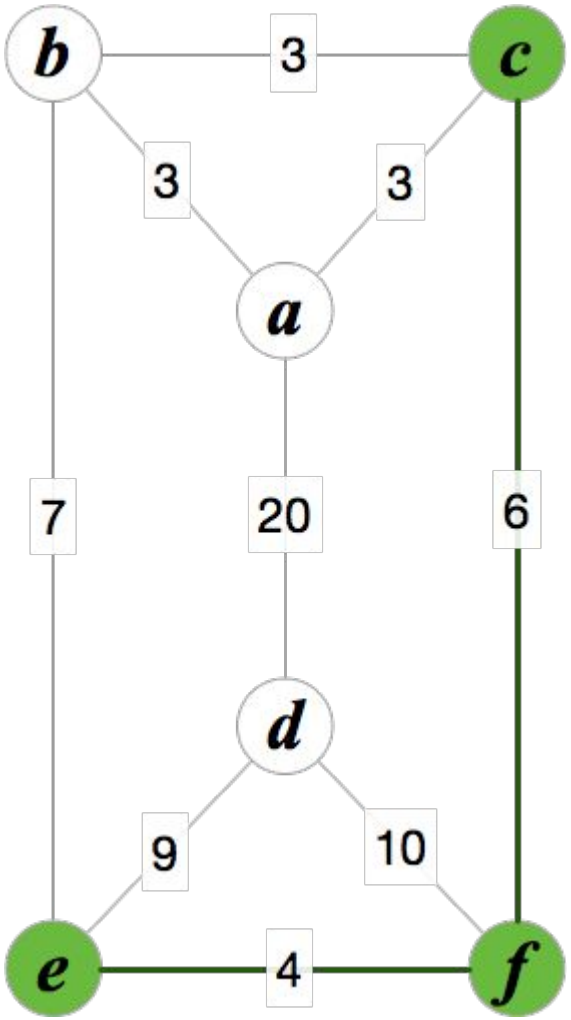
Prim's algorithm: greedily add vertices

pick a root vertex
initialize T = {}
while (true)
    if T is spanning or
    if there is no vertex connected by an edge to T
        return T
    else, of vertices connected by an edge to T,
        add vertex v with minimum edge cost



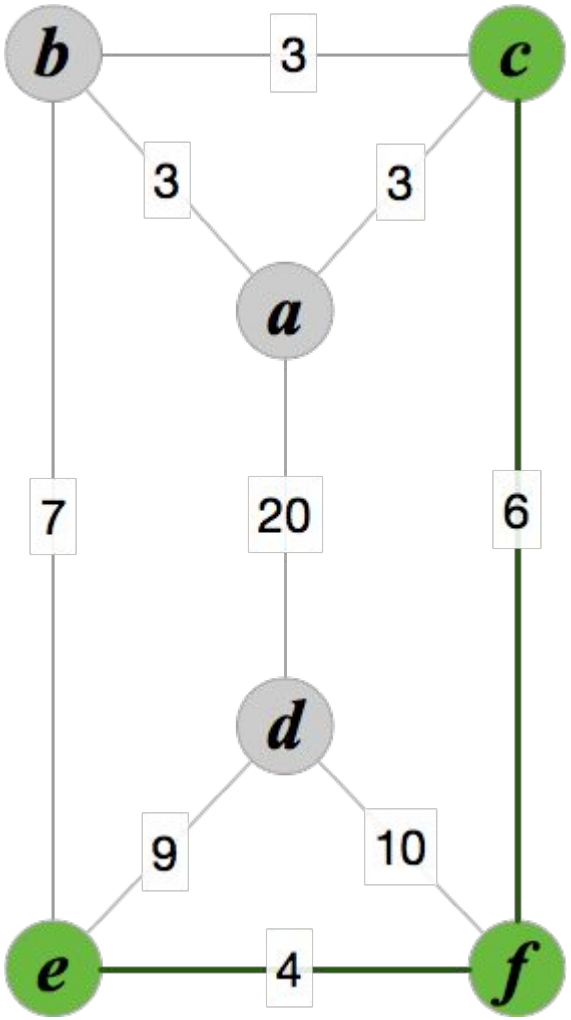
Prim's algorithm: greedily add vertices

pick a root vertex
initialize T = {}
while (true)
    if T is spanning or
    if there is no vertex connected by an edge to T
        return T
    else, of vertices connected by an edge to T,
        add vertex v with minimum edge cost



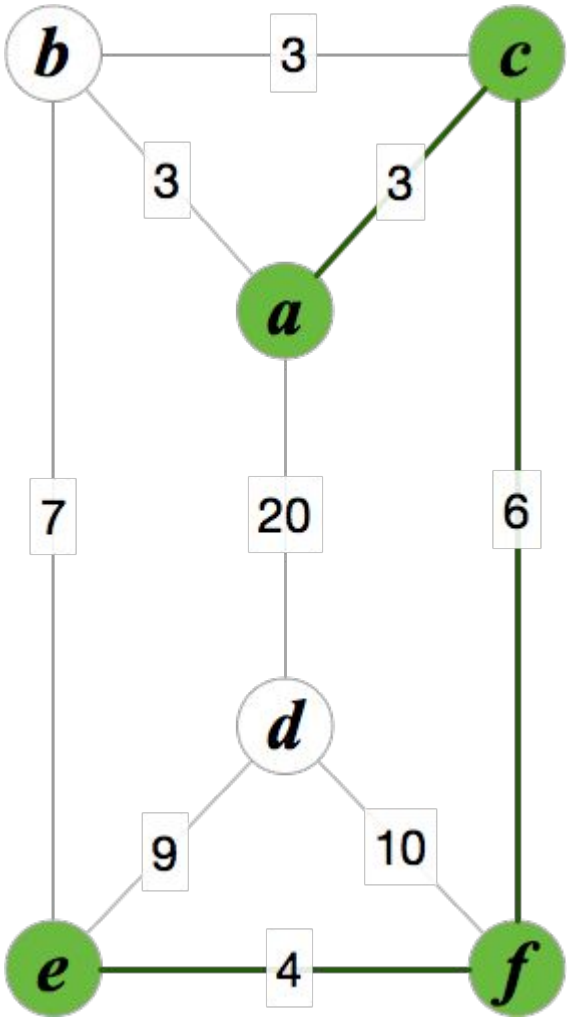
Prim's algorithm: greedily add vertices

pick a root vertex
initialize T = {}
while (true)
    if T is spanning or
    if there is no vertex connected by an edge to T
        return T
    else, of vertices connected by an edge to T,
        add vertex v with minimum edge cost



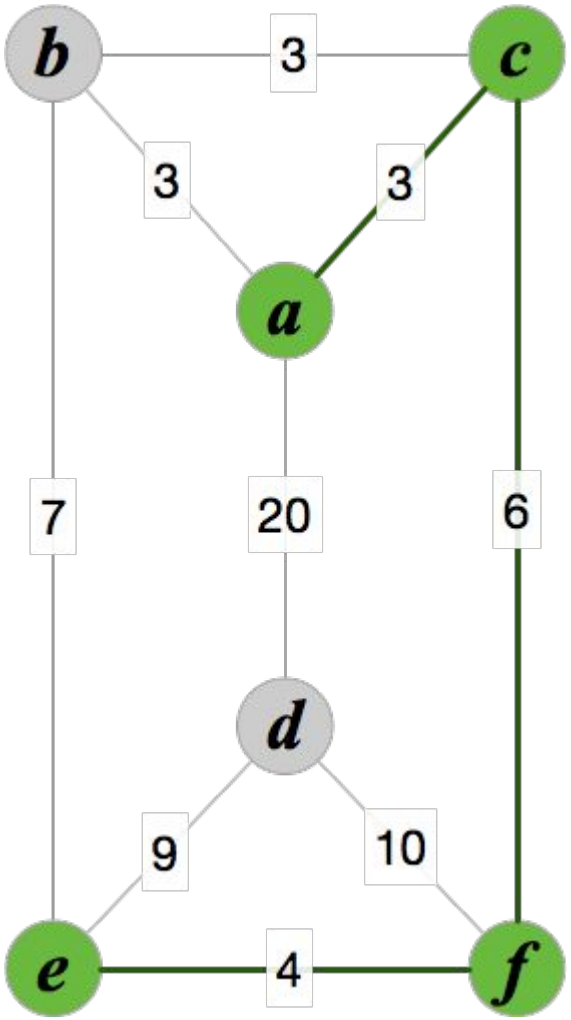
Prim's algorithm: greedily add vertices

pick a root vertex
initialize T = {}
while (true)
    if T is spanning or
    if there is no vertex connected by an edge to T
        return T
    else, of vertices connected by an edge to T,
        add vertex v with minimum edge cost



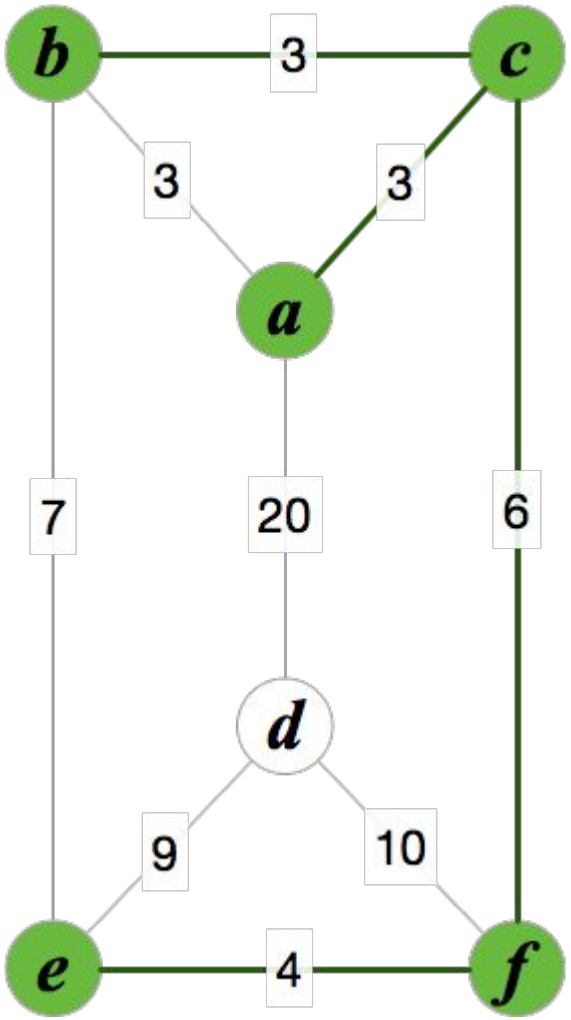
Prim's algorithm: greedily add vertices

pick a root vertex
initialize T = {}
while (true)
    if T is spanning or
    if there is no vertex connected by an edge to T
        return T
    else, of vertices connected by an edge to T,
        add vertex v with minimum edge cost



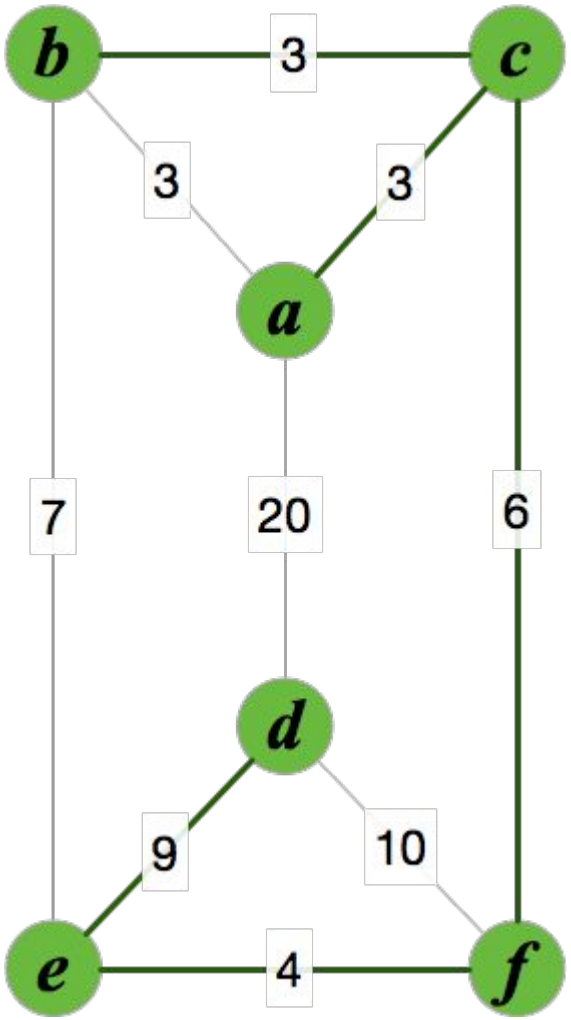
Prim's algorithm: greedily add vertices

pick a root vertex
initialize T = {}
while (true)
    if T is spanning or
    if there is no vertex connected by an edge to T
        return T
    else, of vertices connected by an edge to T,
        add vertex v with minimum edge cost



Prim's algorithm: greedily add vertices

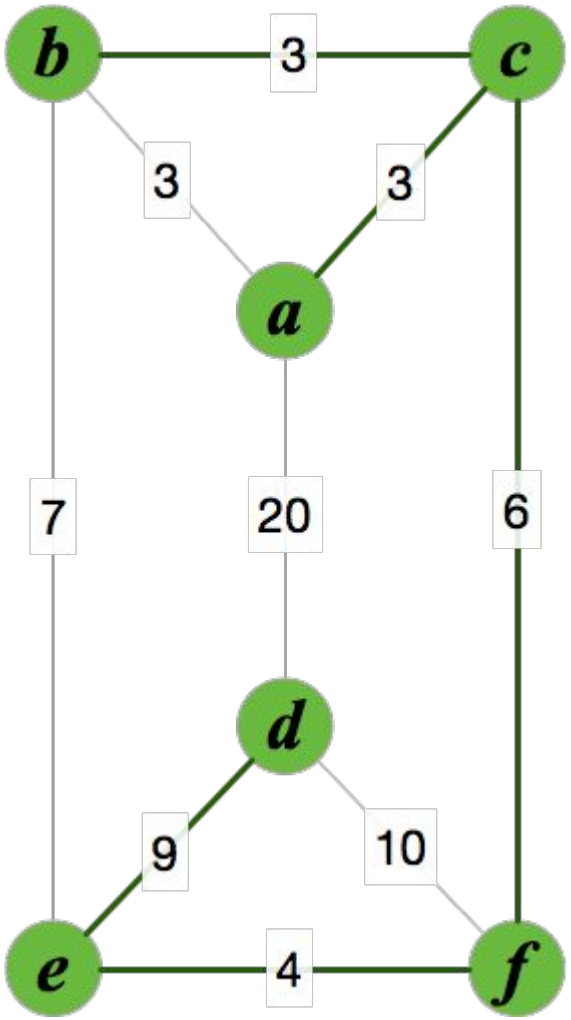
pick a root vertex
initialize T = {}
while (true)
    if T is spanning or
    if there is no vertex connected by an edge to T
        return T
    else, of vertices connected by an edge to T,
        add vertex v with minimum edge cost



