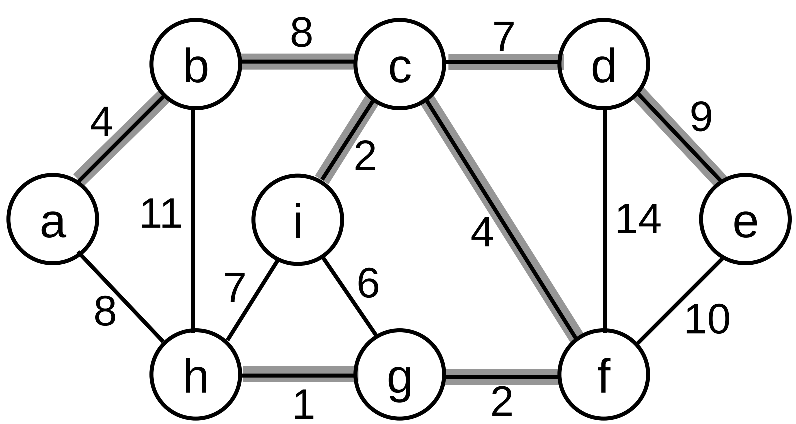
Minimum Spanning Trees (MST)

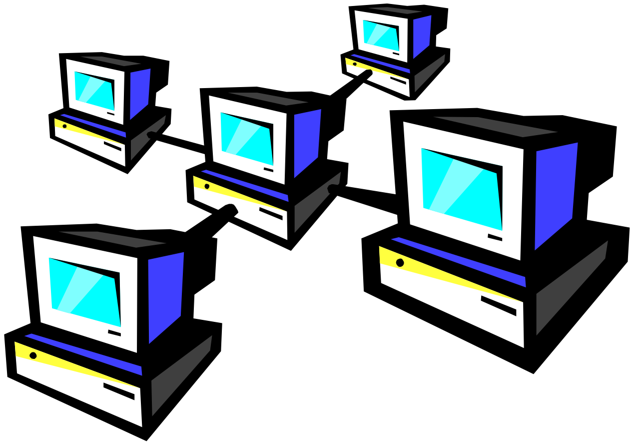
Minimum Spanning Trees

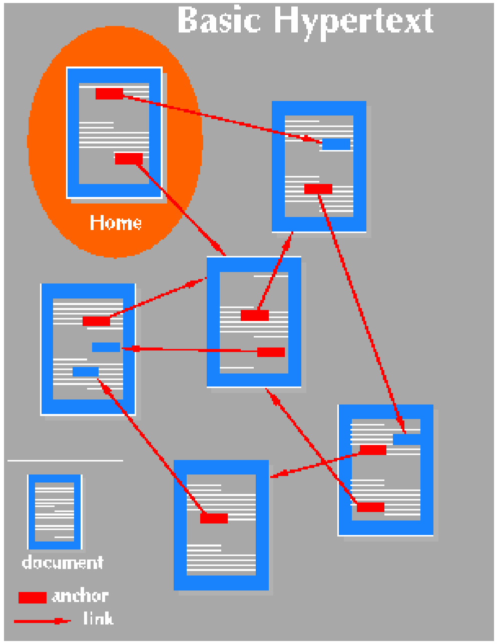
- Spanning Tree
  - A tree (i.e., connected, acyclic graph) which contains all the vertices of the graph
- Minimum Spanning Tree
  - Spanning tree with the **minimum sum of weights**



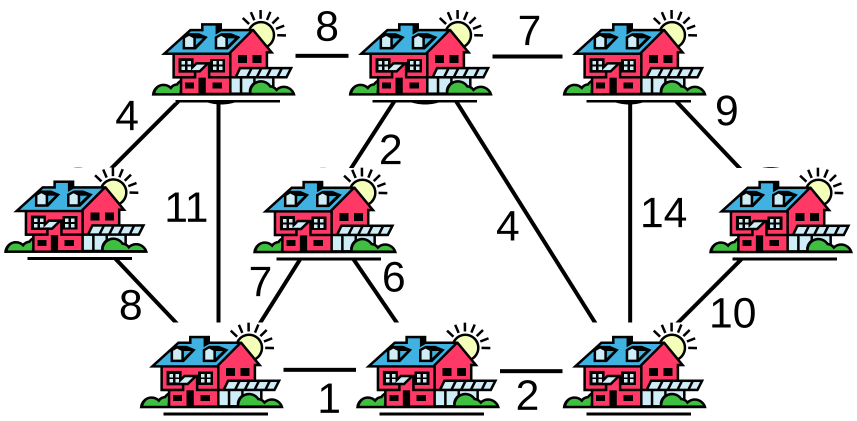
Applications of MST

- Find the least expensive way to connect a set of cities, terminals, computers, etc.





Example



Problem

- A town has a set of houses and a set of roads
- A road connects 2 and only 2 houses
- A road connecting houses u and v has a repair cost w(u, v)

Goal: Repair enough (and no more) roads such that:

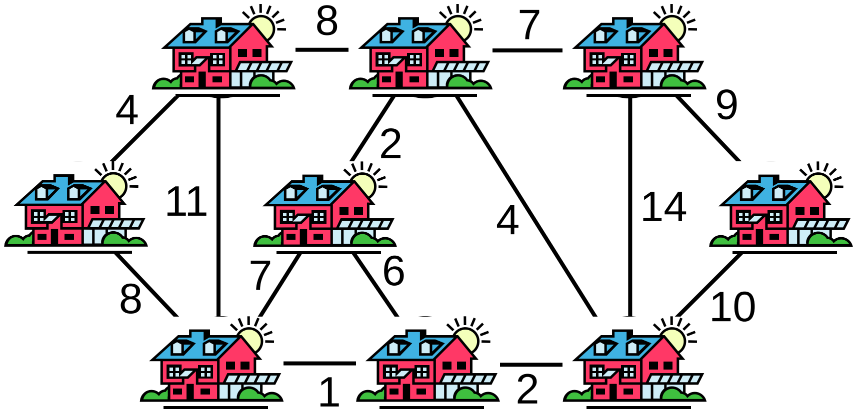
1. Everyone stays connected
   i.e., can reach every house from all other houses using good road
2. Total repair cost is minimum

Minimum Spanning Trees

- A connected, undirected graph:
  - Vertices = houses, Edges = roads
- A **weight** $w(u, v)$ on each edge $(u, v) \in E$

Find $T \subseteq E$ such that:

1. T connects all vertices
2. $w(T) = \Sigma_{(u,v) \in T}\, w(u, v)$ is minimized



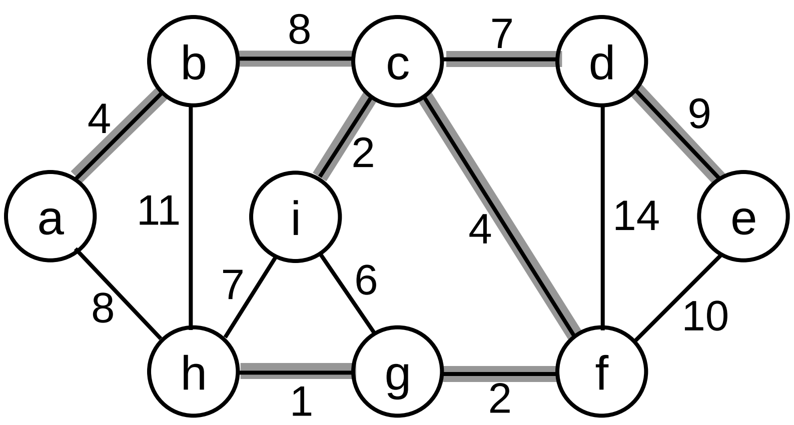
Properties of Minimum Spanning Trees

- Minimum spanning tree might **not** be unique

- MST has no cycles – see why:
  - If there exist any cycle then by definition it is not a tree
  - We can take out an edge of a cycle, and still have the vertices connected while reducing the cost
- # of edges in a MST:|V| - 1

