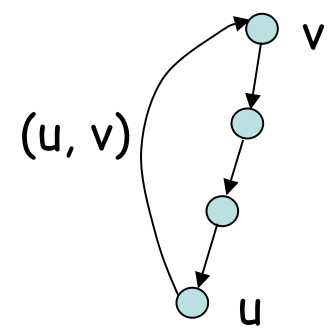
Analysis of Algorithms
CS 477/677

Instructor: Monica Nicolescu

Lecture 24

Lemma

A directed graph is **acyclic** $\Longleftrightarrow$ a DFS on G yields no back edges.



Proof:

"⇒": acyclic ⇒ no back edge

- Assume **back edge** ⇒ prove **cycle**
- Assume there is a back edge **(u, v)**

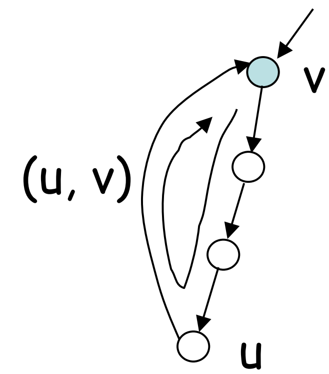
⇒ **v** is an ancestor of **u**

⇒ there is a path from **v** to **u** in G (**v** ⇨ **u**)

⇒ **v** ⇨ **u** + the back edge **(u, v)** yield a cycle

Lemma

A directed graph is **acyclic** ⟺ a DFS on G yields no back edges.



Proof:

"⟸": no back edge ⇒ acyclic

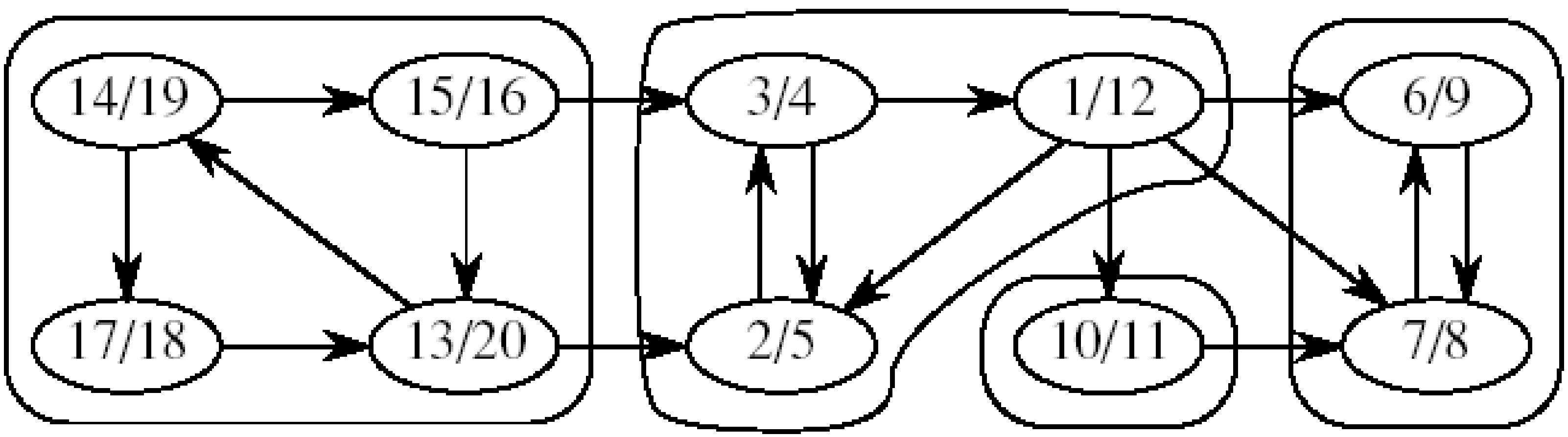
– Assume **cycle** ⇒ prove **back edge**
– Suppose G contains cycle **c**
– Let **v** be the first vertex discovered in **c**, and **(u, v)** be the preceding edge in **c**
– At time **d[v]**, vertices of **c** form a white path **v** ⇨ **u**
– **u** is descendant of **v** in depth-first forest (by white-path theorem)

⇒ **(u, v)** is a back edge

Strongly Connected Components

Given directed graph G = (V, E):

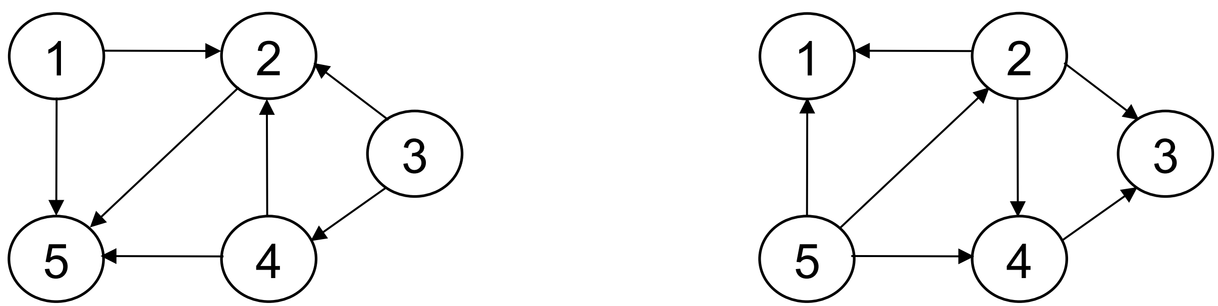
A **strongly connected component** (**SCC**) of G is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices **u**, **v** $\in C$, we have both **u** ⇨ **v** and **v** ⇨ **u**.