Prim's algorithm: greedily add vertices

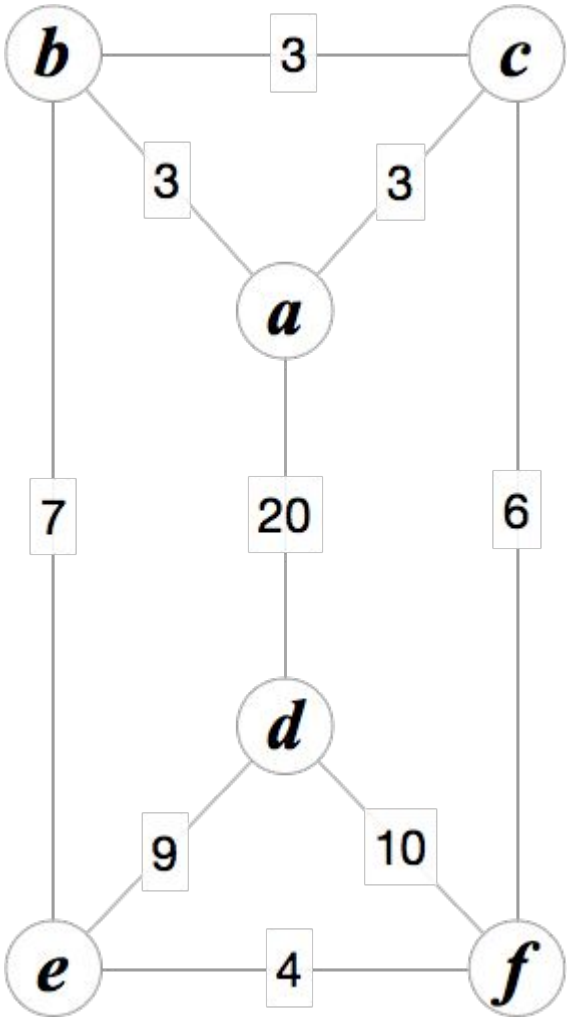
pick a root vertex
initialize $T = \{\}$
while (true)
    if T is spanning or
    if there is no vertex connected by an edge to T
        return T
    else, of vertices connected by an edge to T,
        add vertex v with minimum edge cost

Total cost: 4 + 6 + 3 + 3 + 9 = **25**



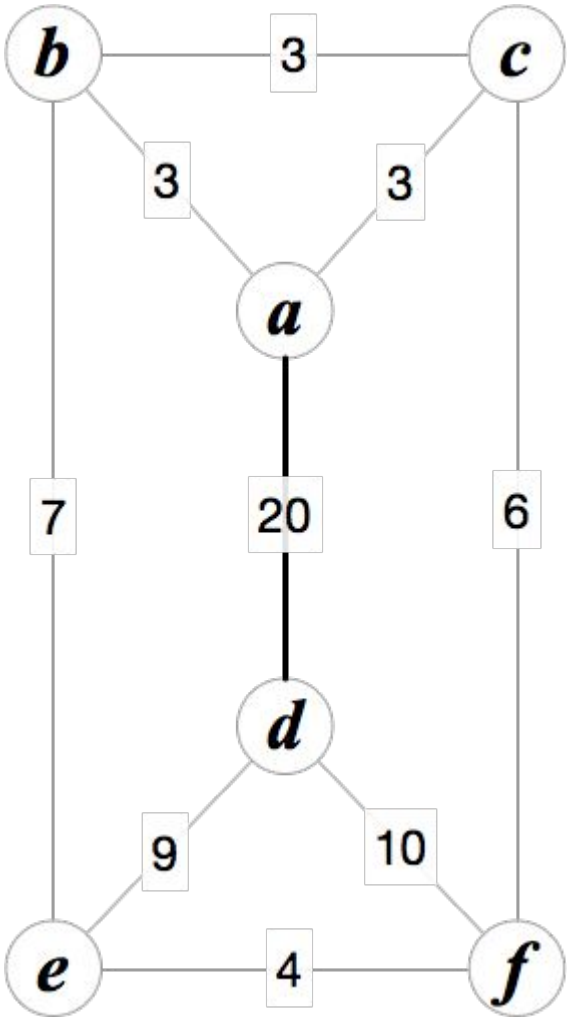
Reverse-Delete

- Sort edges by decreasing cost
- Initialize $T = E$
- For each edge e
  - Remove e from T if it does not disconnect



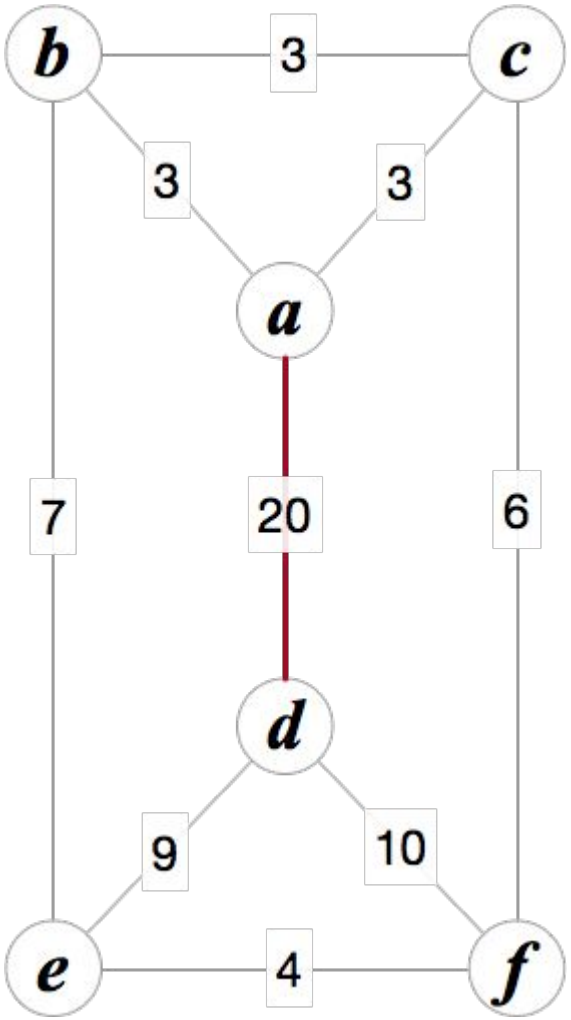
Reverse-Delete

- Sort edges by decreasing cost
- Initialize $T = E$
- For each edge e
  - Remove e from T if it does not disconnect



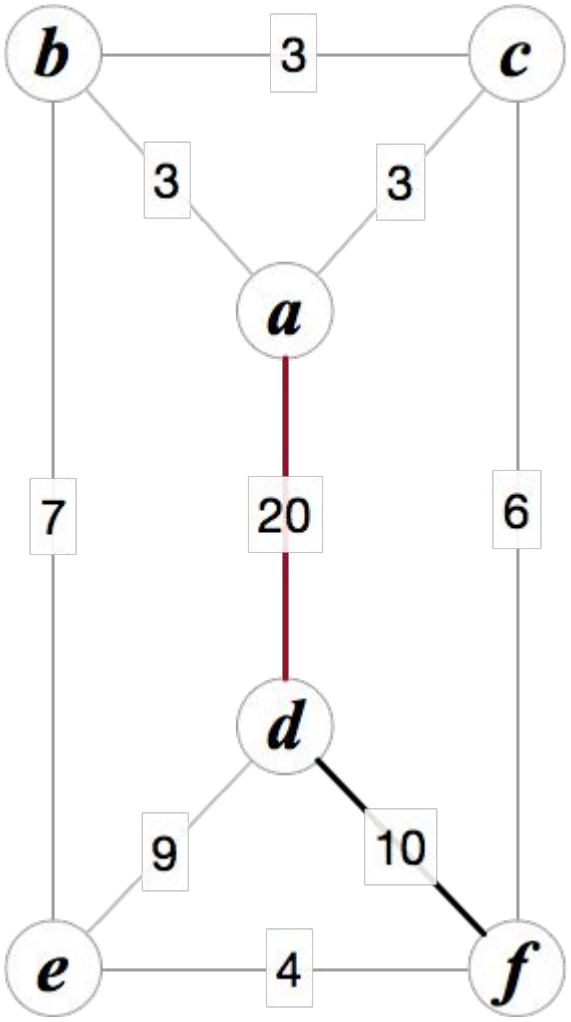
Reverse-Delete

- Sort edges by decreasing cost
- Initialize $T = E$
- For each edge e
  - Remove e from T if it does not disconnect



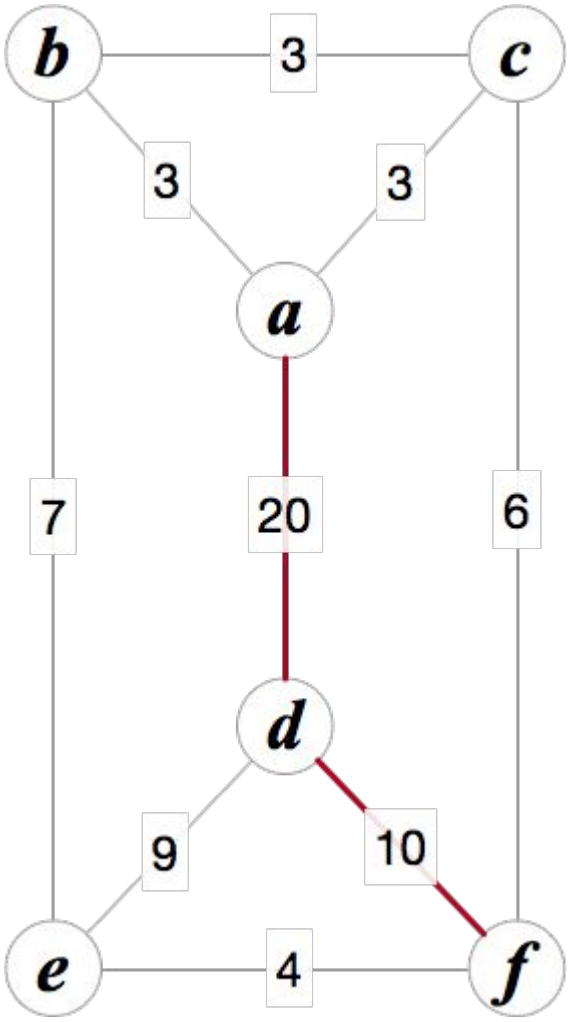
Reverse-Delete

- Sort edges by decreasing cost
- Initialize $T = E$
- For each edge e
  - Remove e from T if it does not disconnect



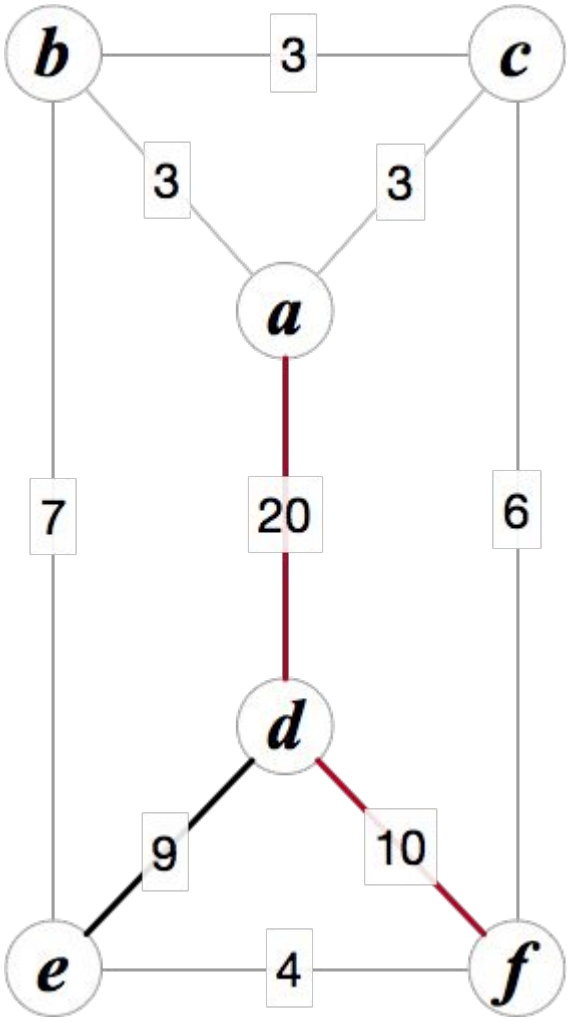
Reverse-Delete

- Sort edges by decreasing cost
- Initialize $T = E$
- For each edge e
  - Remove e from T if it does not disconnect



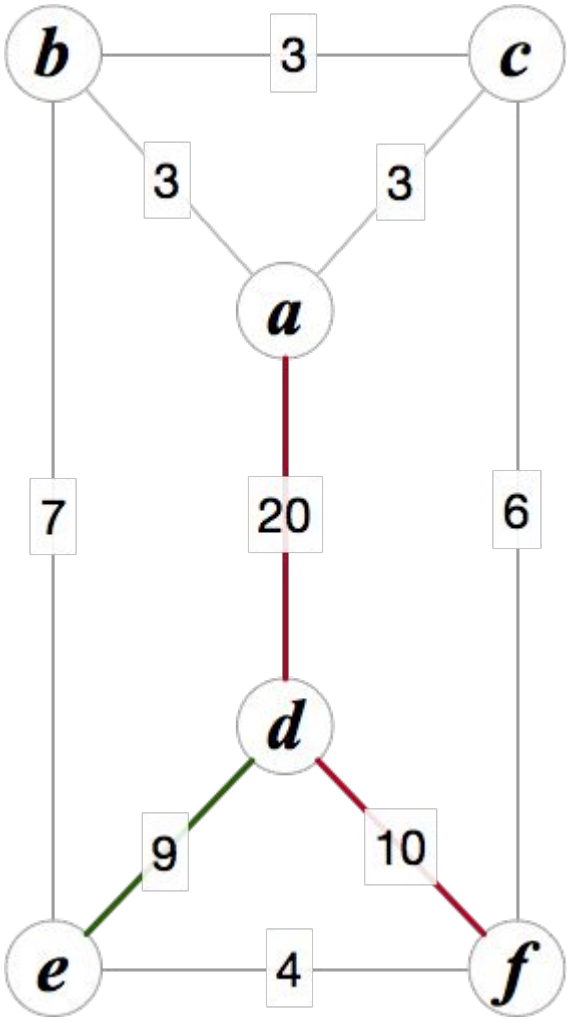
Reverse-Delete

- Sort edges by decreasing cost
- Initialize $T = E$
- For each edge e
  - Remove e from T if it does not disconnect