Growing a MST – Generic Approach

- Grow a set A of edges (initially empty)
- Incrementally add edges to A such that they would belong to a MST

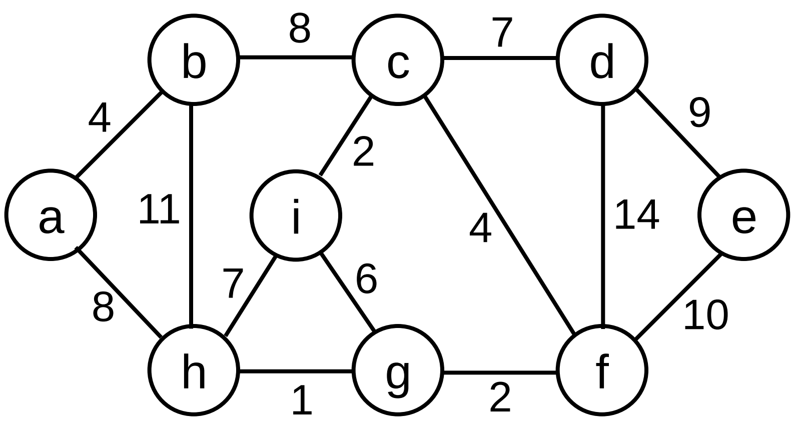


Idea: add only "safe" edges

- An edge (u, v) is **safe** for A if and only if $A \cup \{(u, v)\}$ is also a subset of **some** MST

Generic MST algorithm

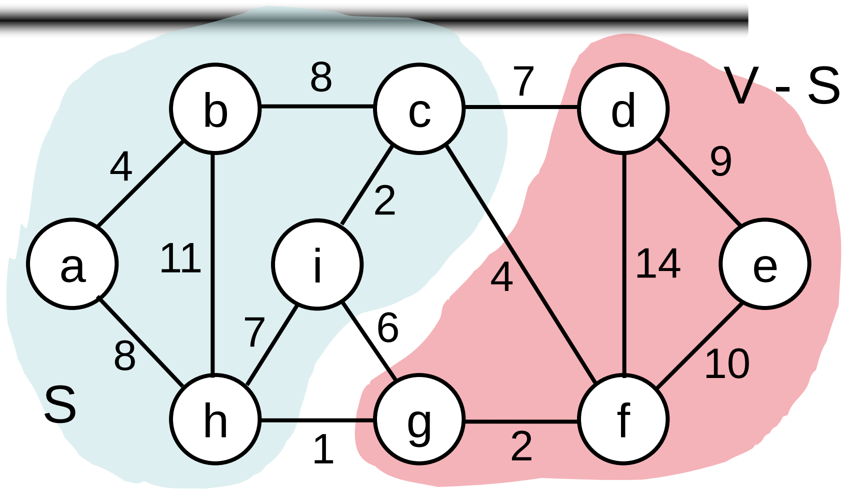
1. $A \leftarrow \varnothing$
2. **while** A is not a spanning tree
3. **do** find an edge **(u, v)** that is safe for A
4. $A \leftarrow A \cup \{(u, v)\}$
5. **return** A



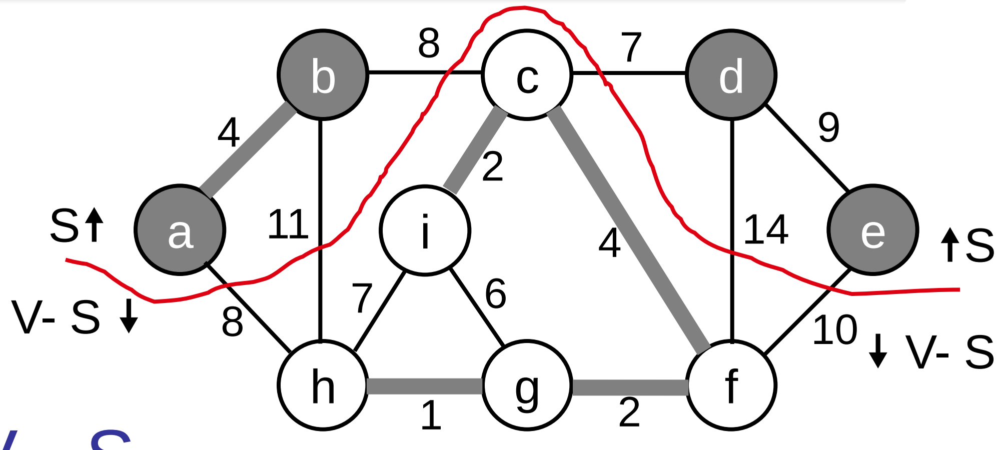
- How do we find safe edges?

Finding Safe Edges

- Let's look at edge (h, g)
  - Is it safe for A initially?
- Later on:
  - Let $S \subset V$ be any set of vertices that includes h but not g (so that g is in V - S)
  - In any MST, there has to be one edge (at least) that connects S with V - S
  - Why not choose the edge with **minimum weight** (h,g)?



Definitions



- A **cut** (S, V - S)

is a partition of vertices

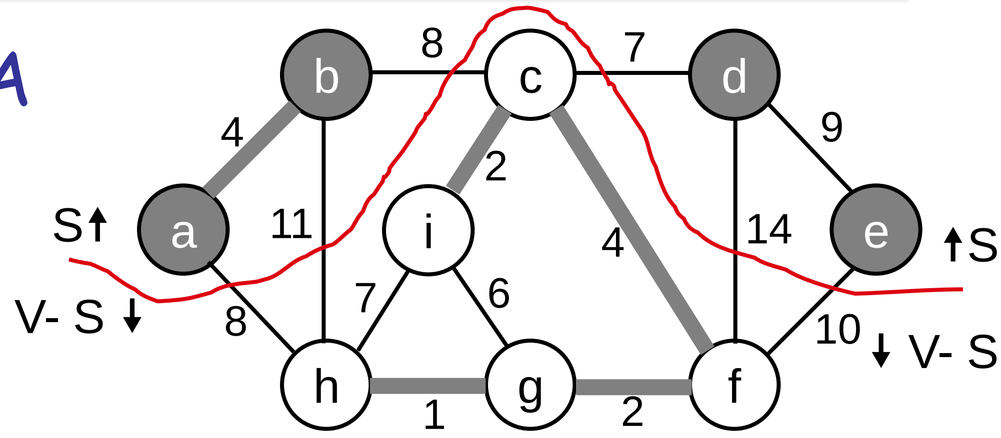
into disjoint sets S and V - S

- An edge **crosses** the cut

(S, V - S) if one endpoint is in S

and the other in V – S

Definitions (cont'd)