The Transpose of a Graph

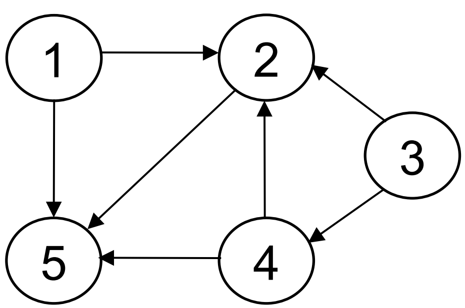
- G^T = **transpose** of G
  - G^T is G with all edges reversed
  - $G^T = (V, E^T)$, $E^T = \{(u, v) : (v, u) \in E\}$
- If using adjacency lists: we can create G^T in

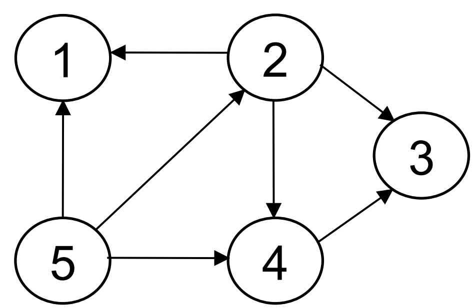
  $\Theta(|V| + |E|)$ time



Finding the SCC

- **Observation:** G and G^T have the same SCC's
  - **u** and **v** are reachable from each other in G $\Longleftrightarrow$ they are reachable from each other in G^T
- Idea for computing the SCC of a graph G = (V, E):
  - Make two depth first searches: one on G and one on G^T

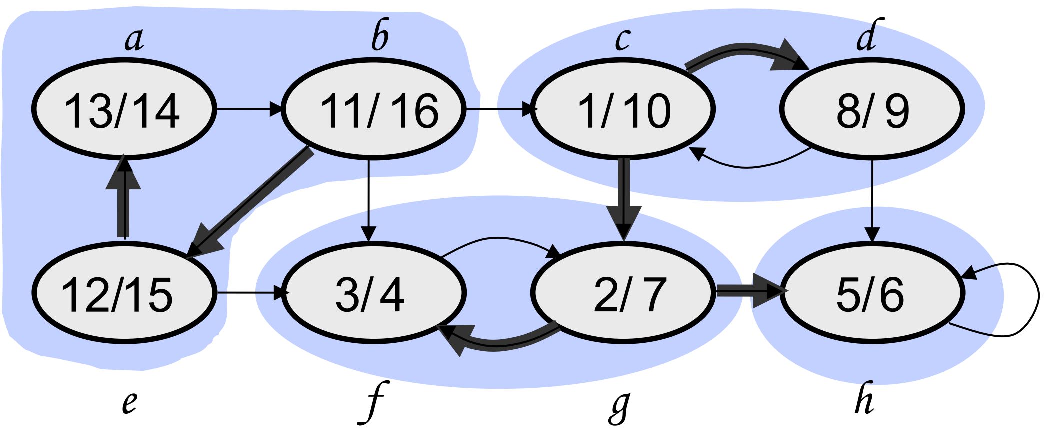




STRONGLY-CONNECTED-COMPONENTS(G)

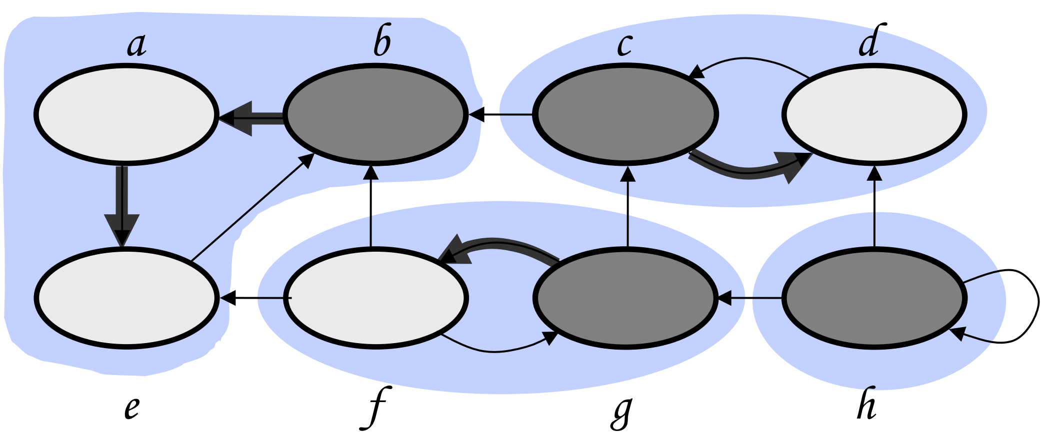
1. call DFS(G) to compute finishing times `f[u]` for each vertex `u`
2. compute G^T
3. call DFS(G^T), but in the main loop of DFS, consider vertices in order of decreasing `f[u]` (as computed in first DFS)
4. output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC

Example



DFS on the initial graph G

b	e	a	c	d	g	h	f
16	15	14	10	9	7	6	4

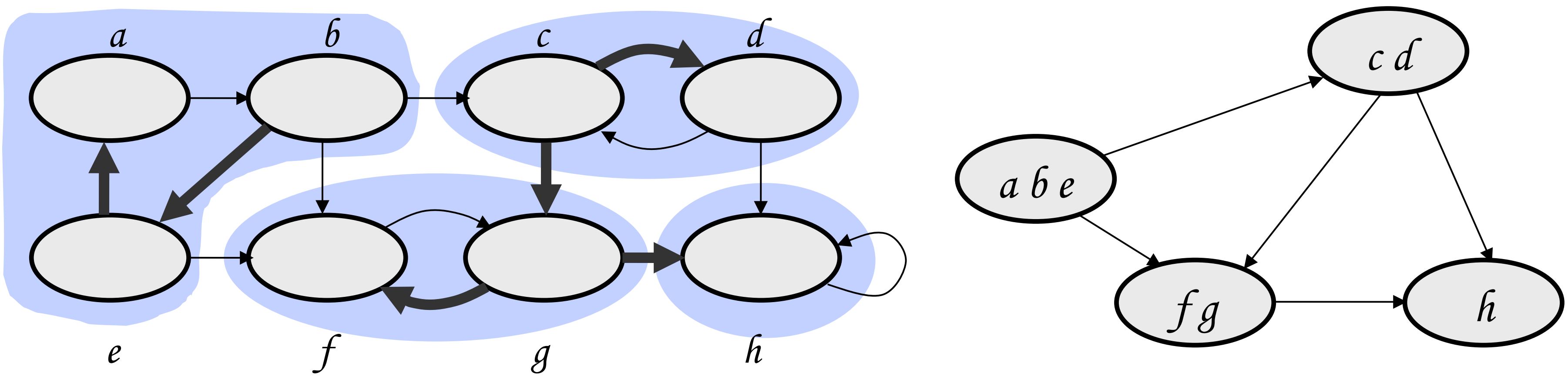


DFS on G^T:

- start at b: visit a, e
- start at c: visit d
- start at g: visit f
- start at h

Strongly connected components: $C_1 = \{a, b, e\}$, $C_2 = \{c, d\}$, $C_3 = \{f, g\}$, $C_4 = \{h\}$