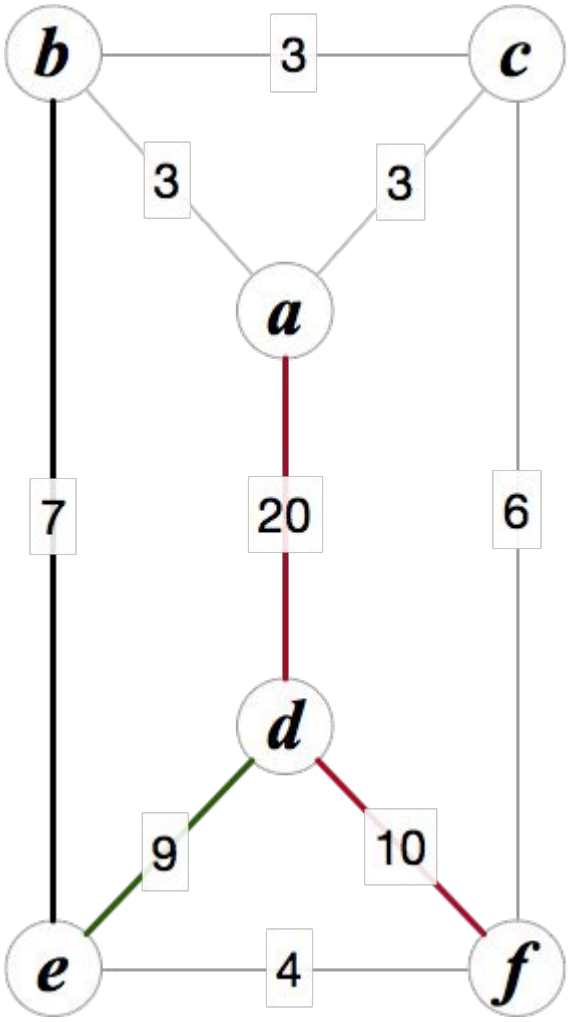
Reverse-Delete

- Sort edges by decreasing cost
- Initialize $T = E$
- For each edge e
  - Remove e from T if it does not disconnect



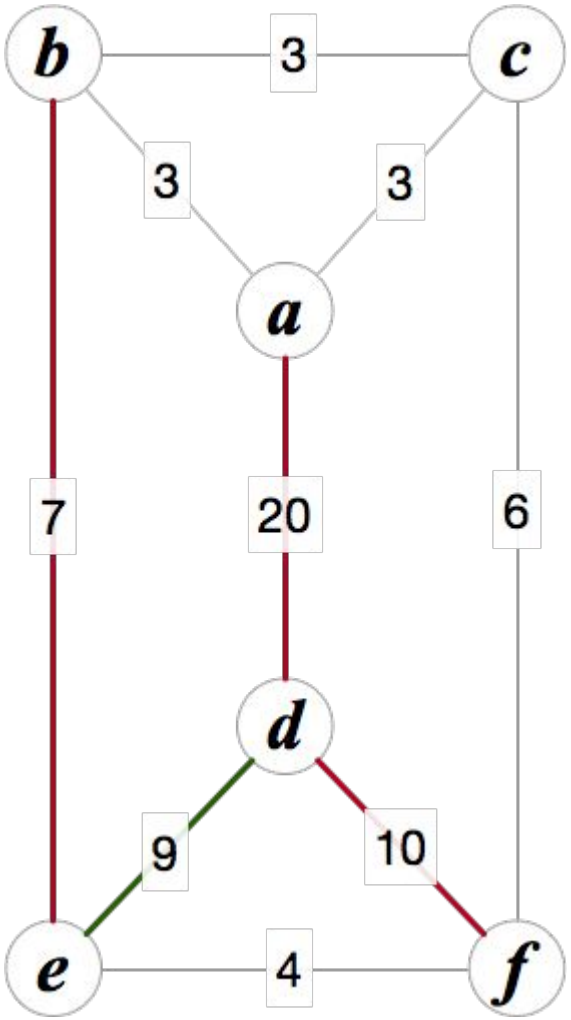
Reverse-Delete

- Sort edges by decreasing cost
- Initialize $T = E$
- For each edge e
  - Remove e from T if it does not disconnect



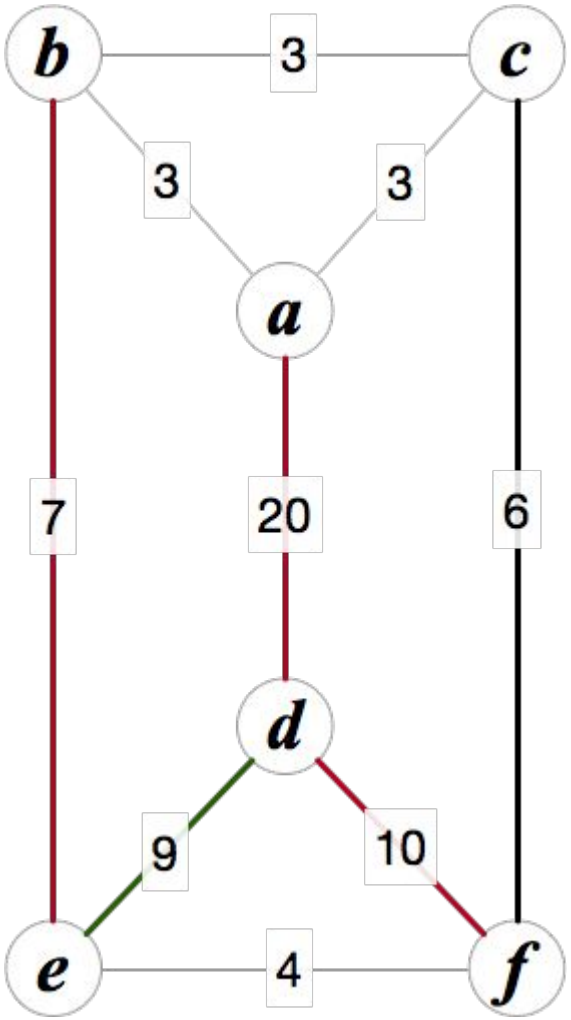
Reverse-Delete

- Sort edges by decreasing cost
- Initialize $T = E$
- For each edge e
  - Remove e from T if it does not disconnect



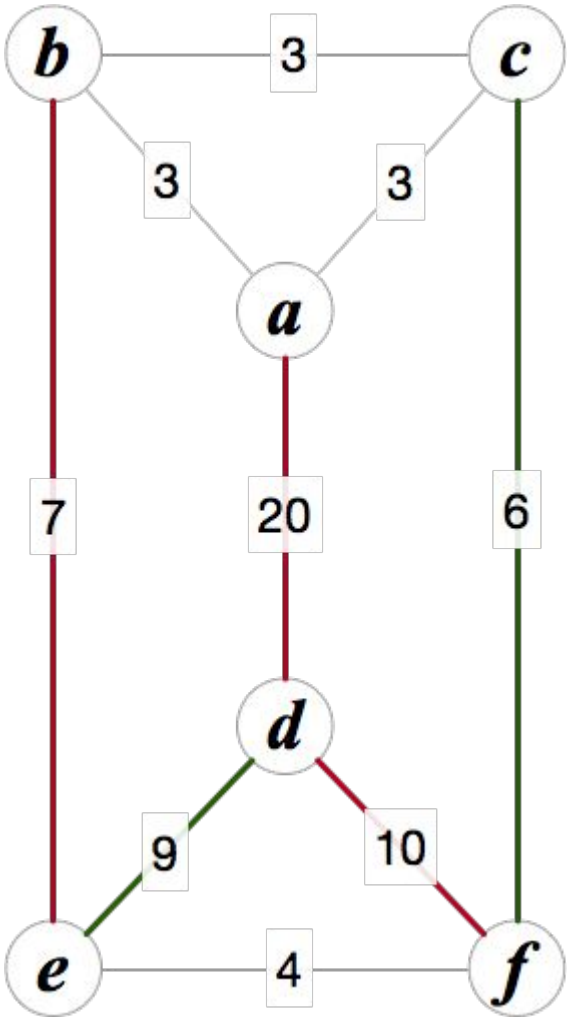
Reverse-Delete

- Sort edges by decreasing cost
- Initialize $T = E$
- For each edge e
  - Remove e from T if it does not disconnect



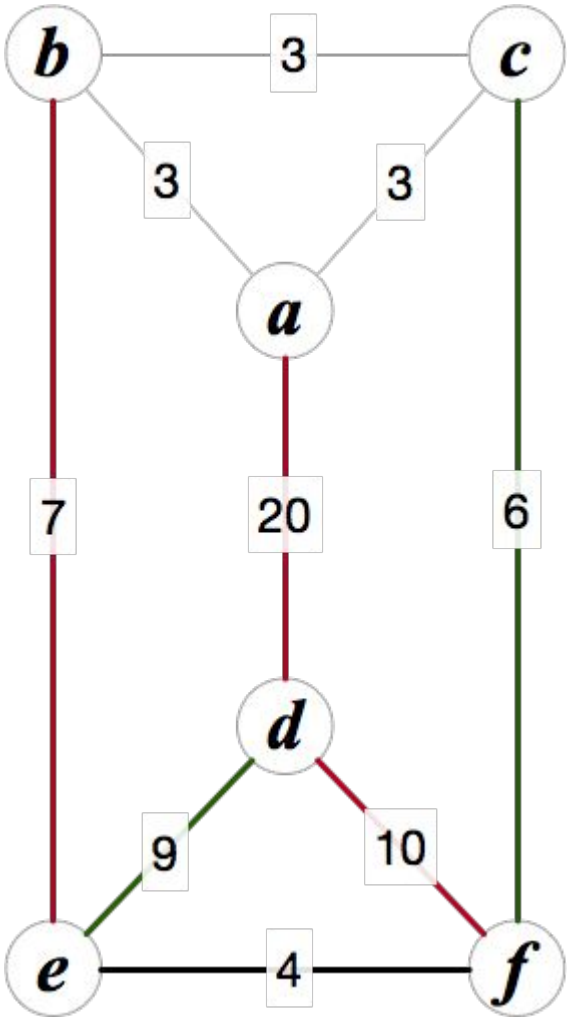
Reverse-Delete

- Sort edges by decreasing cost
- Initialize $T = E$
- For each edge e
  - Remove e from T if it does not disconnect



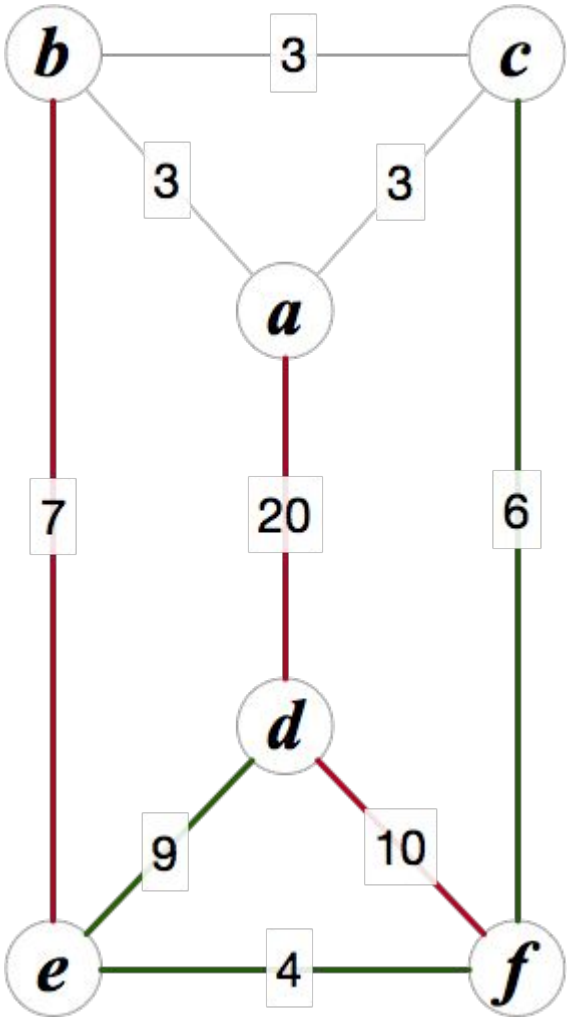
Reverse-Delete

- Sort edges by decreasing cost
- Initialize $T = E$
- For each edge e
  - Remove e from T if it does not disconnect



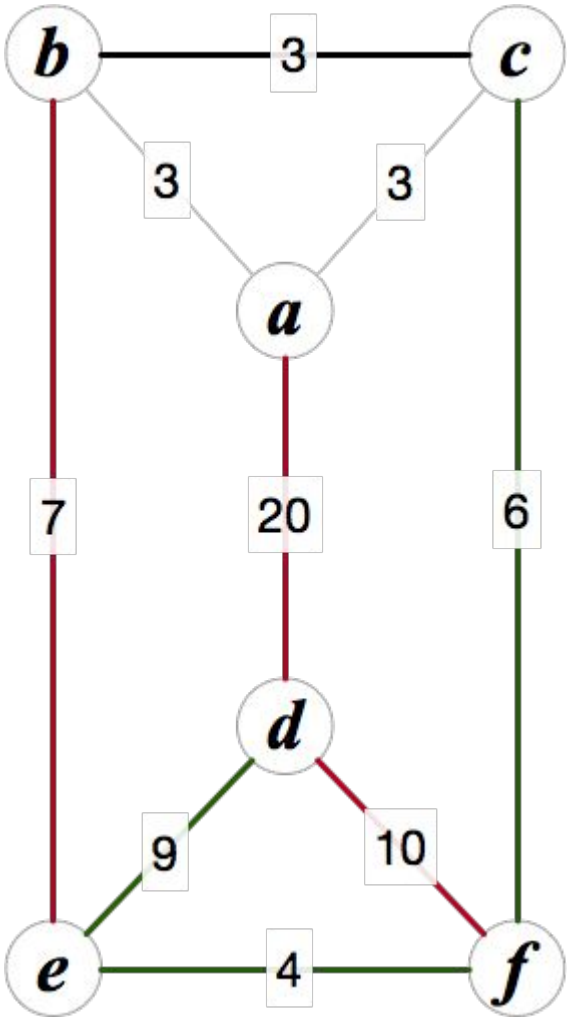
Reverse-Delete

- Sort edges by decreasing cost
- Initialize $T = E$
- For each edge e
  - Remove e from T if it does not disconnect