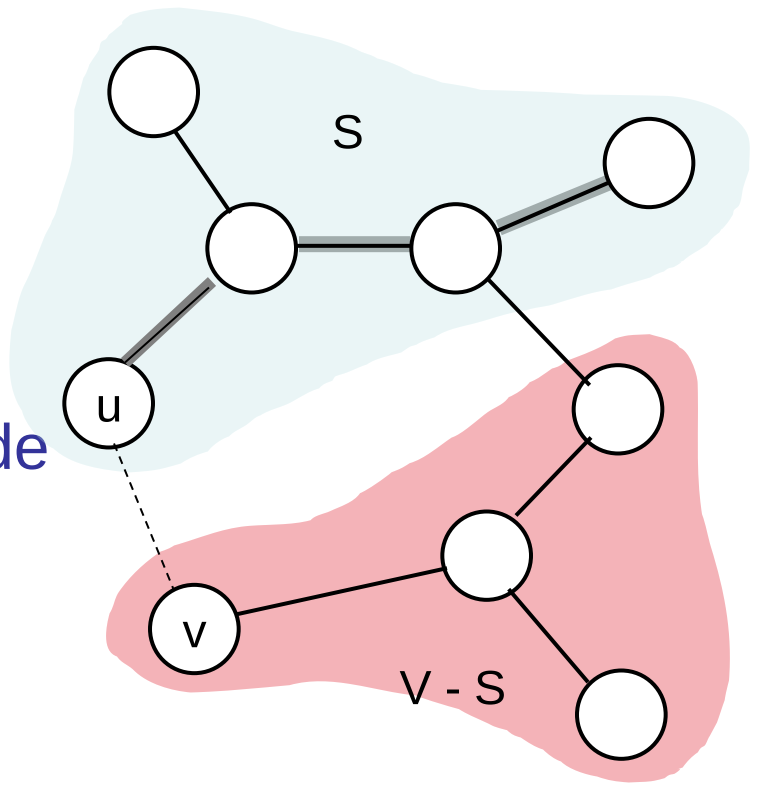
- A cut **respects** a set *A* of edges ⇔ no edge in *A* crosses the cut
- An edge is a **light edge** crossing a cut ⇔ its weight is minimum over all edges crossing the cut
  - Note that for a given cut, there can be > 1 light edges crossing it

Theorem

- Let *A* be a subset of some MST (i.e., T), (S, V - S) be a **cut** that respects A, and (u, v) be a **light edge** crossing (S, V-S). Then (u, v) is safe for *A* .

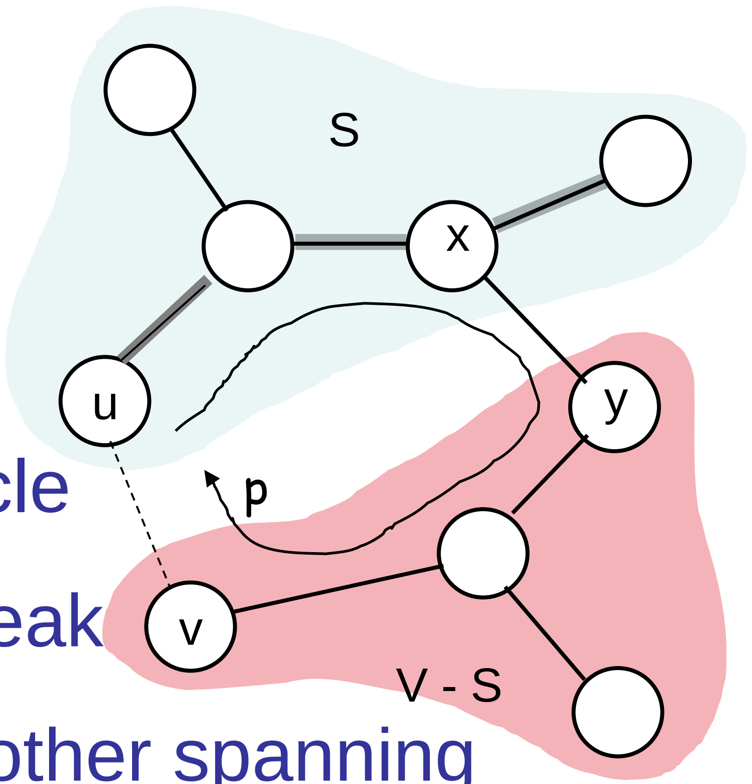
Proof:

- Let A be a subset of T (an MST)
  - edges in A are shaded
- Case1: If T includes (u,v), then it would be safe for A
- Case2: Suppose T does not include the edge (u, v)
- **Idea**: construct another MST T' that includes A ∪ {(u, v)}



Theorem - Proof

- T contains a unique path p between u and v
- Path p must cross the cut (S, V - S) at least once: let (x, y) be that edge

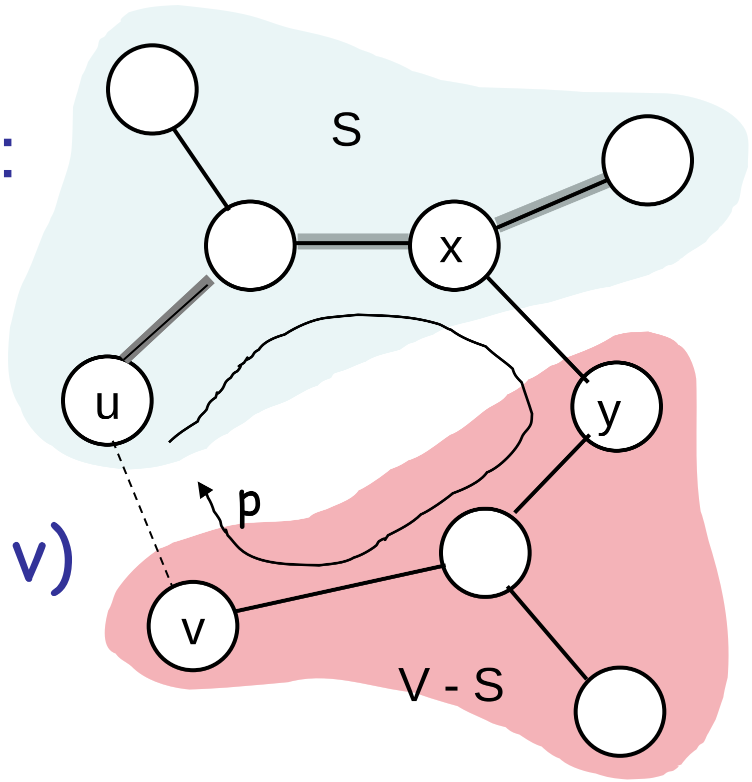


- So adding (u,v) creates a cycle
- If we remove (x,y) we can break the cycle and we can get another spanning tree of minimum cost $T' = T - \{(x, y)\} \cup \{(u, v)\}$

Theorem – Proof (cont.)

$T' = T - \{(x, y)\} \cup \{(u, v)\}$

Have to show that T' is an MST:



- (u, v) is a light edge

  $\Rightarrow w(u, v) \leq w(x, y)$

- $w(T') = w(T) - w(x, y) + w(u, v)$

  $\leq w(T)$

- Since T is a spanning tree

  $w(T) \leq w(T') \Rightarrow$ T' must be an MST as well

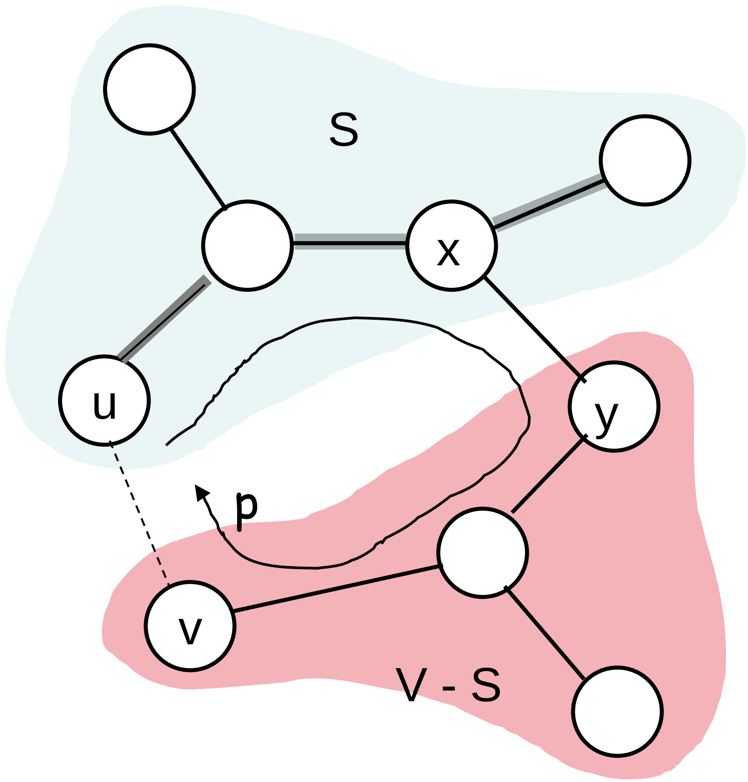
Theorem – Proof (cont.)