Component Graph



- The **component graph** $G^{SCC} = (V^{SCC}, E^{SCC})$:
  - $V^{SCC} = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k\}$, where $\mathbf{v}_i$ corresponds to each strongly connected component C_i
  - There is an edge $(\mathbf{v}_i, \mathbf{v}_j) \in E^{SCC}$ if G contains a directed edge $(\mathbf{x}, \mathbf{y})$ for some $\mathbf{x} \in C_i$ and $\mathbf{y} \in C_j$
- The component graph is a DAG

Lemma 1

Let C and C' be distinct SCC's in G

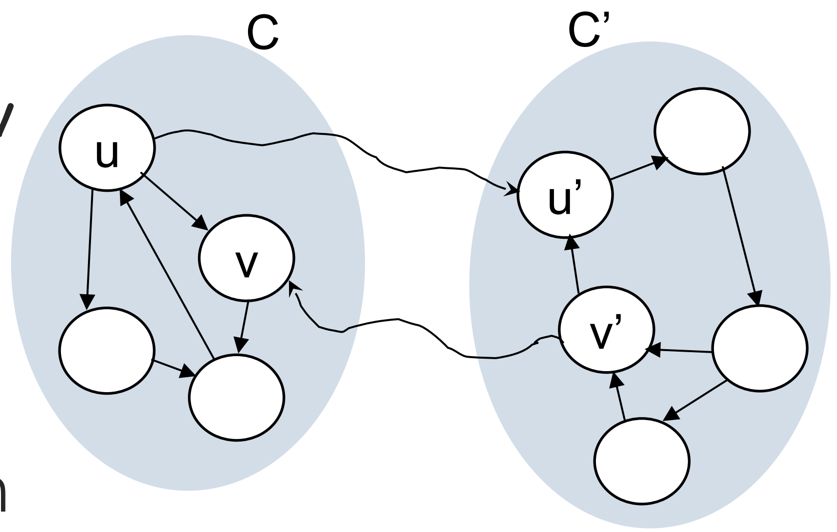
Let **u**, **v** ∈ C, and **u'**, **v'** ∈ C'

Suppose there is a path **u** ⇨ **u'** in G

Then there cannot also be a path **v'** ⇨ **v** in G.

Proof

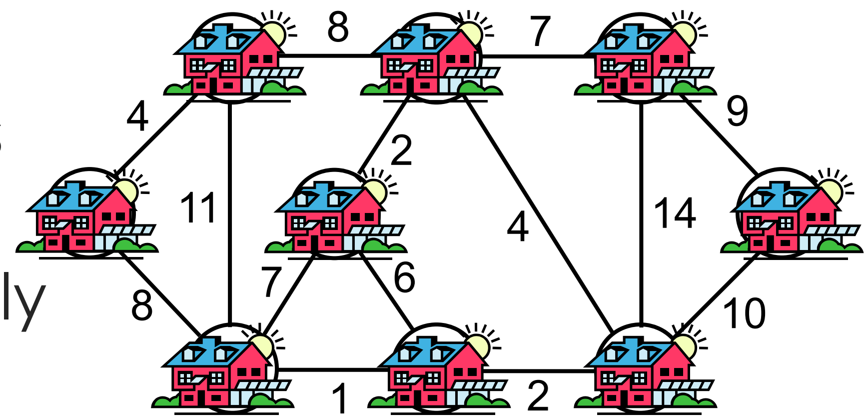
- Suppose there is a path **v'** ⇨ **v**
- There exists **u** ⇨ **u'** ⇨ **v'**
- There exists **v'** ⇨ **v** ⇨ **u**
- u and v' are reachable from each other, so they are not in separate SCC's: contradiction!



Minimum Spanning Trees

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:

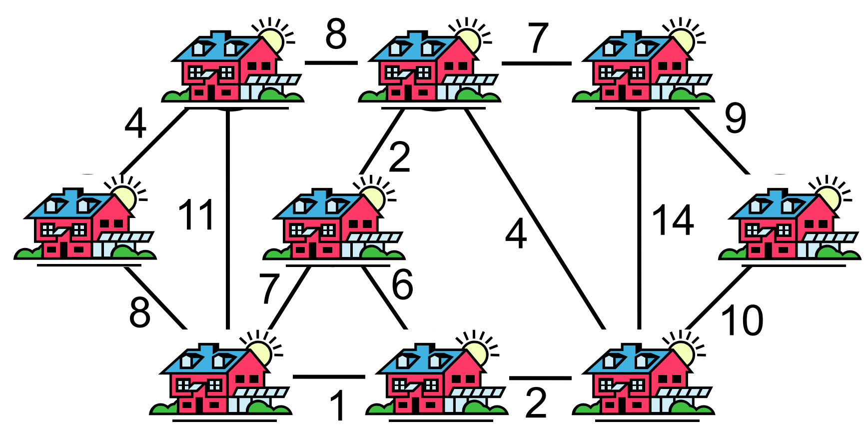
1. Everyone stays connected: can reach every house from all other houses, and
2. Total repair cost is minimum

Minimum Spanning Trees

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

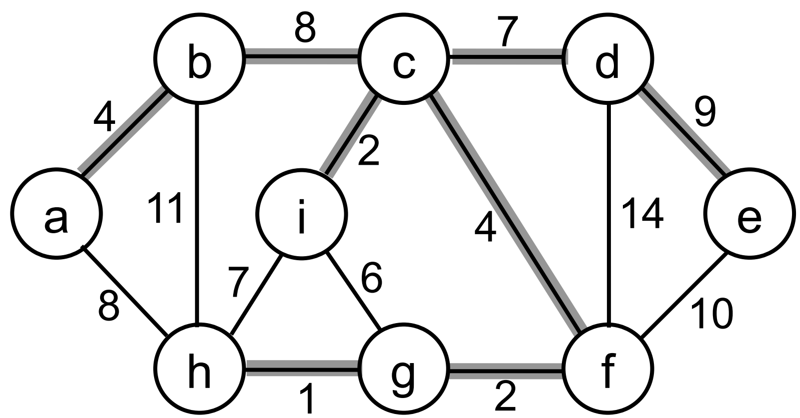
Find T ⊆ E such that:

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



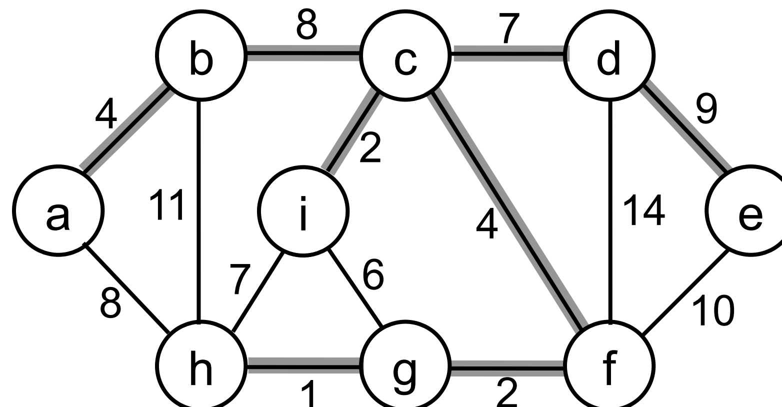
Minimum Spanning Trees

- T forms a tree = **spanning tree**
- A spanning tree whose weight is minimum over all spanning trees is called a ***minimum spanning tree***, or ***MST***.