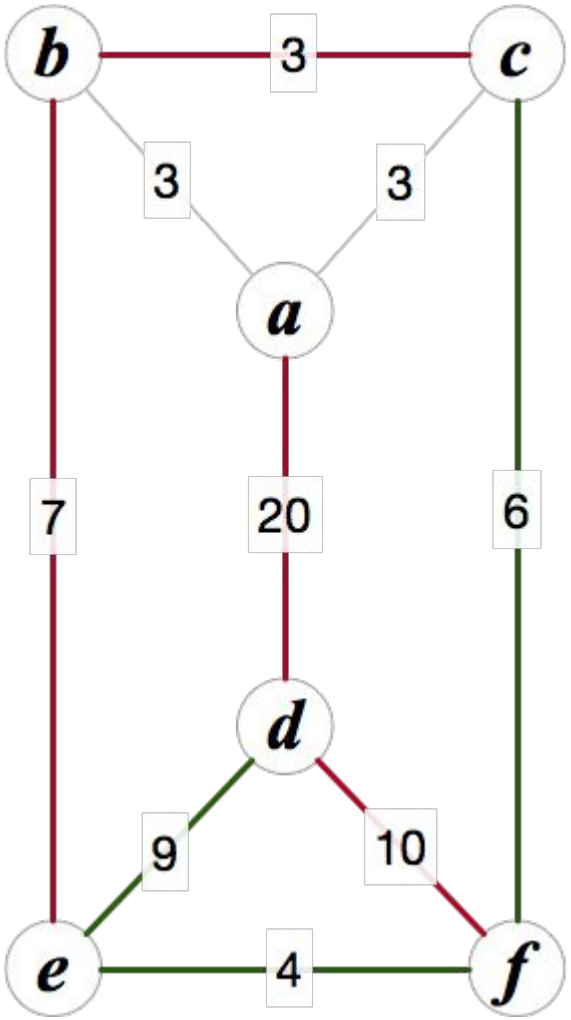
Reverse-Delete

- Sort edges by decreasing cost
- Initialize $T = E$
- For each edge e
  - Remove e from T if it does not disconnect



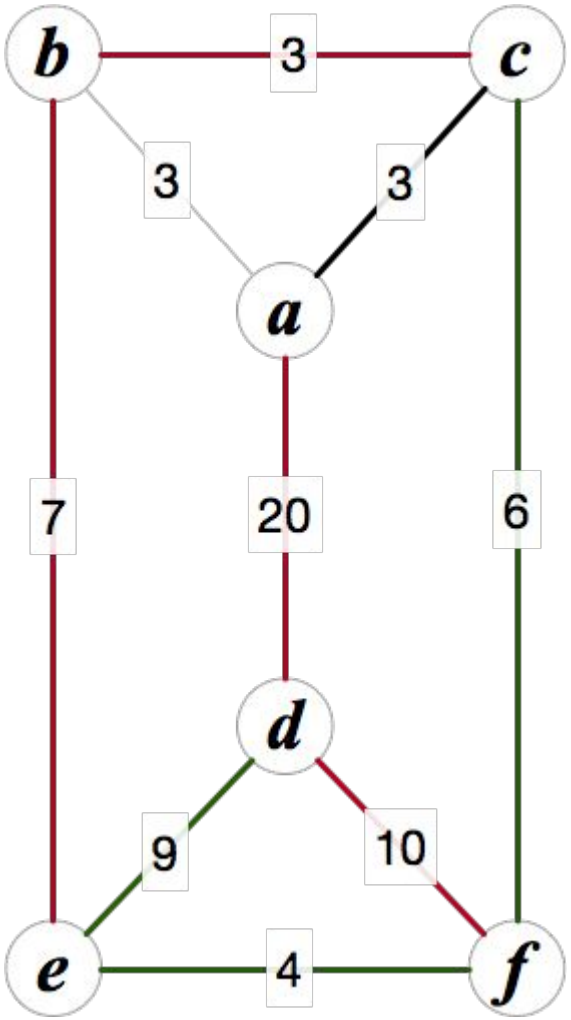
Reverse-Delete

- Sort edges by decreasing cost
- Initialize $T = E$
- For each edge e
  - Remove e from T if it does not disconnect



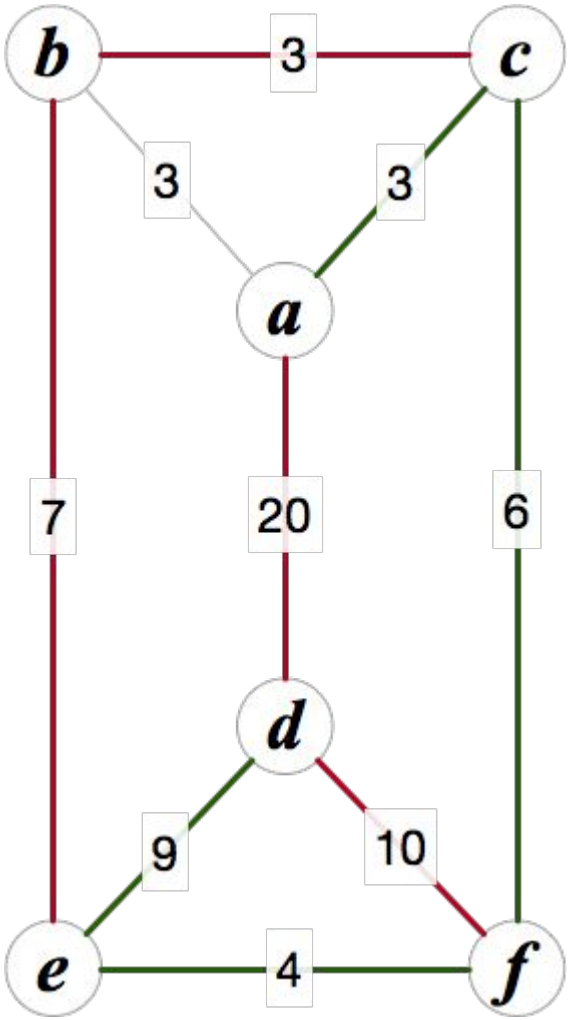
Reverse-Delete

- Sort edges by decreasing cost
- Initialize $T = E$
- For each edge e
  - Remove e from T if it does not disconnect



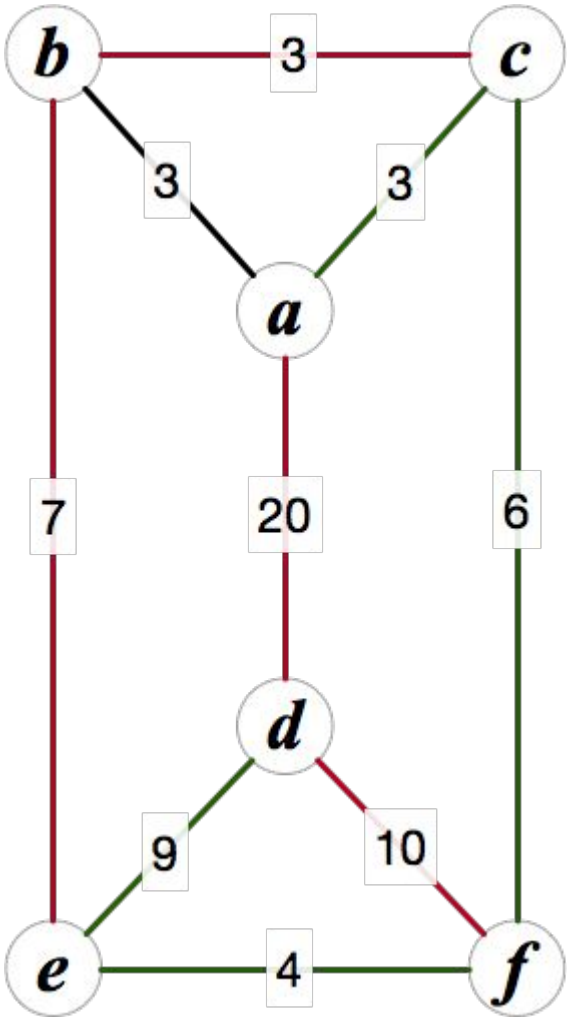
Reverse-Delete

- Sort edges by decreasing cost
- Initialize $T = E$
- For each edge e
  - Remove e from T if it does not disconnect



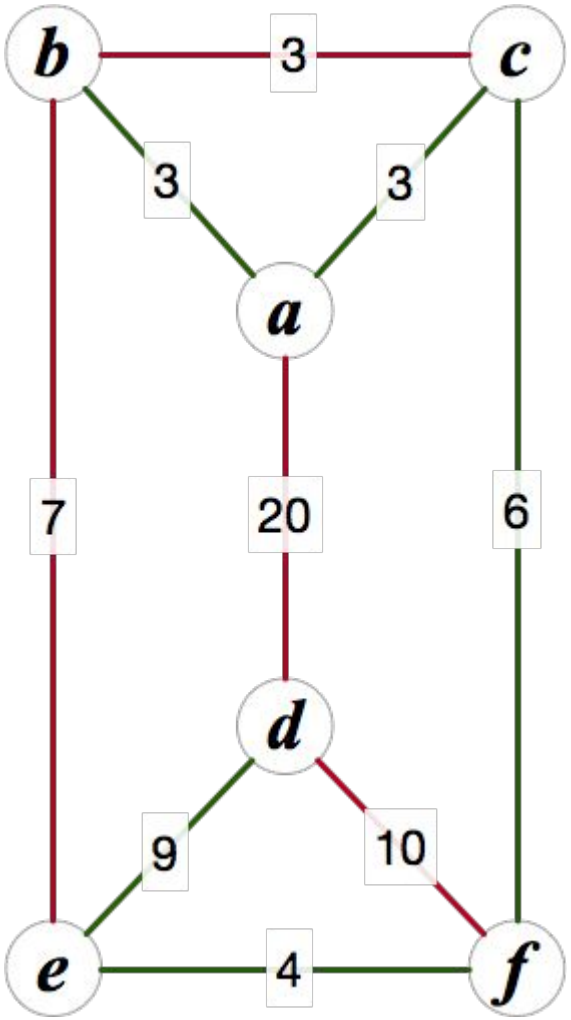
Reverse-Delete

- Sort edges by decreasing cost
- Initialize $T = E$
- For each edge e
  - Remove e from T if it does not disconnect



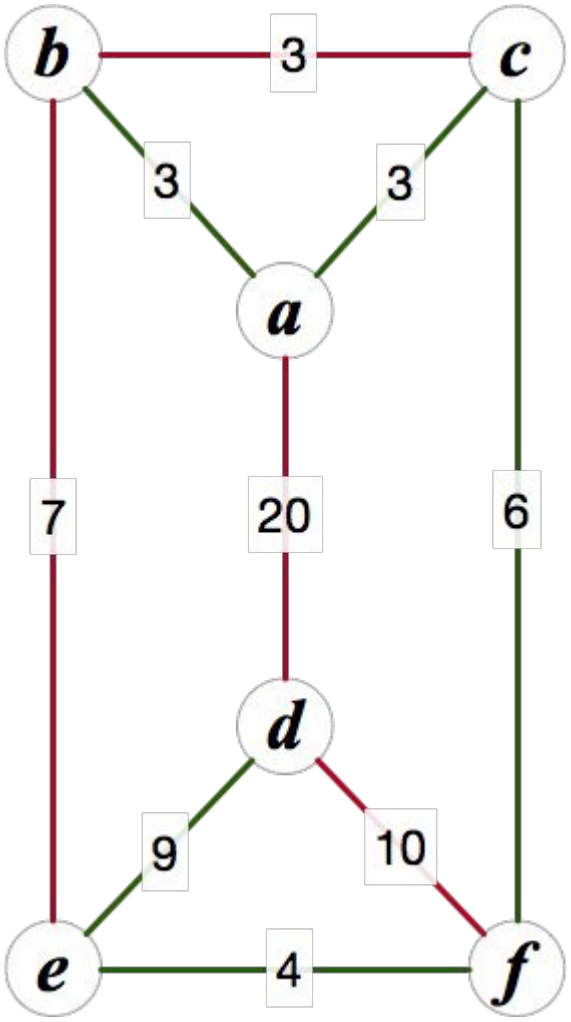
Reverse-Delete

- Sort edges by decreasing cost
- Initialize $T = E$
- For each edge e
  - Remove e from T if it does not disconnect



Reverse-Delete

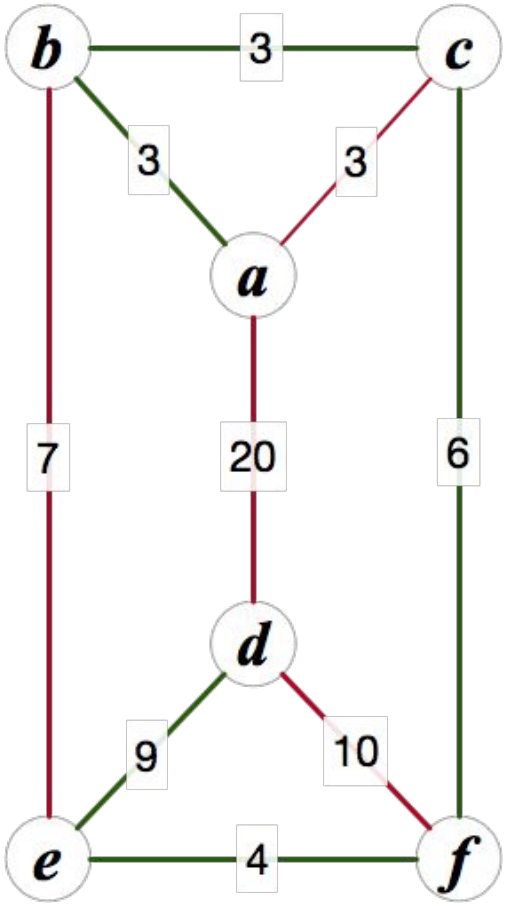
- Sort edges by decreasing cost
- Initialize $T = E$
- For each edge e
  - Remove e from T if it does not disconnect



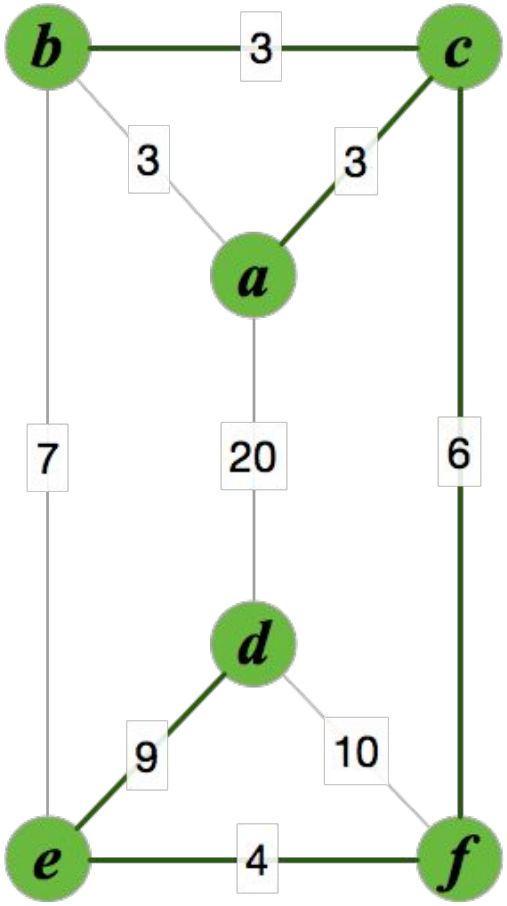
Total cost: 3 + 3 + 4 + 6 + 9 = **25**

They all produce a MST… always?