Need to show that (u, v) is safe for A:

i.e., (u, v) can be a part of an MST

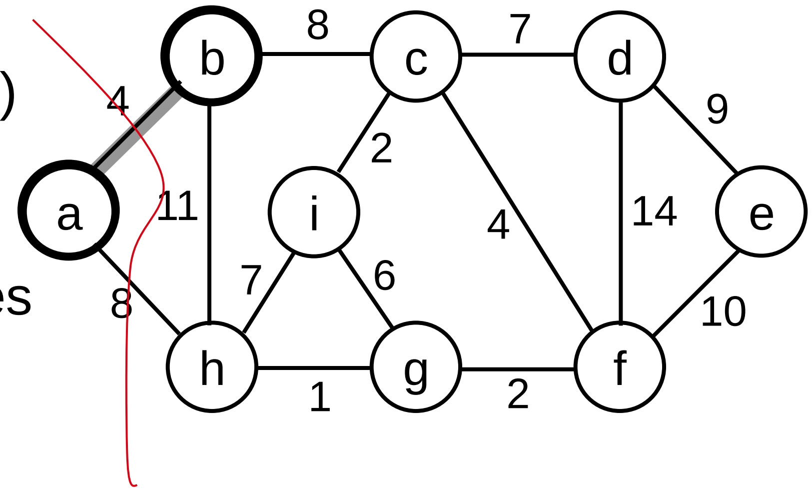
- $A \cup \{(u, v)\} \subseteq T'$
- Since T' is an MST

$\Rightarrow$ (u, v) is safe for A



Prim's Algorithm

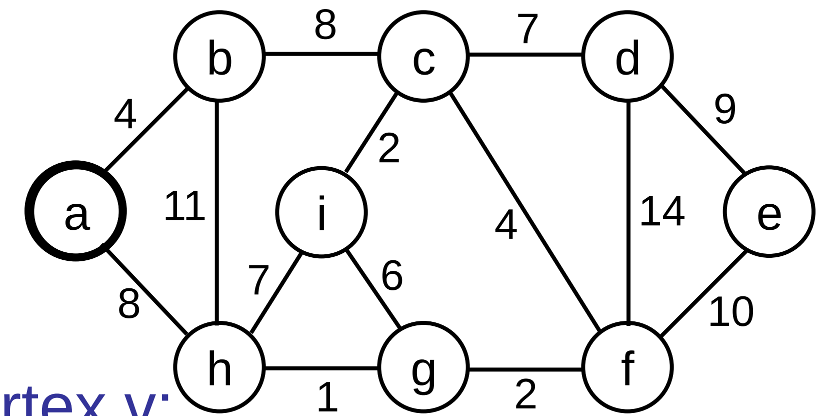
- The edges in set A always <u>form a single tree</u>
- Starts from an arbitrary "root": $V_A = \{a\}$
- At each step:
  - Find a light edge crossing $(V_A, V - V_A)$
  - Add this edge to A
  - Repeat until the tree spans all vertices



How to Find Light Edges Quickly?

Use a priority queue Q:

- Contains vertices not yet included in the tree, i.e., ($V - V_A$)
  - $V_A = \{a\}$, Q = {b, c, d, e, f, g, h, i}

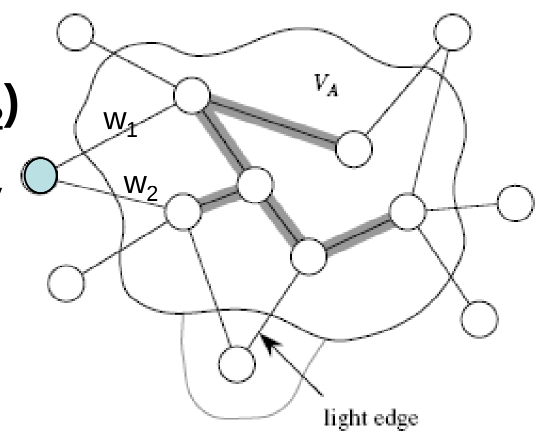


- We associate a key with each vertex v:

key[v] = minimum weight of any edge (u, v) connecting v to V_A

Key[v]=min(w_1,w_2)

v

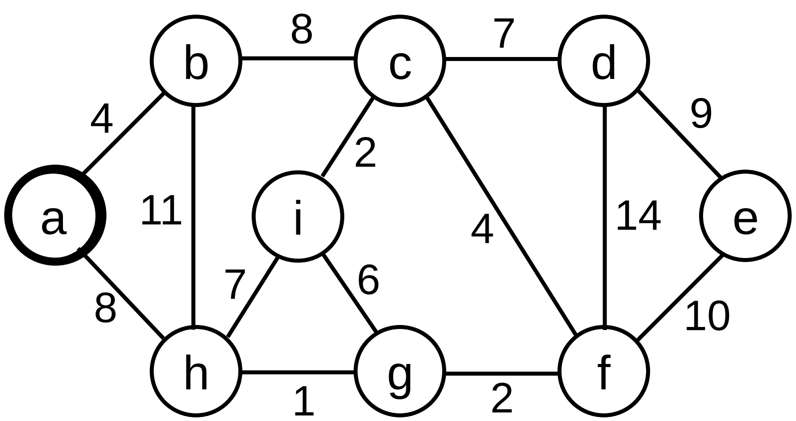


How to Find Light Edges Quickly? (cont.)

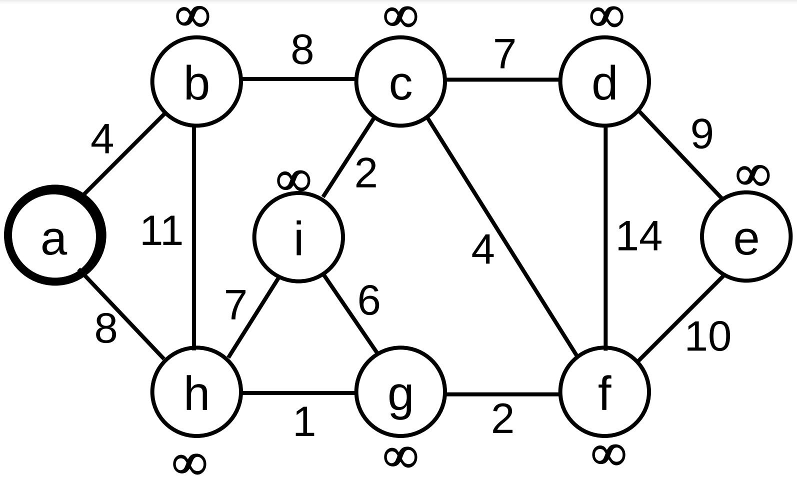
- After adding a new node to V_A we update the weights of all the nodes adjacent to it