Properties of MSTs

- Minimum spanning trees are not unique
  - Can replace (b, c) with (a, h) to obtain a different spanning tree with the same cost
- MST have no cycles
  - We can take out an edge of a cycle, and still have all vertices connected while reducing the cost



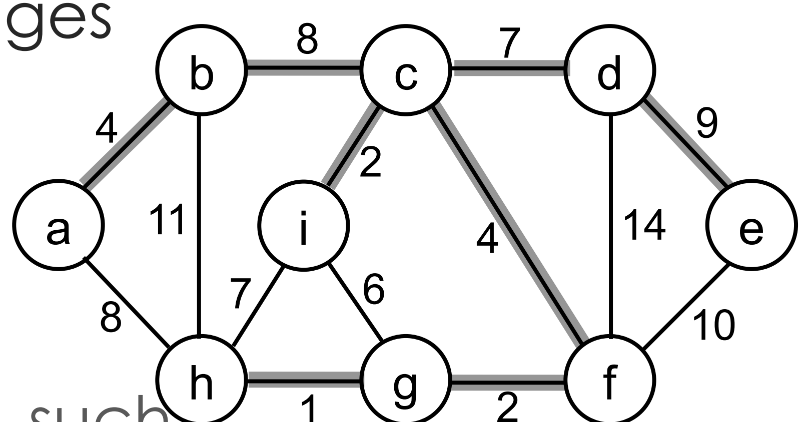
- # of edges in a MST:
  - |V| - 1

Growing a MST

Minimum-spanning-tree problem: find a MST for a connected, undirected graph, with a weight function associated with its edges

A generic solution:

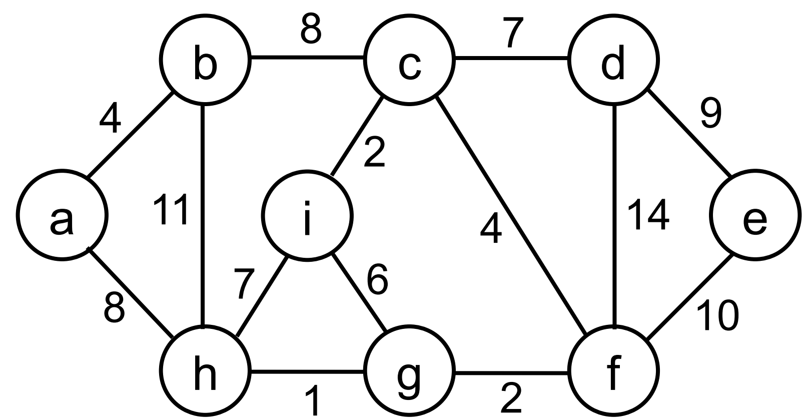
- Build a set A of edges (initially empty)
- Incrementally add edges to A such that they would belong to a MST
- An edge **(u, v)** is **safe** for A if and only if A ∪ {(u, v)} is also a subset of some MST – **greedy choice property**



We will add only safe edges

GENERIC-MST

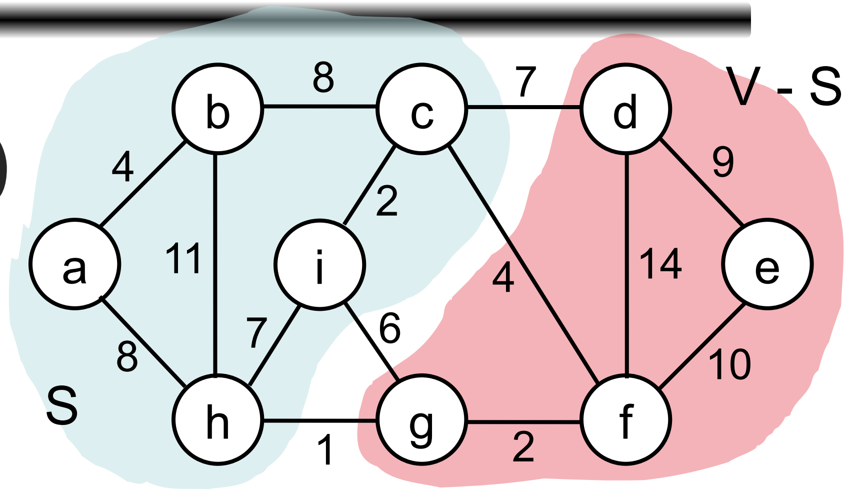
1. $A \leftarrow \varnothing$
2. **while** A is not a spanning tree
3. **do** find an edge **(u, v)** that is safe for A
4. $A \leftarrow A \cup \{(u, v)\}$
5. **return** A



- 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 ⊂ 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)**?



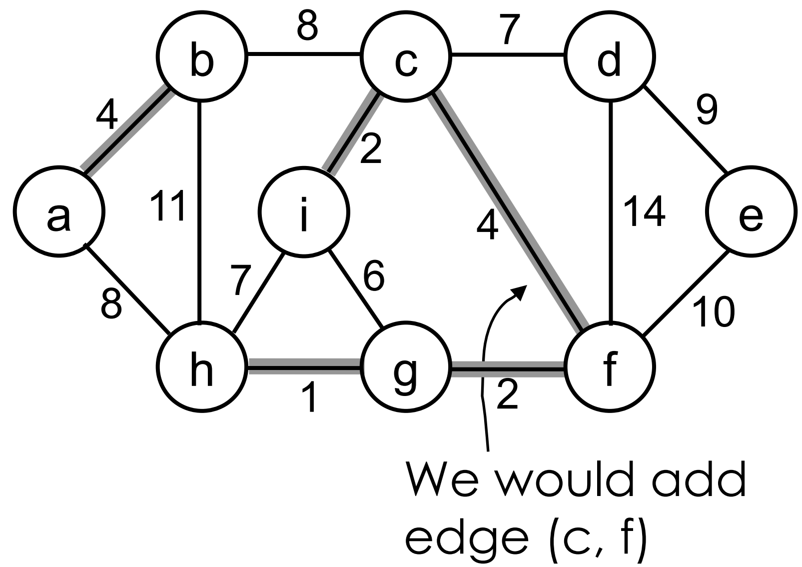
Discussion

In GENERIC-MST:

- ***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
- Since an MST has exactly $|V|$ - 1 edges: after iterating $|V|$ - 1 times, we have only one component

The Algorithm of Kruskal

- Start with each vertex being its own component
- Repeatedly merge two components into one by choosing the light edge that connects them



- Scan the set of edges in monotonically increasing order by weight
- 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**
  - May be any of the elements of the set that has a particular property
  - *E.g.:* $S_u = \{r, s, t, u\}$, the property may be that the element is 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 $\mathbf{S}_u$ and $\mathbf{S}_v$
  - *E.g.:* $S_u = \{r, s, t, u\}$, $S_v = \{v, x, y\}$

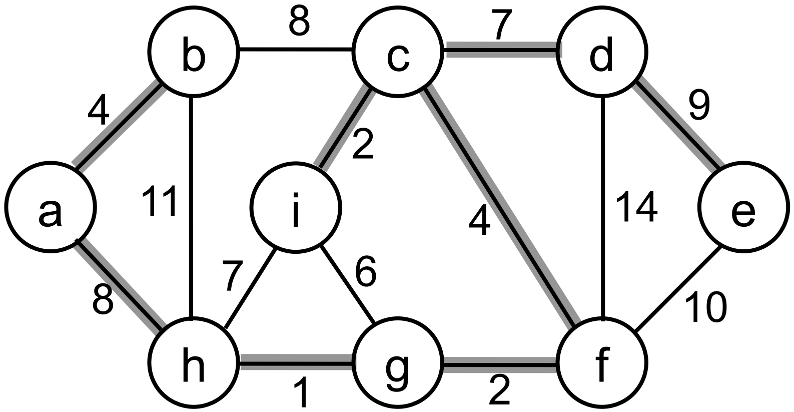
    UNION (u, v) = {r, s, t, u, v, x, y}

KRUSKAL(V, E, w)

```
A ← ∅
for each vertex v ∈ V
        do MAKE-SET(v)
sort E into increasing order by weight w
for each (u, v) taken from the sorted list
     do if FIND-SET(u) ≠ FIND-SET(v)
           then A ← A ∪ {(u, v)}
                UNION(u, v)
return A
```

Running time: $O(|E| \lg|V|)$ – dependent on the implementation of the disjoint-set data structure

Example



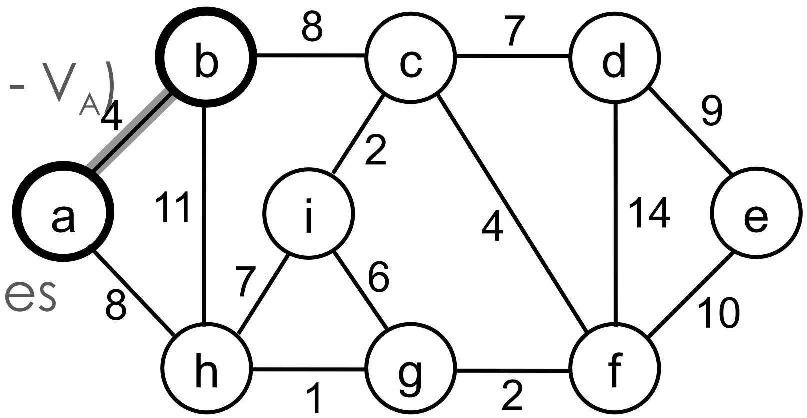
1: (h, g)
2: (c, i), (g, f)
4: (a, b), (c, f)
6: (i, g)
7: (c, d), (i, h)
8: (a, h), (b, c)
9: (d, e)
10: (e, f)
11: (b, h)
14: (d, f)

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

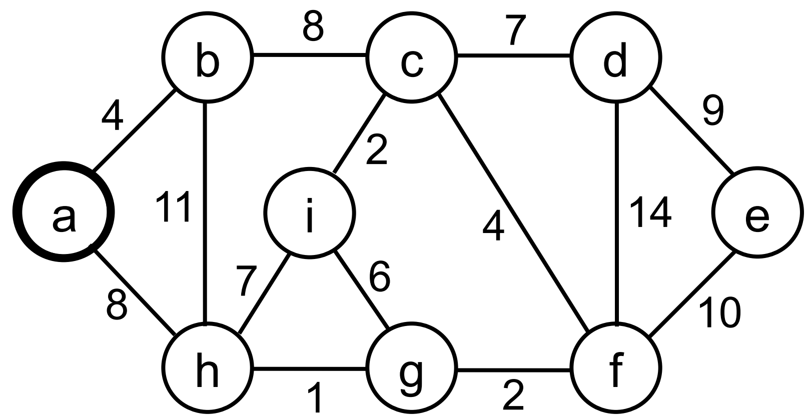
1. Add (h, g) {g, h}, {a}, {b}, {c}, {d}, {e}, {f}, {i}
2. Add (c, i) {g, h}, {c, i}, {a}, {b}, {d}, {e}, {f}
3. Add (g, f) {g, h, f}, {c, i}, {a}, {b}, {d}, {e}
4. Add (a, b) {g, h, f}, {c, i}, {a, b}, {d}, {e}
5. Add (c, f) {g, h, f, c, i}, {a, b}, {d}, {e}
6. Ignore (i, g) {g, h, f, c, i}, {a, b}, {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}

The Algorithm of Prim

- The edges in set A always form a single tree
- Starts from an arbitrary "root": $V_A = \{a\}$
- At each step:
  - Find a safe edge connecting $(V_A, V - V_A)$
  - Add this edge to A
  - Repeat until the tree spans all vertices
- Greedy strategy
  - At each step the edge added contributes the minimum amount possible to the weight of the tree



How to Find Light Edges Quickly?