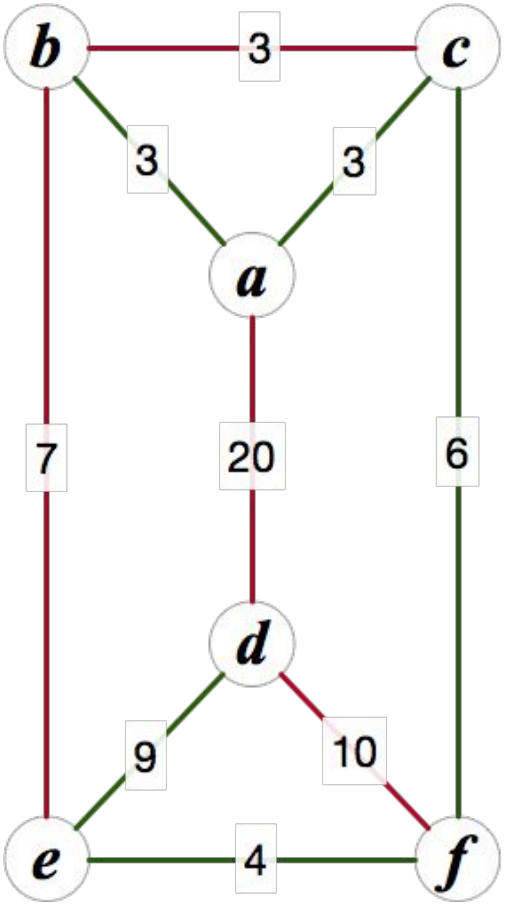
Kruskal's



Prim's



Reverse-Delete



Step 3: prove correctness

Via matroids

(bonus topic)

Matroids (definition)

Defined on pair $(S, \boldsymbol{I})$

- S: "ground set"
- $\boldsymbol{I}$: "independent sets"; $\boldsymbol{I} \subseteq \mathscr{P}(S)$; set of subsets of S

Satisfying:

1. S is finite set
2. **hereditary**: if $B \in \boldsymbol{I}$ and $A \subseteq B$, then $A \in \boldsymbol{I}$
   any subset of an independent set is independent
3. **exchange**: if $A,B \in \boldsymbol{I}$ and $|A| < |B|$, then $\exists\ x \in B - A$ such that $A \cup \{x\} \in \mathbf{I}$
   if one independent set is *larger* than another,
   there is an *element* from the larger set that can be
   added to the smaller while maintaining independence

Example: graphic matroid

Claim: for a graph $G = (V,E)$, the pair $(E,\boldsymbol{F})$ is a matroid, where

- E: edge set
- $\boldsymbol{F}$: $\{E' \subseteq E \mid E'$ is a forest (has no cycles)$\}$

How does this relate to greedy algorithms?

Theorem 16.6 [CLRS]:
All maximal independent sets in a matroid have the same size.

- An independent set is *maximal* if adding any other element will cause independence to be lost
- (a maximal independent set of a matroid is called a "basis")

In the graphic matroid, this means:

- All maximal forests have the *same size*
- If the graph is *connected*, a maximal forest is a *spanning tree*

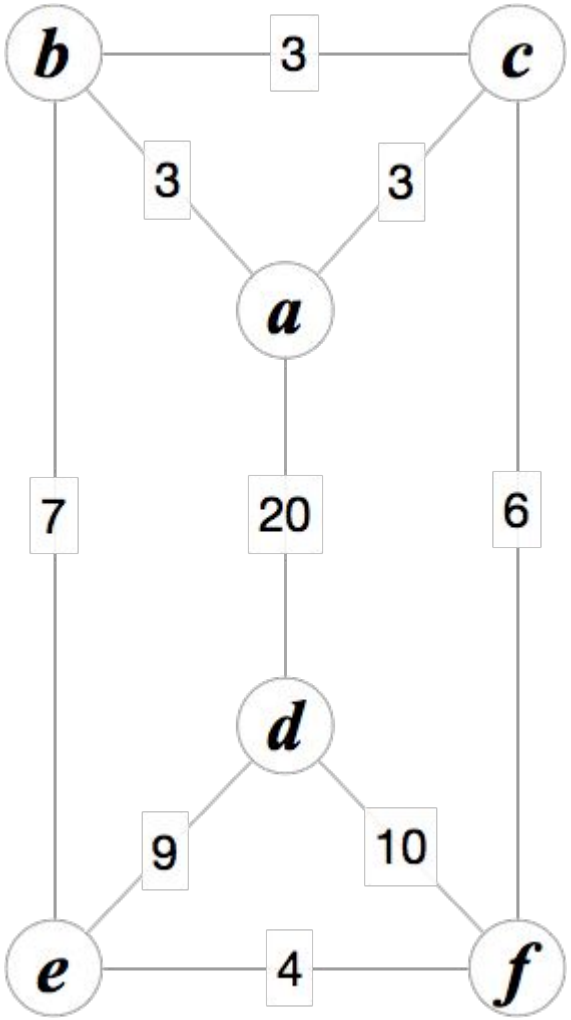
Greedy algorithm correct ⇔ matroidal structure

- In context of finding maximum-sized sets with specified property
  - E.g., no cycles

Step 4: analyze running time

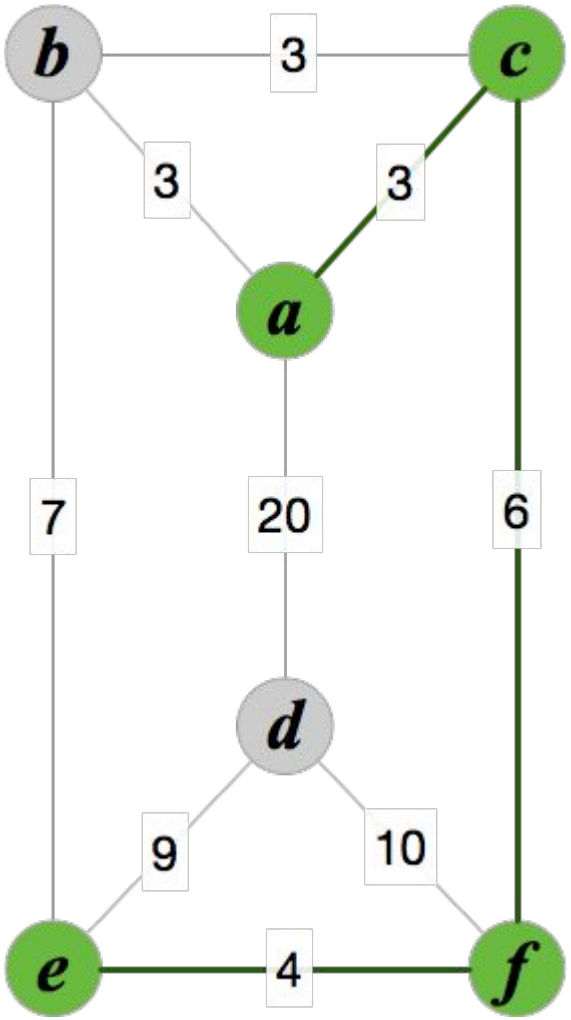
Prim's algorithm: greedily add vertices

pick a root vertex
initialize T = {}
while (true)
    if T is spanning or
    if there is no vertex connected by an edge to T
        return T
    else, of vertices connected by an edge to T,
        add vertex v with minimum edge cost



Prim's algorithm: greedily add vertices

pick a root vertex
initialize $T = \{\}$
while (true)
    if T is spanning or
    if there is no vertex connected by an edge to T
        return T
    else, of vertices connected by an edge to T,
        add vertex v with minimum edge cost



Feels like Dijsktra's… use a priority queue!

- extract-min for each vertex: $O(n \log n)$
- update for each edge: $O(m \log n)$

Total: $O((m + n)\log n)$

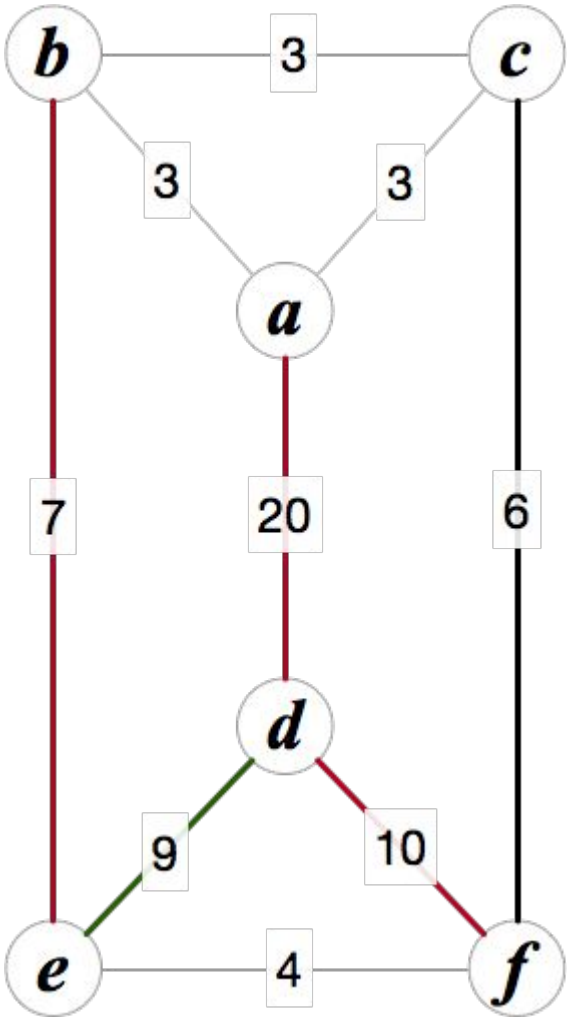
Reverse-Delete algorithm

- Sort edges by decreasing cost
- Initialize $T = E$
- For each edge e
  - Remove e from T if it does not disconnect

- Sorting: $O(m \log m)$
- Check each edge for disconnecting
  - Naive: DFS/BFS $O(m+n)$

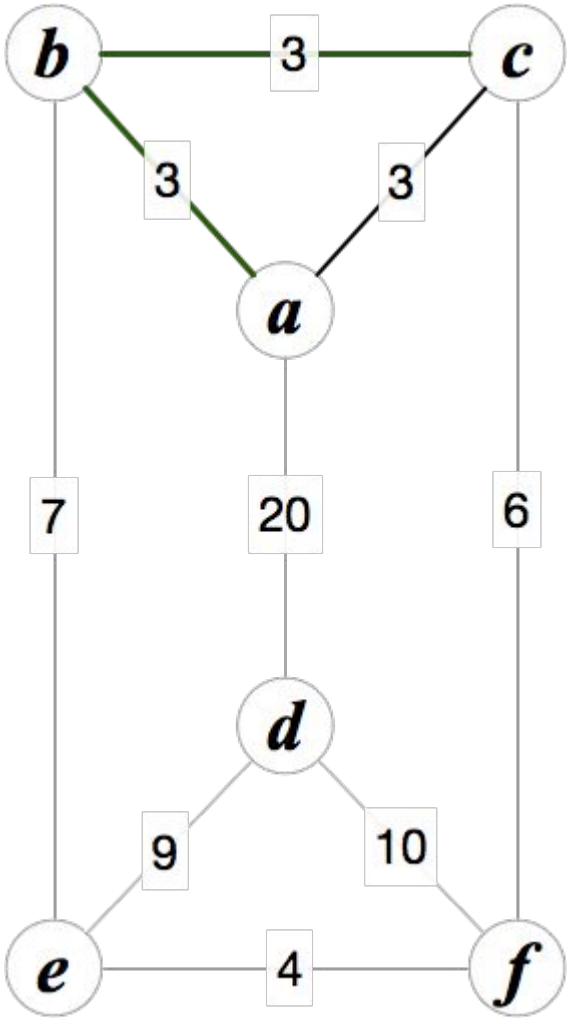
Total: $O(m \log m + m^2 + mn) = O(m^2)$