  e.g., after adding **a** to the tree, k[b]=4 and k[h]=8

- Key of v is ∞ if v is not adjacent to any vertices in V_A



Example

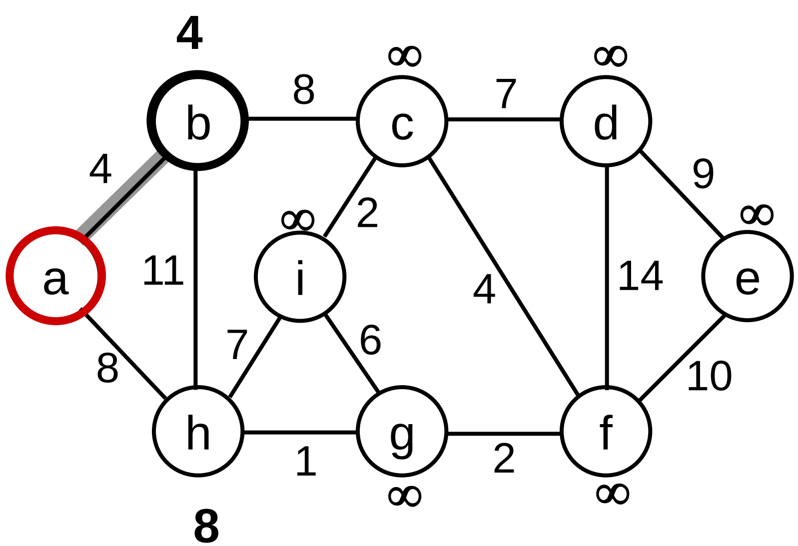


$0\ \infty\ \infty\ \ \infty\ \infty\ \infty\ \infty\ \infty\ \infty$

Q = {a, b, c, d, e, f, g, h, i}

$V_A = \varnothing$

Extract-MIN(Q) $\Rightarrow$ a



key [b] = 4 π [b] = a

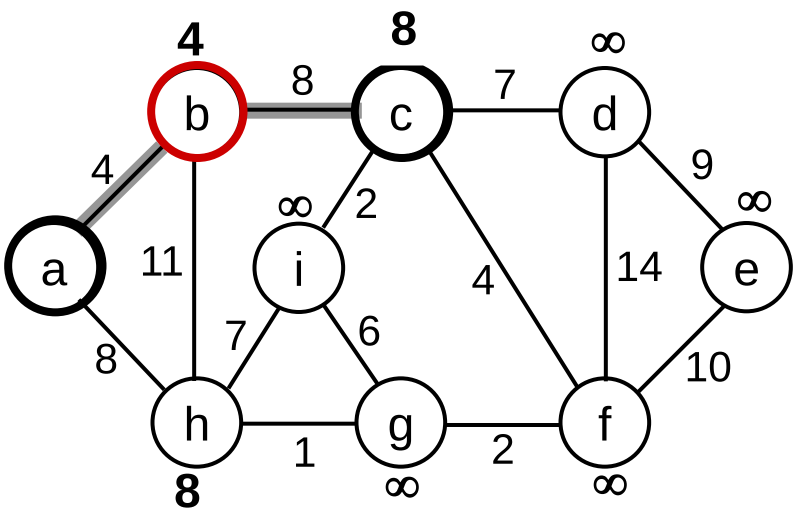
key [h] = 8 π [h] = a

4 $\infty\ \infty\ \infty\ \infty\ \infty$ **8** ∞

Q = {b, c, d, e, f, g, h, i} V_A = {a}

Extract-MIN(Q) $\Rightarrow$ b

Example



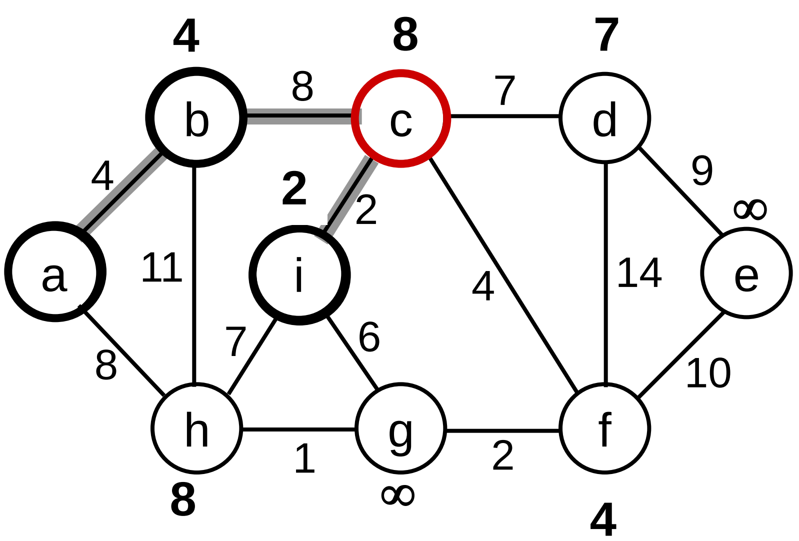
key [c] = 8 π [c] = b

key [h] = 8 π [h] = a - unchanged

8 ∞ ∞ ∞ ∞ 8 ∞

Q = {c, d, e, f, g, h, i} V_A = {a, b}

Extract-MIN(Q) ⇒ c



key [d] = 7 π [d] = c

key [f] = 4 π [f] = c

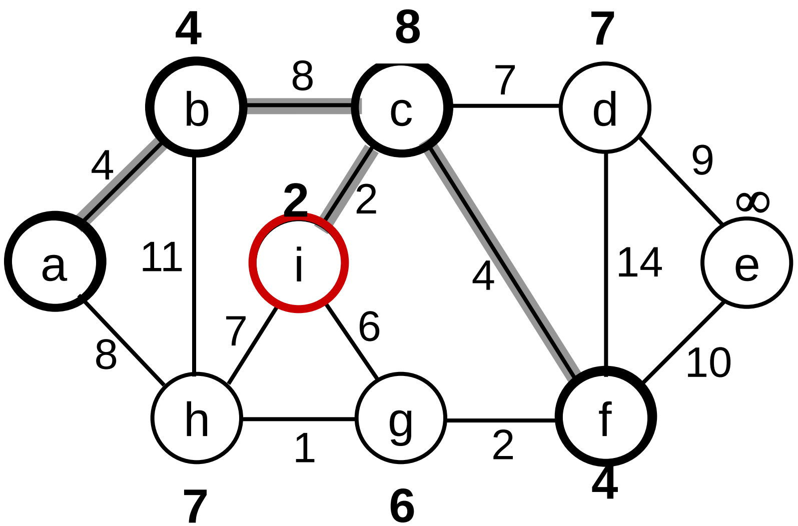
key [i] = 2 π [i] = c

7 ∞ 4 ∞ 8 2

Q = {d, e, f, g, h, i} V_A = {a, b, c}

Extract-MIN(Q) ⇒ i

Example



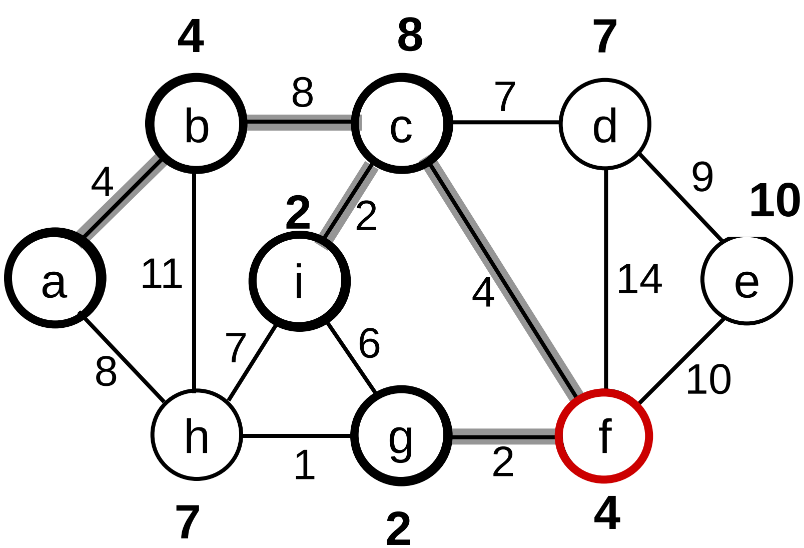
key [h] = 7 π [h] = i

key [g] = 6 π [g] = i

7 ∞ 4 6 8

Q = {d, e, f, g, h} V_A = {a, b, c, i}