Use a priority queue Q:

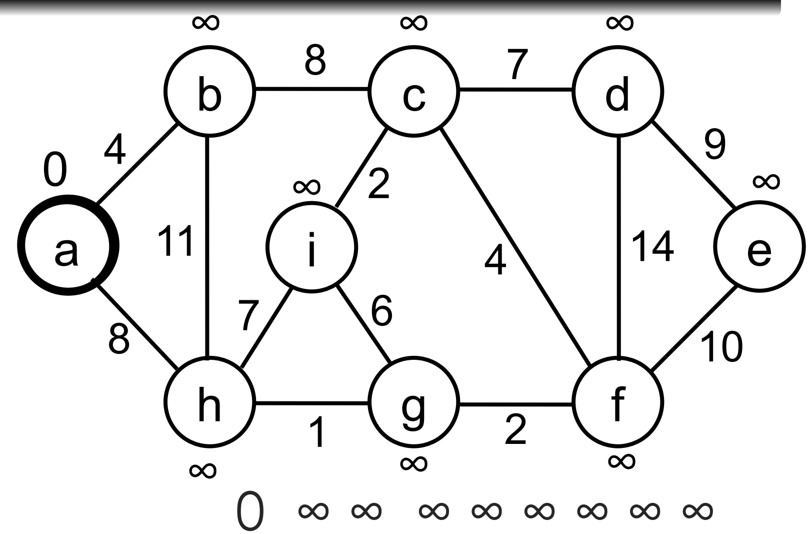
- Contains all vertices not yet included in the tree ($V - V_A$)
  - V = {a}, Q = {b, h, c, d, e, f, g, i}
- With each vertex **v** we associate a key:

  key[**v**] = minimum weight of any edge **(u, v)** connecting **v** to a vertex in the tree

  - Key of **v** is ∞ if v is not adjacent to any vertices in V_A
  - After adding a new node to V_A we update the weights of all the nodes adjacent to it
  - We added node a ⇒ key[b] = 4, key[h] = 8

PRIM(V, E, w, r)

```
Q ← ∅
for each u ∈ V
    do key[u] ← ∞
        π[u] ← NIL
        INSERT(Q, u)
DECREASE-KEY(Q, 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))
```

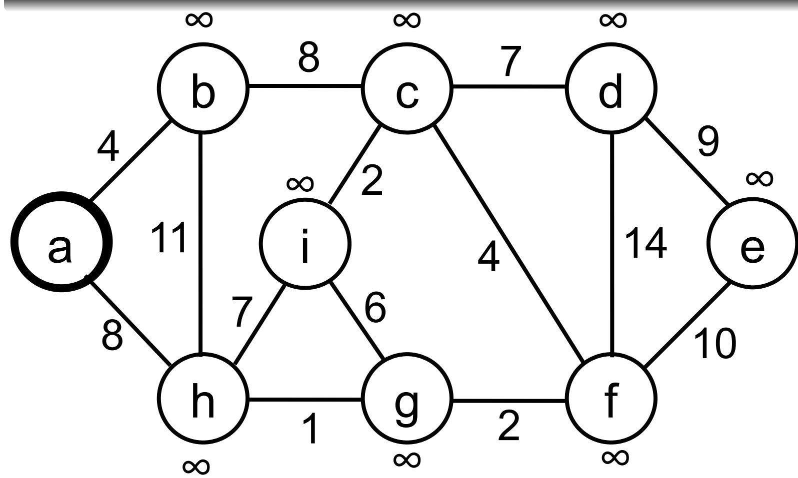


$Q = \{a, b, c, d, e, f, g, h, i\}$

$V_A = \emptyset$

Extract-MIN(Q) ⇒ a

Example

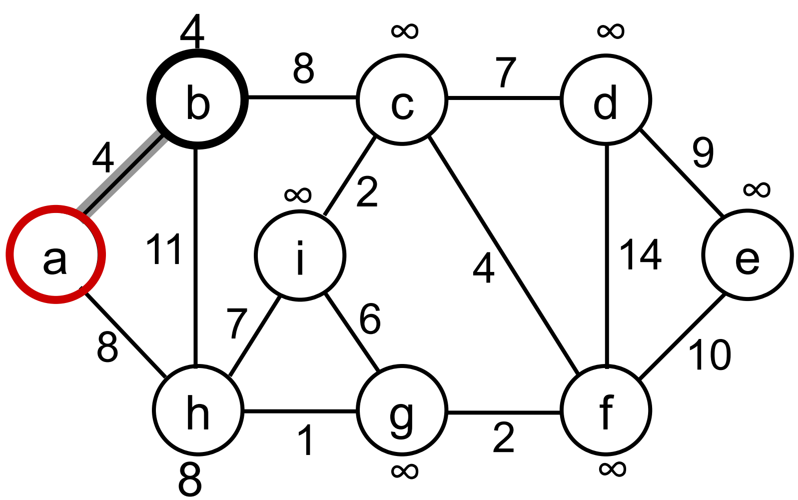


$0 \;\; \infty \;\; \infty \;\; \infty \;\; \infty \;\; \infty \;\; \infty \;\; \infty \;\; \infty$

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

$V_A = \emptyset$

Extract-MIN(Q) $\Rightarrow$ a



key [b] = 4 $\quad \boldsymbol{\pi}$ [b] = a

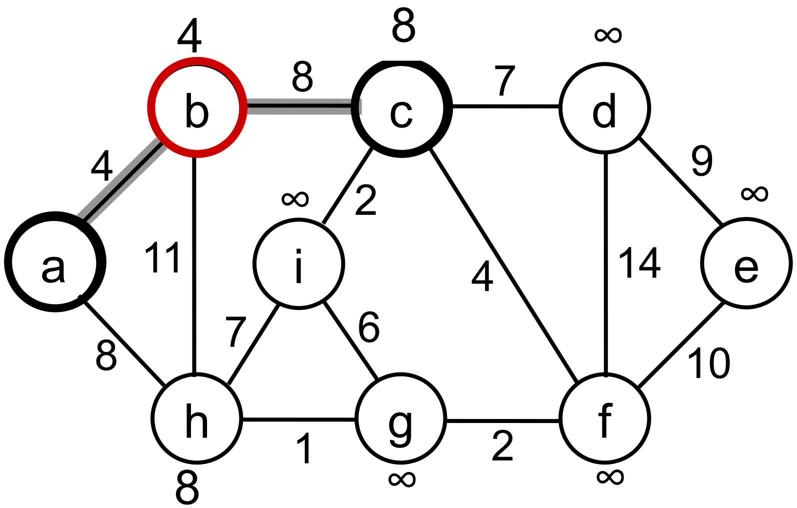
key [h] = 8 $\quad \boldsymbol{\pi}$ [h] = a

$4 \;\; \infty \;\; \infty \;\; \infty \;\; \infty \;\; \infty \;\; 8 \;\; \infty$