- [Thorup 2000] $O(m \log n (\log \log n)^3)$



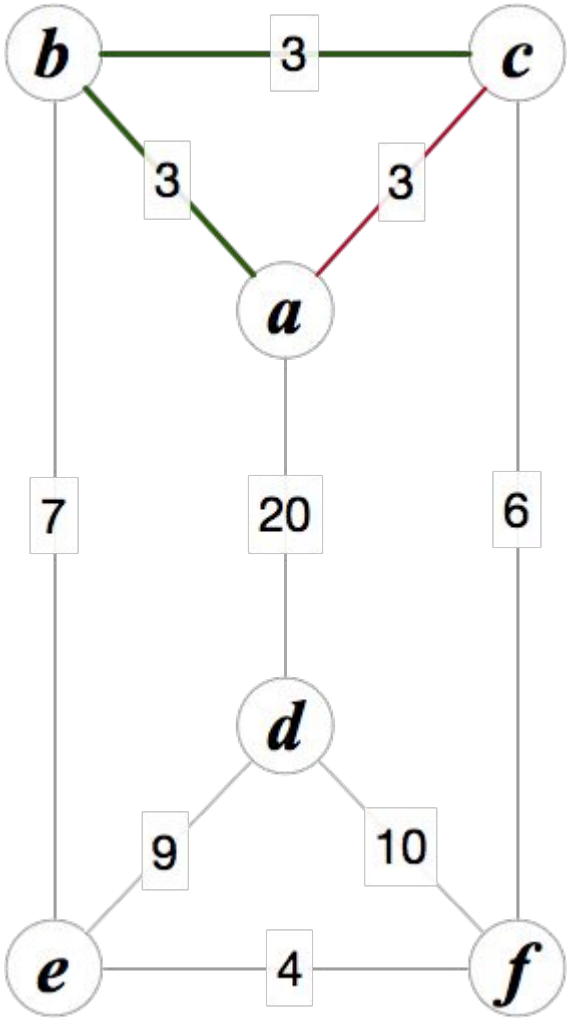
Kruskal's algorithm: greedily add edges

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - Add e to T if it does not create a cycle



Kruskal's algorithm: greedily add edges

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - **Add e to T if it does not create a cycle**



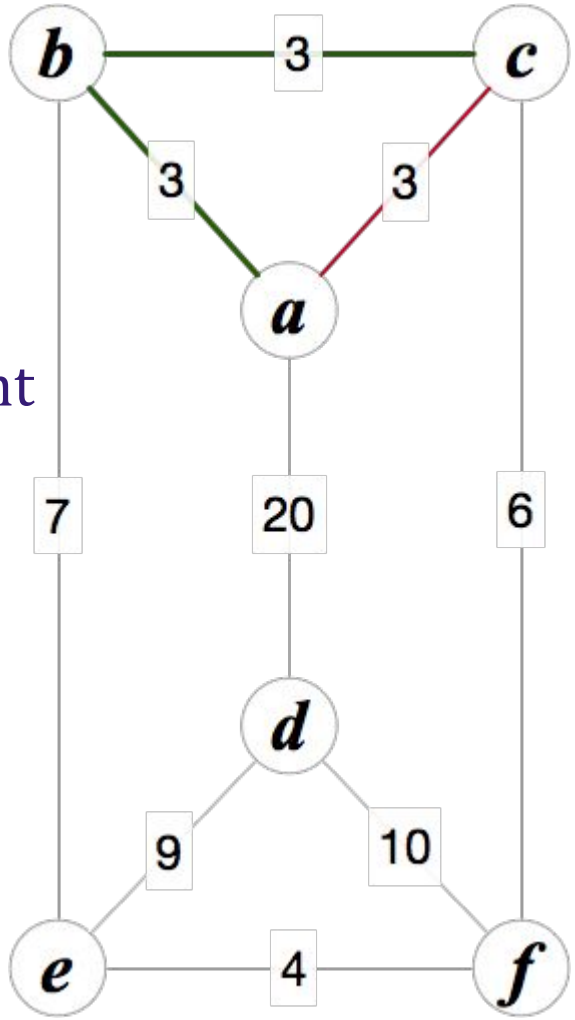
Cycle check: connected components

Edge causes cycle

$\Leftrightarrow$

endpoints in same connected component

- Use BFS or DFS to check if same component



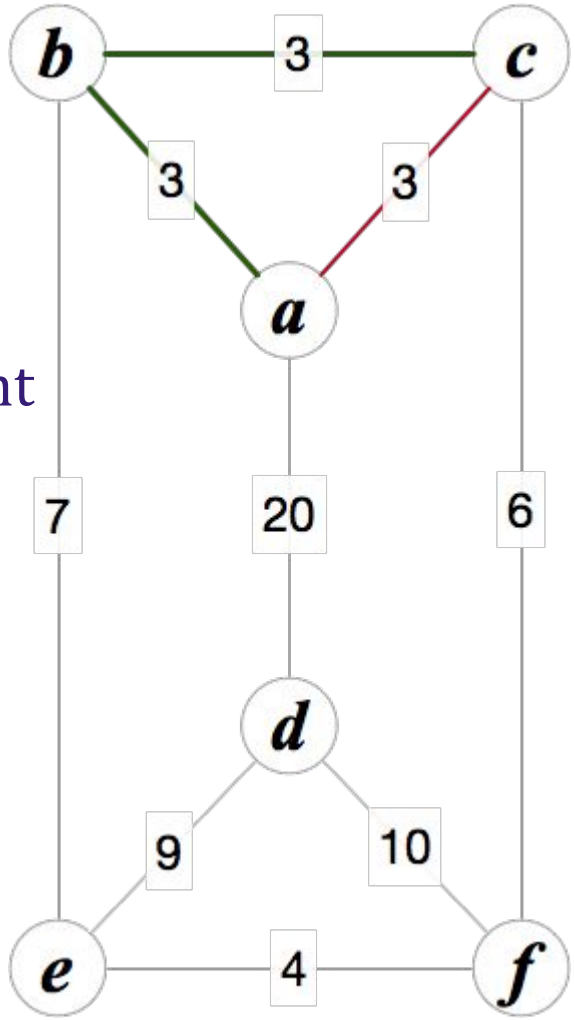
Cycle check: connected components

Edge causes cycle

⇔

endpoints in same connected component

- Use BFS or DFS to check if same component
  - Always O(n)
- Kruskal's: check for each edge
  - Total running time: O($m \log m + mn$)

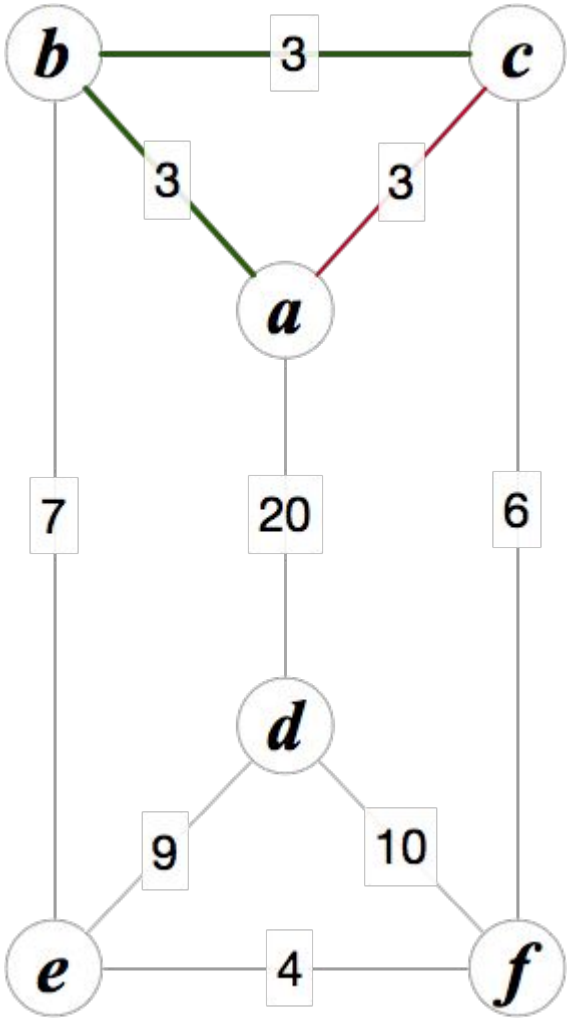


Cycle check with union-find

Edge causes cycle

⇔

endpoints in same connected component



- How to maintain components and find component ID?
- Need: data structures for disjoint sets
  - "Union-Find"
  - Base set of n elements
  - `union`(x, y) operation:
    union two (disjoint) sets containing x and y
  - `find`(x) operation:
    find ID of set containing element x

Union-find for disjoint sets

- Base set of n elements
- `union`(x, y) operation:
  union two (disjoint) sets containing x and y
- `find`(x) operation:
  find ID of set containing element x

Union-find: simple implementation

- Array of length n
- Index i: ID of set containing i
- `union`(x, y): $O(n)$
  - Update elements of x's set (or y's)
- `find`(x): $O(1)$
  - Access A[x]

Kruskal's

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - **Add e to T if it does not create a cycle**

Union-find: simple implementation

- Array of length n
- Index i: ID of set containing i
- `union`(x, y): $O(n)$
  - Update elements of x's set (or y's)
- `find`(x): $O(1)$
  - Access A[x]

Kruskal's

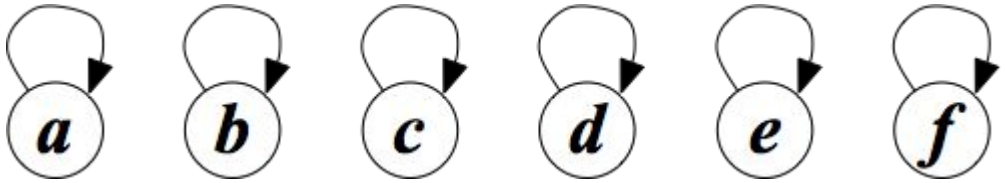
- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - **Add e to T if it does not create a cycle**

- Sorting: $O(m \log m)$
- Cycle check => find
  - $O(1)*m = O(m)$
- Add edge => union
  - $O(n)*n = O(n^2)$

Total: $O(m \log m + n^2)$

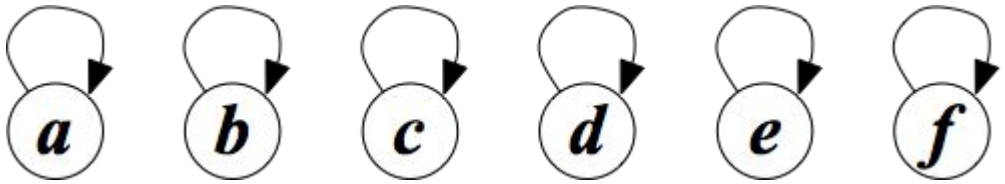
Union-find with pointers

- Each set elects “representative” to be ID
- Elements point to their representative
- Initially, all elements point to themselves



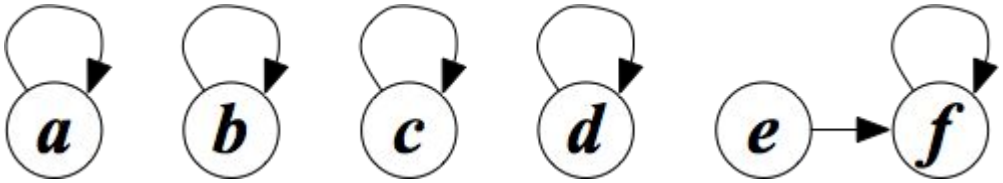
Union operation: O(1) to update pointer

- **`union`**(e, f)



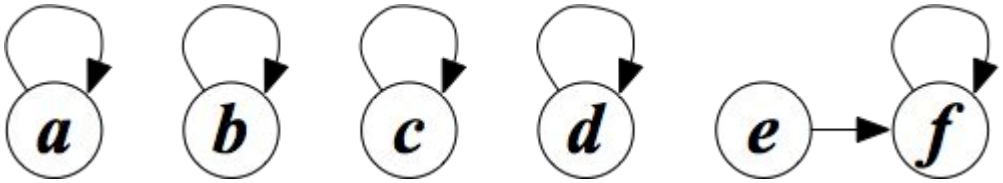
Union operation: O(1) to update pointer

- `union`(*e*, *f*)



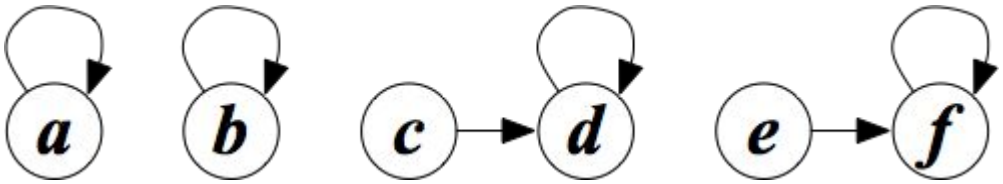
Union operation: O(1) to update pointer

- `union`(*e*, *f*)
- **`union`(*c*, *d*)**



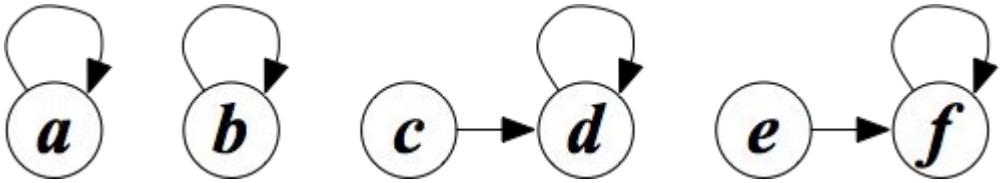
Union operation: O(1) to update pointer

- `union`(e, f)
- `union`(c, d)



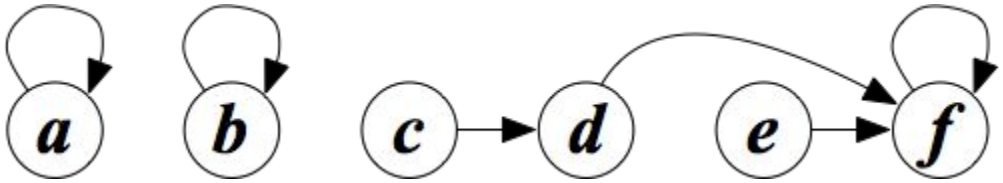
Union operation: O(1) to update pointer

- `union`(e, f)
- `union`(c, d)
- **`union`(d, f)**



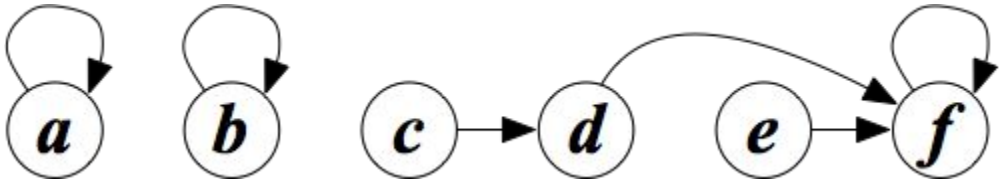
Union operation: O(1) to update pointer

- `union`(e, f)
- `union`(c, d)
- `union`(d, f)