Extract-MIN(Q) $\Rightarrow$ f



key [g] = 2 π [g] = f

key [d] = 7 π [d] = c unchanged

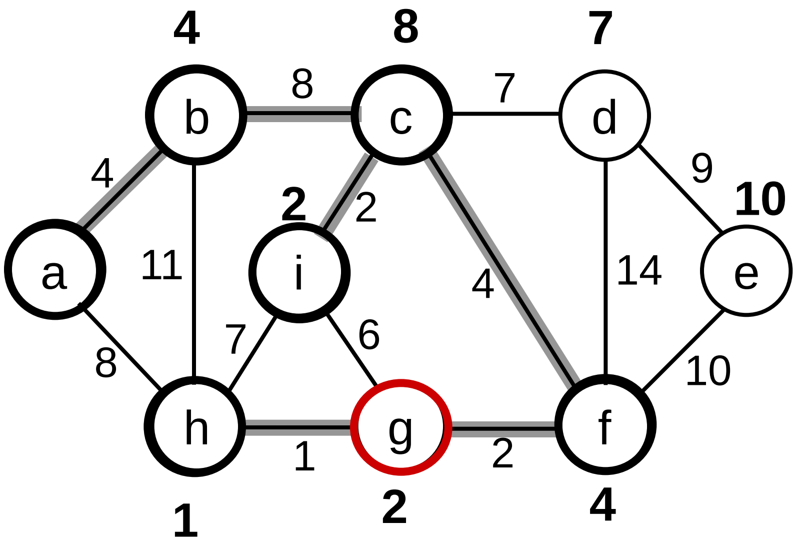
key [e] = 10 π [e] = f

7 10 2 8

Q = {d, e, g, h} V_A = {a, b, c, i, f}

Extract-MIN(Q) $\Rightarrow$ g

Example

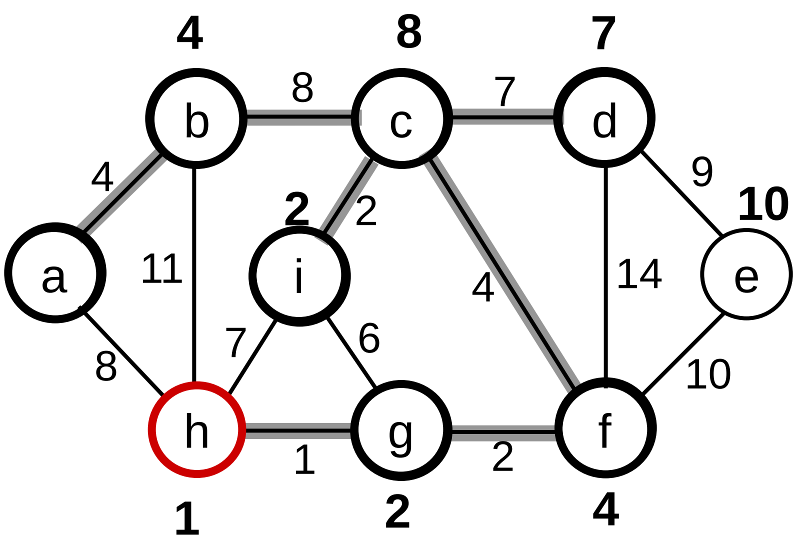


key [h] = 1   π [h] = g

7 10 1

Q = {d, e, h} V_A = {a, b, c, i, f, g}

Extract-MIN(Q) ⇒ h

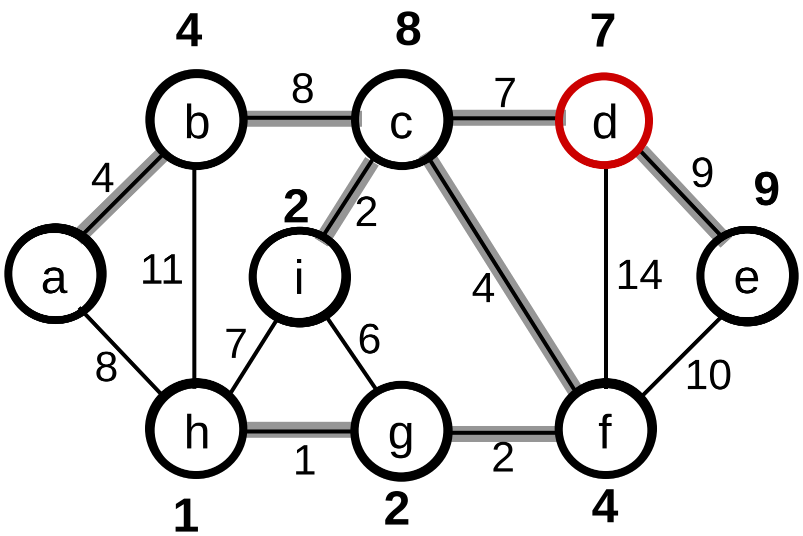


7 10

Q = {d, e} V_A = {a, b, c, i, f, g, h}

Extract-MIN(Q) ⇒ d

Example



key [e] = 9 π [e] = f

9

Q = {e} V_A = {a, b, c, i, f, g, h, d}

Extract-MIN(Q) $\Rightarrow$ e

Q = $\varnothing$ V_A = {a, b, c, i, f, g, h, d, e}

PRIM(V, E, w, r)

Total time: $O(V\lg V + E\lg V) = O(E\lg V)$

```
Q ← ∅
for each u ∈ V
    do key[u] ← ∞
       π[u] ← NIL
       INSERT(Q, u)
DECREASE-KEY(Q, r, 0)        ▶ key[r] ← 0
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       for each v ∈ Adj[u]
           do if v ∈ Q and w(u, v) < key[v]
                 then π[v] ← u
                      DECREASE-KEY(Q, v, w(u, v))
```