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

Extract-MIN(Q) $\Rightarrow$ b

Example



key [c] = 8 π [c] = b

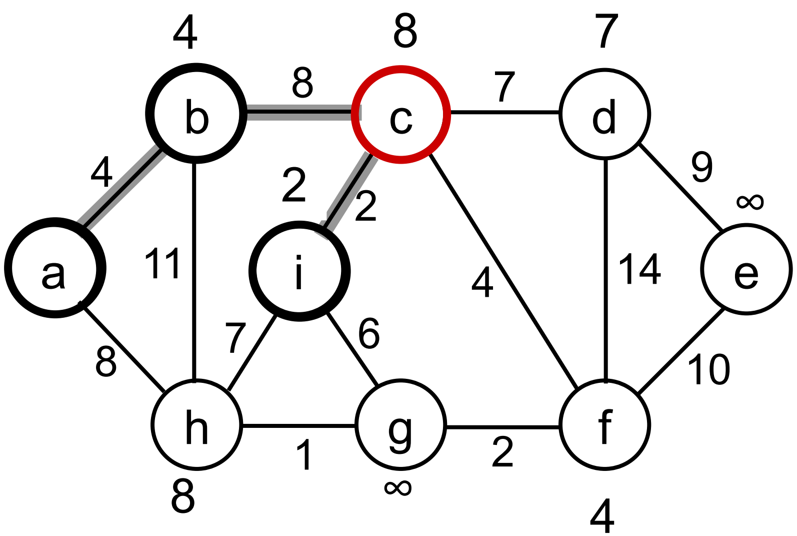
key [h] = 8 π [h] = a -

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

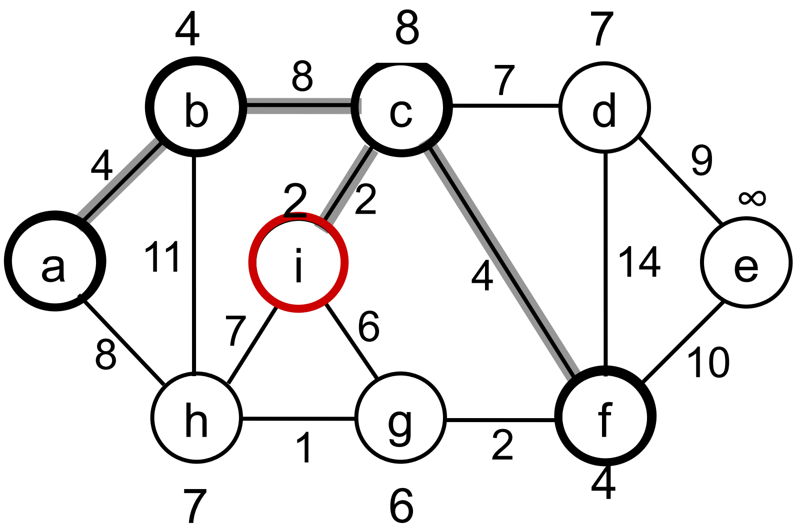
key [i] = 2 π [i] = c

7 ∞ 4 ∞ 8 2

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

Extract-MIN(Q) ⇒ i

Example



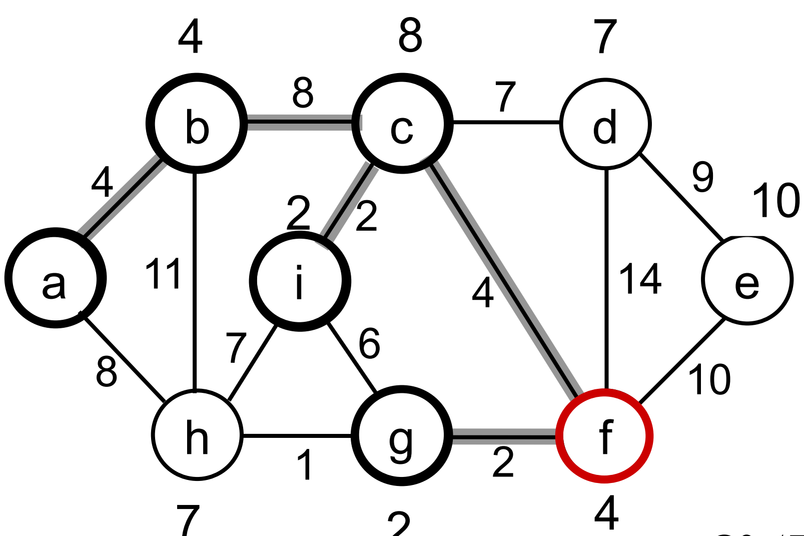
key [h] = 7 π [h] = i

key [g] = 6 π [g] = i

7 ∞ 4 6 7

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

Extract-MIN(Q) ⇒ f



key [g] = 2 π [g] = f

key [d] = 7 π [d] = c

unchanged

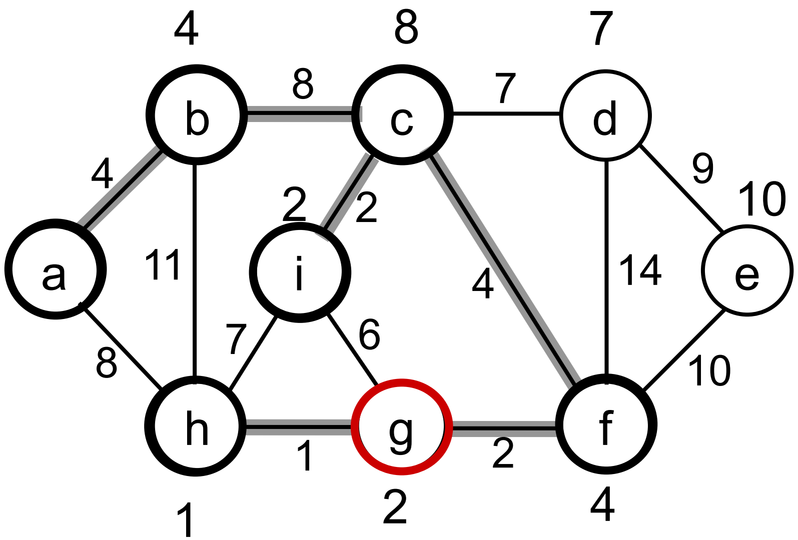
key [e] = 10 π [e] = f

7 10 2 7

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

Extract-MIN(Q) ⇒ g

Example

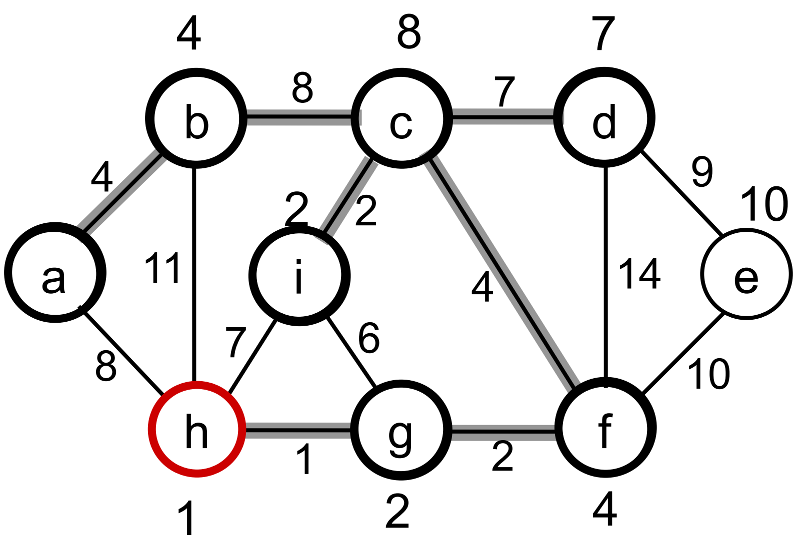


key [h] = 1 $\qquad$ **π** [h] = g

7 10 1

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

Extract-MIN(Q) ⇒ h

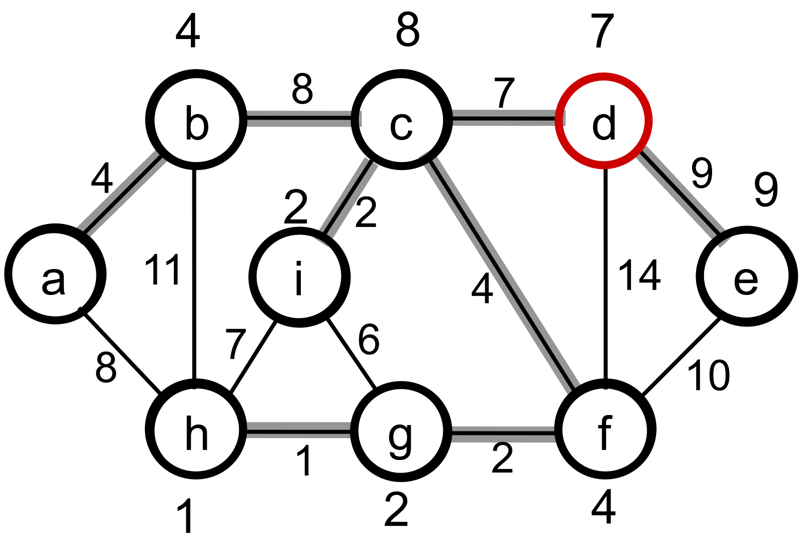


7 10