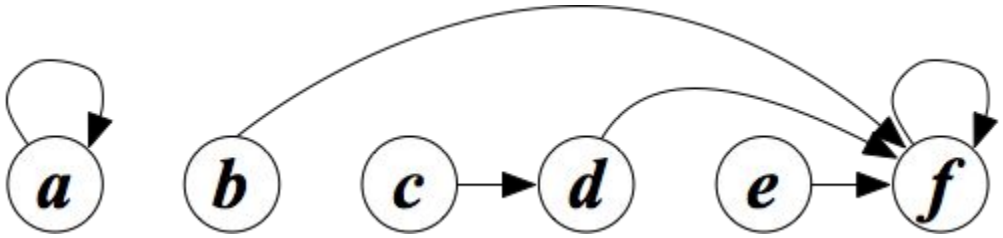
Union operation: O(1) to update pointer

- `union`(e, f)
- `union`(c, d)
- `union`(d, f)
- **`union`(b, f)**



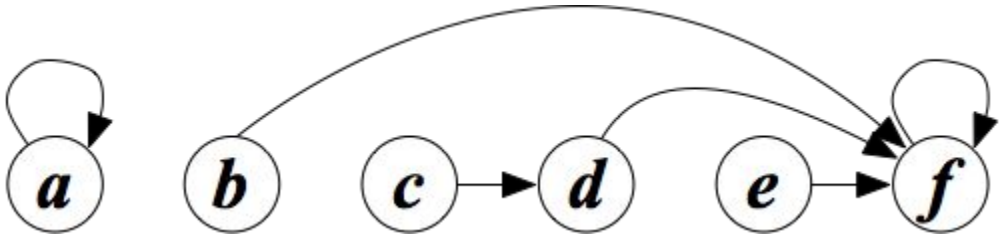
Union operation: O(1) to update pointer

- `union`(e, f)
- `union`(c, d)
- `union`(d, f)
- `union`(b, f)



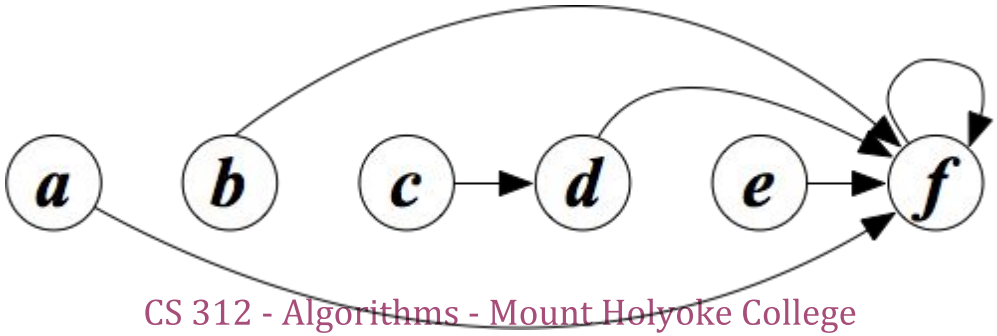
Union operation: O(1) to update pointer

- `union`(e, f)
- `union`(c, d)
- `union`(d, f)
- `union`(b, f)
- **`union`(a, f)**



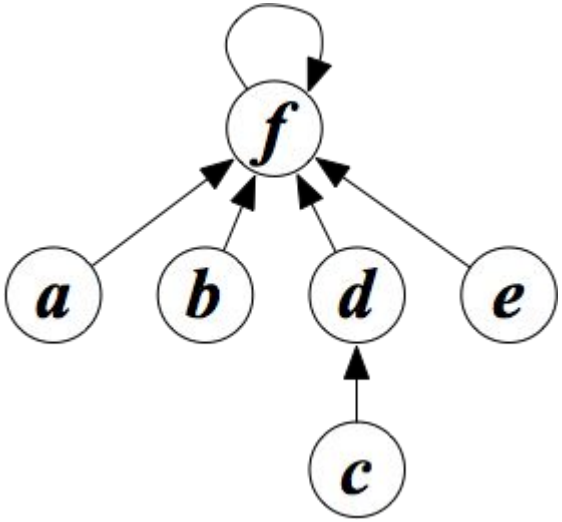
Union operation: O(1) to update pointer

- `union`(e, f)
- `union`(c, d)
- `union`(d, f)
- `union`(b, f)
- `union`(a, f)



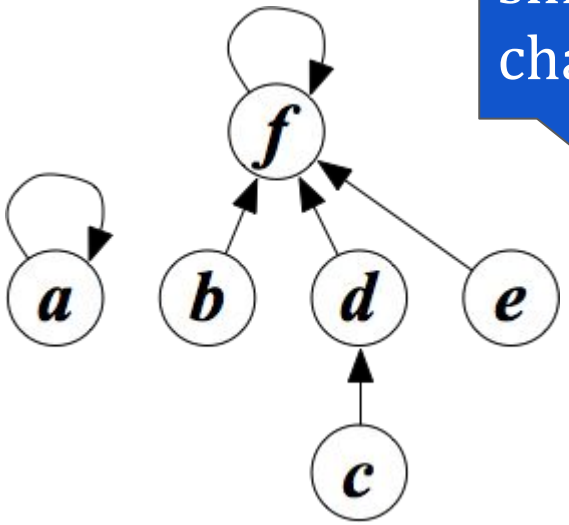
Union operation: O(1) to update pointer

- `union`(e, f)
- `union`(c, d)
- `union`(d, f)
- `union`(b, f)
- `union`(a, f)



Union operation: which pointer?

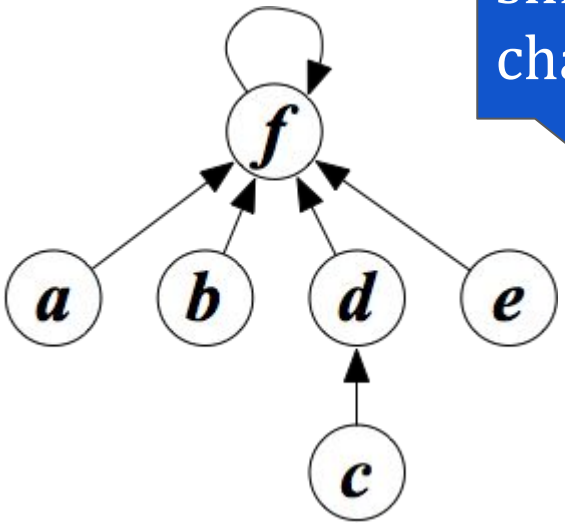
- `union`(e, f)
- `union`(c, d)
- `union`(d, f)
- `union`(b, f)
- **`union`(a, f)**



Heuristic: smaller set changes ID

Union operation: which pointer?

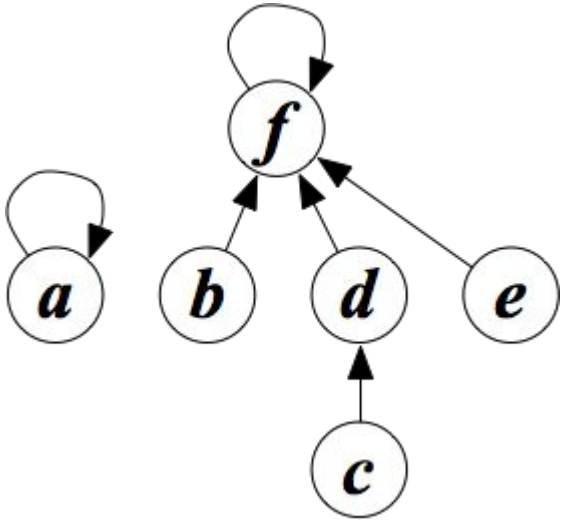
- `union`(e, f)
- `union`(c, d)
- `union`(d, f)
- `union`(b, f)
- `union`(a, f)



Heuristic: smaller set changes ID

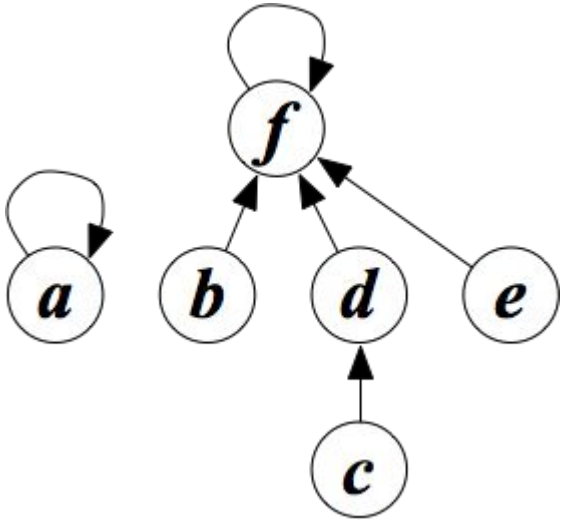
Find operation: how long?

- Depth of tree



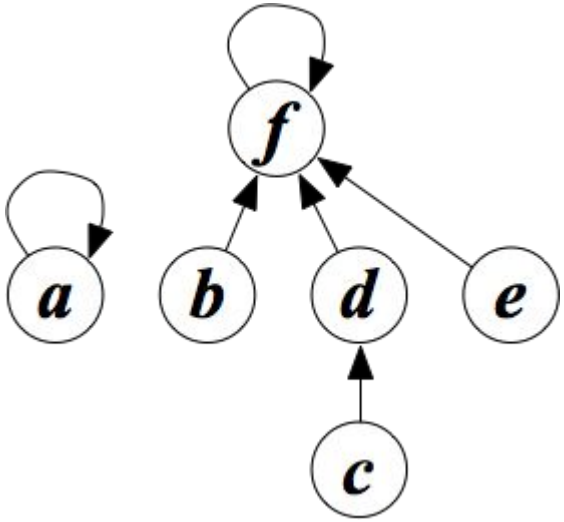
Find operation: how long?

- Depth of tree
- Let d = depth, k = # elements in set
- Claim $d \leq \log_2(k)$
- $\Rightarrow$ `find` is $O(\log n)$



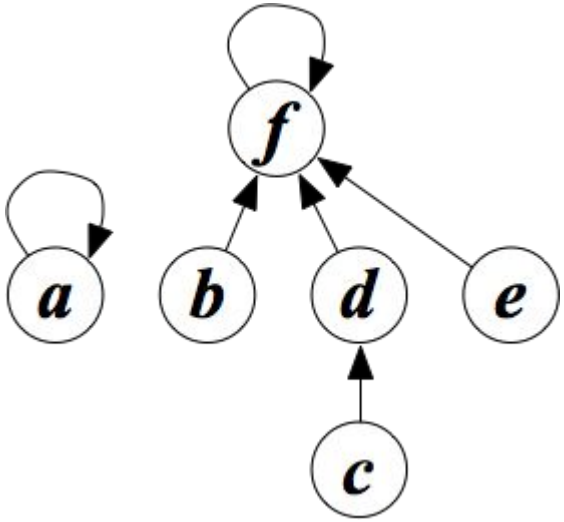
Claim $d \leq \log_2(k)$

- d = depth
- k = # elements in set



Claim $2^d \leq k$

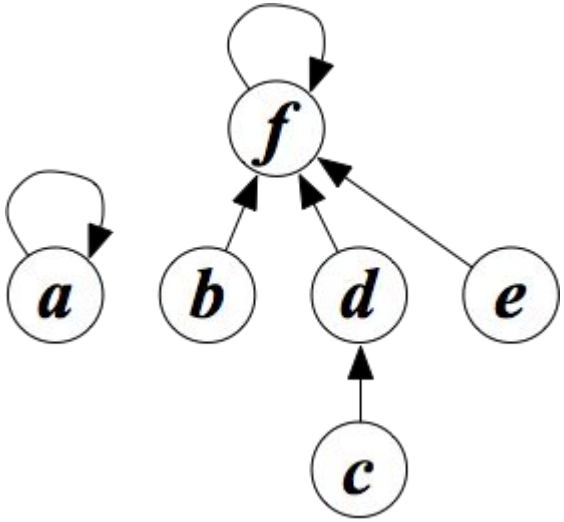
- d = depth
- k = # elements in set



Claim $2^d \leq k$

Proof: by induction on d

- d = depth
- k = # elements in set

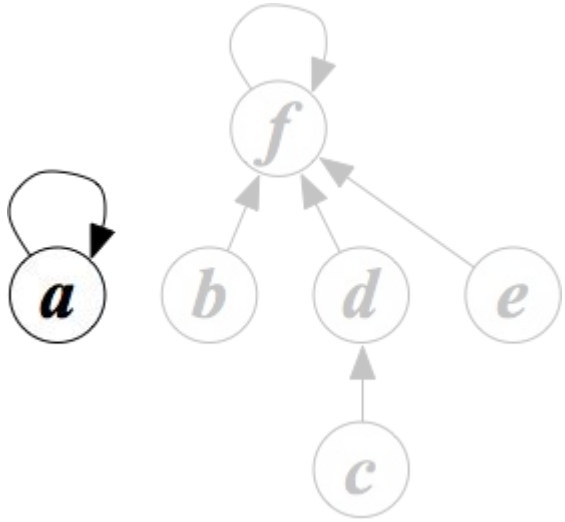


Claim $2^d \leq k$

- d = depth
- k = # elements in set

Proof: by induction on d

Base case: $d = 0$. Then $k = 1$. Done since $2^0 \leq 1$.



Claim $2^d \leq k$

- d = depth
- k = # elements in set

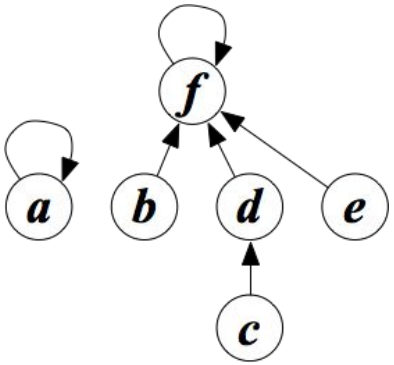
Proof: by induction on d

IH: Assume $2^d \leq k$ for $d \leq c$ for some c.

Inductive case:

$d > c$ after `union`(x, y) of sets of size $k_x \leq k_y$[WLOG]

- Depths d_x and d_y
- Unioned set depth $d = \max\{d_x + 1, d_y\}$



Claim $2^d \leq k$

- d = depth
- k = # elements in set

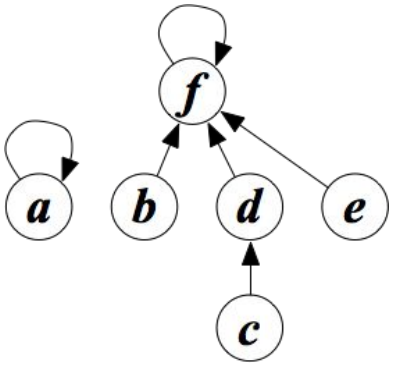
Proof: by induction on d

IH: Assume $2^d \leq k$ for $d \leq c$ for some c.