$O(V)$ if Q is implemented as a min-heap

$O(\lg V)$

Executed |V| times

Takes $O(\lg V)$

Min-heap operations: $O(V\lg V)$

Executed O(E) times total

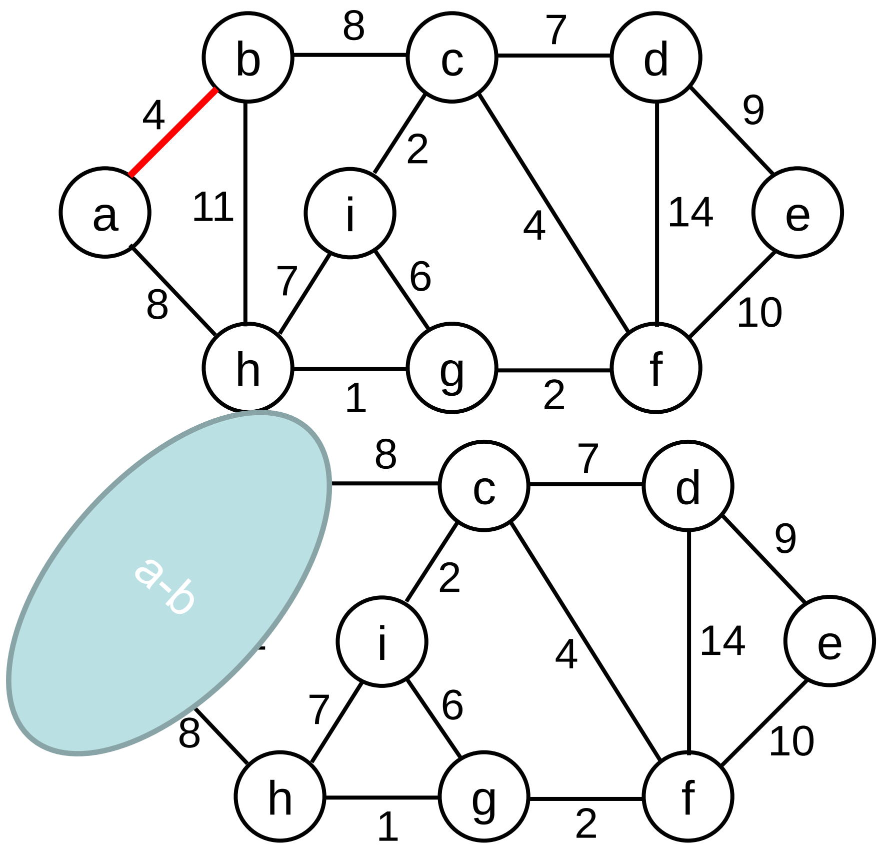
Constant

Takes $O(\lg V)$

$O(E\lg V)$

Prim’s Algorithm

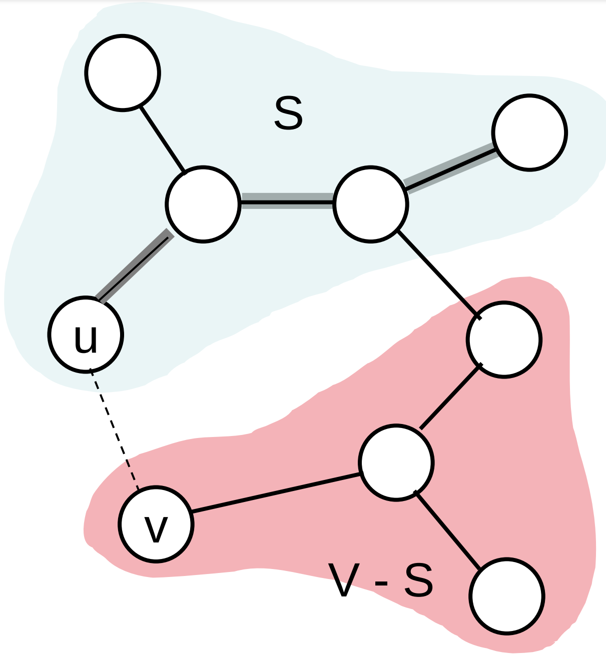
- Prim’s algorithm is a **“greedy”** algorithm
  - Greedy algorithms find solutions based on a sequence of choices which are **“locally”** optimal at each step.
- Nevertheless, Prim’s greedy strategy produces a globally optimum solution!
- What about optimal substructure property?



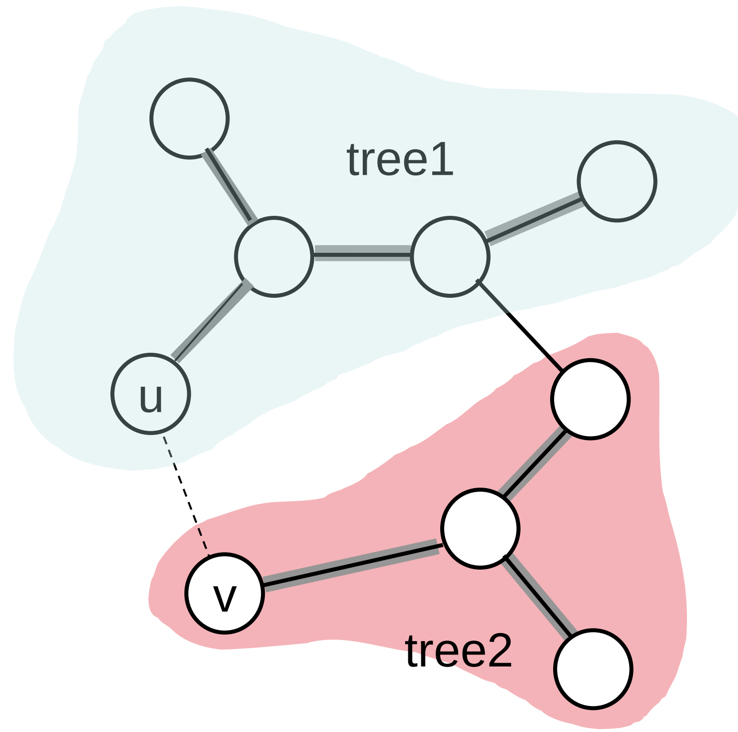
a
b
c
d
e
f
g
h
i
4
8
11
7
2
6
1
2
4
14
9
10
a-b

A different instance of the generic approach

(instance 1)



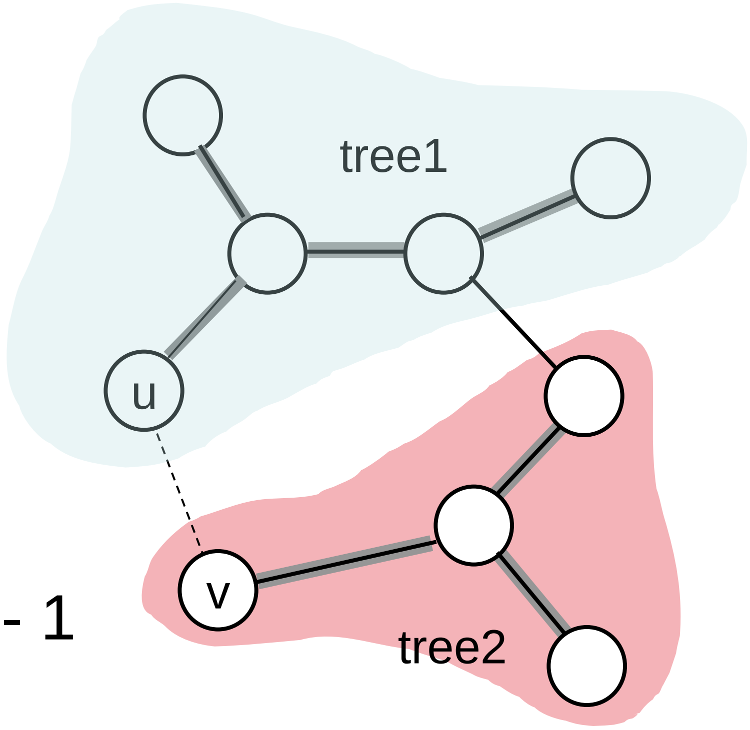
(instance 2)



- *A* is a forest containing connected components
  - Initially, each component is a single vertex
- Any safe edge merges two of these components into one
  - Each component is a tree

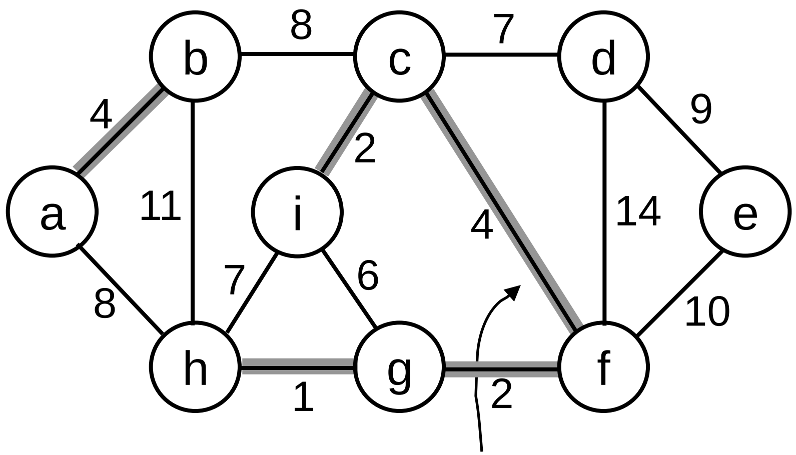
Kruskal's Algorithm

- How is it different from Prim's algorithm?
  - Prim's algorithm grows one tree all the time
  - Kruskal's algorithm grows multiple trees (i.e., a forest) at the same time.
  - Trees are merged together using **safe** edges
  - Since an MST has exactly |V| - 1 edges, after |V| - 1 merges, we would have only one component



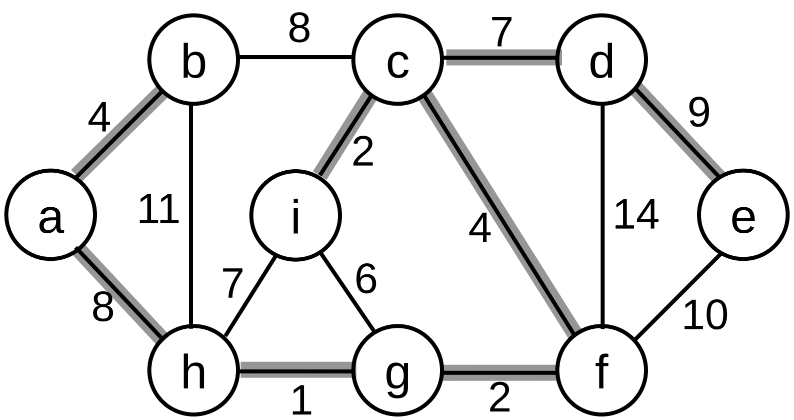
Kruskal's Algorithm

- Start with each vertex being its own component
- Repeatedly merge two components into one by choosing the **light** edge that connects them
- Which components to consider at each iteration?
  - Scan the set of edges in monotonically increasing order by weight



We would add edge (c, f)

Example



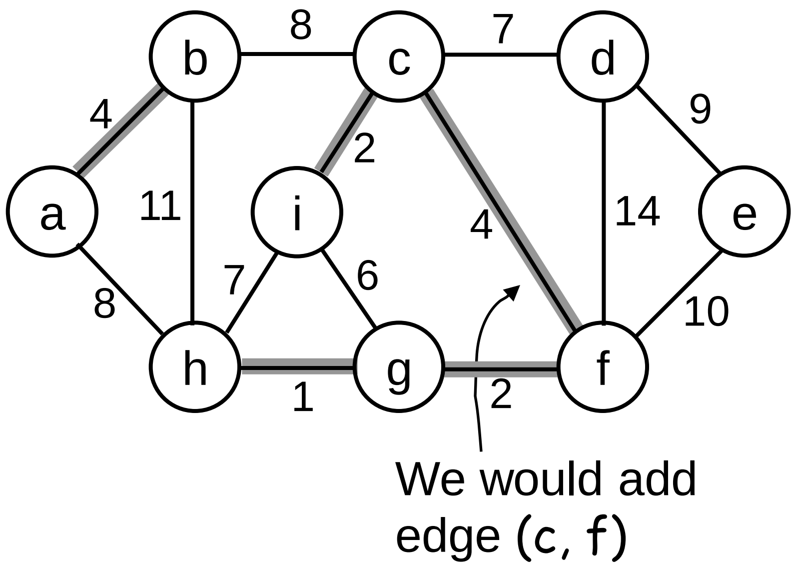
1: (h, g)
2: (c, i), (g, f)
4: (a, b), (c, f)
6: (i, g)
7: (c, d), (i, h)
8: (a, h), (b, c)
9: (d, e)
10: (e, f)
11: (b, h)
14: (d, f)

{a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}

1. Add (h, g) {g, h}, {a}, {b}, {c}, {d}, {e}, {f}, {i}
2. Add (c, i) {g, h}, {c, i}, {a}, {b}, {d}, {e}, {f}
3. Add (g, f) {g, h, f}, {c, i}, {a}, {b}, {d}, {e}
4. Add (a, b) {g, h, f}, {c, i}, {a, b}, {d}, {e}
5. Add (c, f) {g, h, f, c, i}, {a, b}, {d}, {e}
6. Ignore (i, g) {g, h, f, c, i}, {a, b}, {d}, {e}
7. Add (c, d) {g, h, f, c, i, d}, {a, b}, {e}
8. Ignore (i, h) {g, h, f, c, i, d}, {a, b}, {e}
9. Add (a, h) {g, h, f, c, i, d, a, b}, {e}
10. Ignore (b, c) {g, h, f, c, i, d, a, b}, {e}
11. Add (d, e) {g, h, f, c, i, d, a, b, e}
12. Ignore (e, f) {g, h, f, c, i, d, a, b, e}
13. Ignore (b, h) {g, h, f, c, i, d, a, b, e}
14. Ignore (d, f) {g, h, f, c, i, d, a, b, e}

Implementation of Kruskal's Algorithm

- Uses a disjoint-set data structure to determine whether an edge connects vertices in different components



Operations on Disjoint Data Sets

- MAKE-SET(u) – creates a new set whose only member is u
- FIND-SET(u) – returns a representative element from the set that contains u
  - Any of the elements of the set that has a particular property
  - *E.g.:* $S_u = \{r, s, t, u\}$, the property is that the element be the first one alphabetically

    FIND-SET(u) = r FIND-SET(s) = r
  - FIND-SET has to return the same value for a given set

Operations on Disjoint Data Sets

- UNION(u, v) – unites the dynamic sets that contain u and v, say S_u and S_v
  - *E.g.:* $S_u = \{r, s, t, u\}$, $S_v = \{v, x, y\}$

    UNION $(u, v) = \{r, s, t, u, v, x, y\}$
- Running time for FIND-SET and UNION depends on implementation.

KRUSKAL(V, E, w) (cont.)

A ← ∅
for each vertex v ∈ V
    do MAKE-SET(v) — **O(V)**
sort E into non-decreasing order by *w* — **O(ElgE)**
for each (u, v) taken from the sorted list ← **O(E)**
    do if FIND-SET(u) ≠ FIND-SET(v)
        then A ← A ∪ {(u, v)}
            UNION(u, v) ← **O(lgV)**
return A

- Running time: O(V+ElgE+ElgV)=O(ElgE)
- Since E=O(V^2), we have lgE=O(2lgV)=O(lgV)

O(ElgV)

Kruskal's Algorithm

- Kruskal's algorithm is a **"greedy"** algorithm
- Kruskal's greedy strategy produces a globally optimum solution
- Proof for generic approach applies to Kruskal's algorithm too