Inductive case:

$d > c$ after `union`(x, y) of sets of size $k_x \leq k_y$[WLOG]

- Depths d_x and d_y
- Unioned set depth $d = \max\{\boldsymbol{d_x + 1}, d_y\}$



Claim $2^d \leq k$

- d = depth
- k = # elements in set

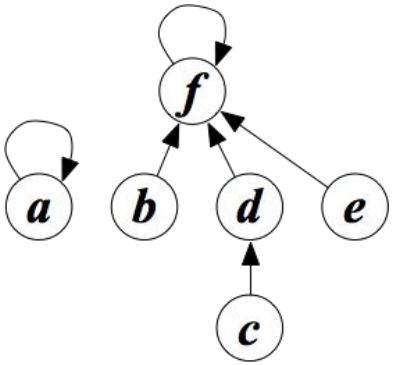
Proof: by induction on d

IH: Assume $2^d \leq k$ for $d \leq c$ for some c.

Inductive case:

$d > c$ after `union`(x, y) of sets of size $k_x \leq k_y$[WLOG]

- Depths d_x and d_y
- Unioned set depth $d = \max\{\boldsymbol{d_x + 1}, d_y\}$
- Unioned set size $k = k_x + k_y \geq 2\, k_x$



Claim $2^d \leq k$

- d = depth
- k = # elements in set

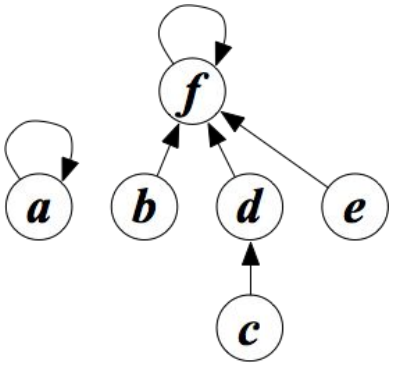
Proof: by induction on d

IH: Assume $2^d \leq k$ for $d \leq c$ for some c.

Inductive case:

$d > c$ after `union`(x, y) of sets of size $k_x \leq k_y$[WLOG]

- Depths d_x and d_y
- Unioned set depth $d = \max\{\boldsymbol{d_x + 1}, d_y\}$
- Unioned set size $k \geq 2\, k_x$



Claim $2^d \leq k$

- d = depth
- k = # elements in set

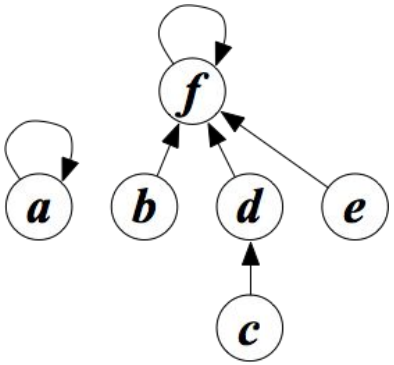
Proof: by induction on d

IH: Assume $2^d \leq k$ for $d \leq c$ for some c.

Inductive case:

$d > c$ after `union`(x, y) of sets of size $k_x \leq k_y$[WLOG]

- Depths d_x and d_y
- Unioned set depth $d = \max\{\boldsymbol{d_x + 1}, d_y\}$
- Unioned set size $k \geq 2\ k_x$
- By induction, $k_x \geq 2^{d_x}$



Claim $2^d \leq k$

- d = depth
- k = # elements in set

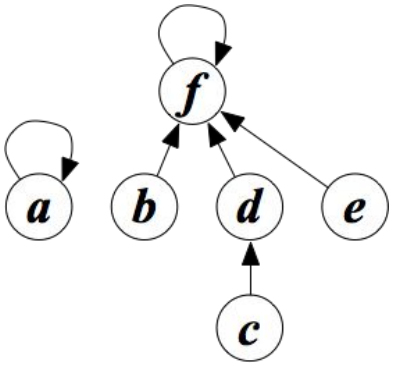
Proof: by induction on d

IH: Assume $2^d \leq k$ for $d \leq c$ for some c.

Inductive case:

$d > c$ after `union`(x, y) of sets of size $k_x \leq k_y$[WLOG]

- Depths d_x and d_y
- Unioned set depth $d = \max\{\boldsymbol{d_x + 1}, d_y\}$
- Unioned set size $\boldsymbol{k \geq 2\, k_x}$
- By induction, $k_x \geq 2^{d_x}$, so $k \geq 2{*}2^{d_x}$



Claim $2^d \leq k$

- d = depth
- k = # elements in set

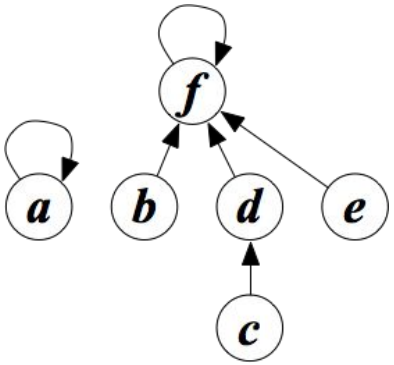
Proof: by induction on d

IH: Assume $2^d \leq k$ for $d \leq c$ for some c.

Inductive case:

$d > c$ after `union`(x, y) of sets of size $k_x \leq k_y$[WLOG]

- Depths d_x and d_y
- Unioned set depth $d = \max\{\boldsymbol{d_x + 1}, d_y\}$
- Unioned set size $k \geq 2\ k_x$
- By induction, $k_x \geq 2^{d_x}$, so $k \geq 2^{d_x + 1}$



Claim $2^d \leq k$

- d = depth
- k = # elements in set

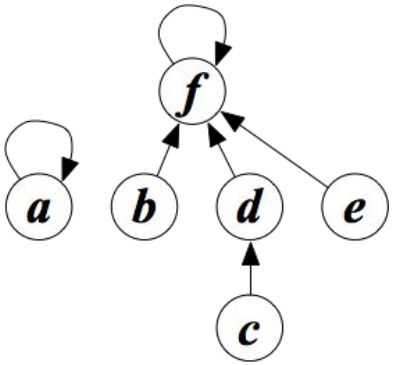
Proof: by induction on d

IH: Assume $2^d \leq k$ for $d \leq c$ for some c.

Inductive case:

$d > c$ after `union`(x, y) of sets of size $k_x \leq k_y$[WLOG]

- Depths d_x and d_y
- Unioned set depth $d = \max\{d_x + 1, d_y\}$



Claim $2^d \leq k$

- d = depth
- k = # elements in set

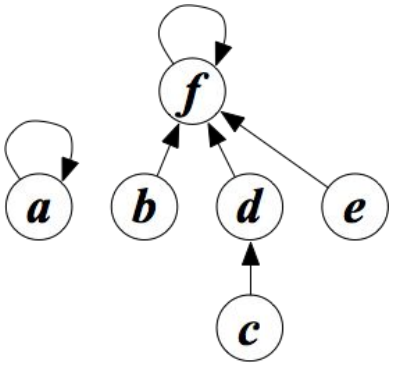
Proof: by induction on d

IH: Assume $2^d \leq k$ for $d \leq c$ for some c.

Inductive case:

$d > c$ after `union`(x, y) of sets of size $k_x \leq k_y$[WLOG]

- Depths d_x and d_y
- Unioned set depth $d = \max\{d_x + 1, \boldsymbol{d_y}\}$



Claim $2^d \leq k$

- d = depth
- k = # elements in set

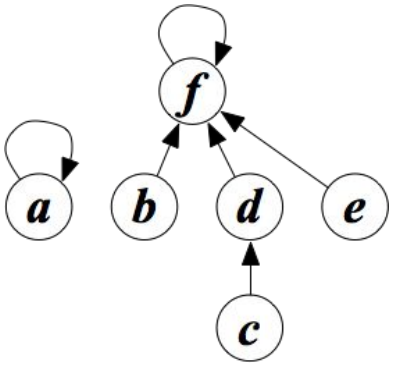
Proof: by induction on d

IH: Assume $2^d \leq k$ for $d \leq c$ for some c.

Inductive case:

$d > c$ after `union`(x, y) of sets of size $k_x \leq k_y$[WLOG]

- Depths d_x and d_y
- Unioned set depth $d = \max\{d_x + 1, \boldsymbol{d_y}\}$
- Unioned set size $k = k_x + k_y \geq k_y$



Claim $2^d \leq k$

- d = depth
- k = # elements in set

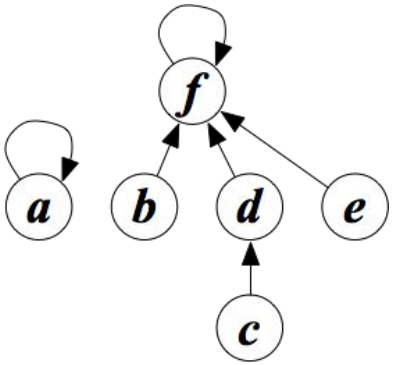
Proof: by induction on d

IH: Assume $2^d \leq k$ for $d \leq c$ for some c.

Inductive case:

$d > c$ after `union`(x, y) of sets of size $k_x \leq k_y$[WLOG]

- Depths d_x and d_y
- Unioned set depth $d = \max\{d_x + 1, \boldsymbol{d_y}\}$
- Unioned set size $k \geq k_y$



Claim $2^d \leq k$

- d = depth
- k = # elements in set

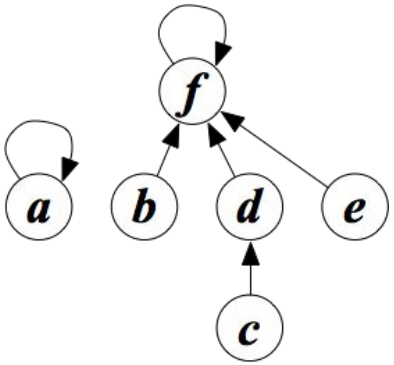
Proof: by induction on d

IH: Assume $2^d \leq k$ for $d \leq c$ for some c.

Inductive case:

$d > c$ after `union`(x, y) of sets of size $k_x \leq k_y$[WLOG]

- Depths d_x and d_y
- Unioned set depth $d = \max\{d_x + 1, \boldsymbol{d_y}\}$
- Unioned set size $k \geq k_y$
- By induction, $k_y \geq 2^{dy}$, so $k \geq 2^{dy}$



Claim $2^d \leq k$

- d = depth
- k = # elements in set

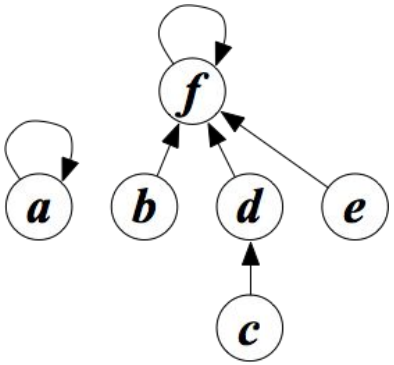
Proof: by induction on d

IH: Assume $2^d \leq k$ for $d \leq c$ for some c.

Inductive case:

$d > c$ after `union`(x, y) of sets of size $k_x \leq k_y$[WLOG]

- Depths d_x and d_y
- Unioned set depth $d = \max\{d_x + 1, d_y\}$
- Unioned set size $k \geq k_y$
- By induction, $k_y \geq 2^{dy}$, so $k \geq 2^{dy}$



Claim $2^d \leq k$

- d = depth
- k = # elements in set

Proof: by induction on d

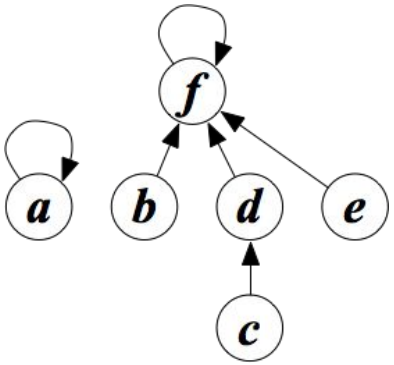
IH: Assume $2^d \leq k$ for $d \leq c$ for some c.

Inductive case:

$d > c$ after `union`(x, y) of sets of size $k_x \leq k_y$[WLOG]

- Depths d_x and d_y
- Unioned set depth $d = \max\{d_x + 1, d_y\}$
- Thus, $k \geq 2^d$

Q.E.D.



Union-find: pointers

- `union`(x, y): $O(1)$
  - Update pointer of smaller set
- `find`(x): $O(\log n)$
  - Depth of tree

Kruskal's

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - **Add e to T if it does not create a cycle**

- Sorting: $O(m \log n)$
- Cycle check => find
  - $O(\log n)*m = O(m \log n)$
- Add edge => union
  - $O(1)*n = O(n)$

Total: $O(m \log n)$

Union-find: pointers

- `union`(x, y): $O(1)$
  - Update pointer of smaller set
- `find`(x): $O(\log n)$
  - Depth of tree

Kruskal's

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - **Add e to T if it does not create a cycle**

- Sorting: $O(m \log n)$
- Cycle check => find
  - $O(\log n)*m = O(m \log n)$
- Add edge => union
  - $O(1)*n = O(n)$

Total: $O(m \log n)$

Union-find: union by rank with path compression

- $O(q\ \alpha(n))$
  for q disjoint-set operations
  (`union` or `find`)
  on n elements
- $\alpha(n)$ is inverse of
  "Ackermann's function"
  which grows "astronomically"
- $\alpha(n) \leq 4$ for values in practice

Kruskal's

- Sort edges by cost
- Initialize $T = \{\}$
- For each edge e
  - **Add e to T if it does not create a cycle**

- Sorting: $O(m \log n)$
- Cycle check => find
- Add edge => union
- $m+n$ disjoint-set operations

Total: $O(m \log n + (m+n)\alpha(n))$
=> essentially $O(m \log n)$

Which algorithm do you think will produce a spanning tree of minimum cost?

Kruskal's: add next edge sorted by cost, unless it creates a cycle

Prim's: start at a root, add vertex with minimum edge connection cost

Reverse-delete: start with all edges, remove next most costly edge, unless it disconnects the graph

None of the above

All of the above

I don't know

The running time for performing BFS or DFS to check if two vertices are in the same component during Kruskal's algorithm is $O(n)$:

- for every edge check
- never
- for only some edge checks
- I don't know

Let $f = m \log(n^2)$ and $g = m \log n$. True or false?

$$f = O(g)$$

True

False

I don't know.

Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**