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

Extract-MIN(Q) ⇒ d

Example



key [e] = 9 $\boldsymbol{\pi}$ [e] = d

9

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

Extract-MIN(Q) ⇒ e

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

PRIM(V, E, w, r)

```
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))
```

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

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

Executed V times

Takes $O(\lg V)$

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

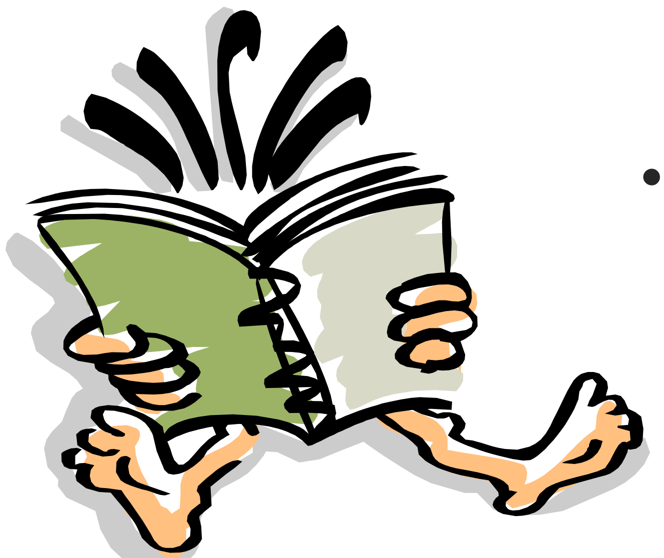
Executed O(E) times

Constant

Takes $O(\lg V)$

$O(E\lg V)$

Readings



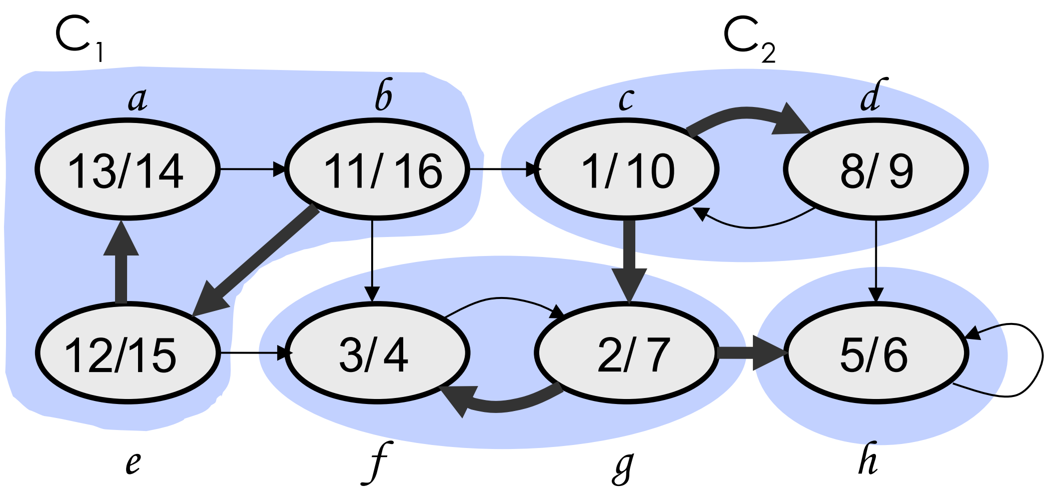
- Chapters 25, 31

optional

ADDITIONAL SLIDES

Notations

- Extend notation for **d** (starting time) and **f** (finishing time) to sets of vertices $U \subseteq V$:
  - $d(U) = \min_{u \in U} \{ d[u] \}$ (earliest discovery time)
  - $f(U) = \max_{u \in U} \{ f[u] \}$ (latest finishing time)



$d(C_1)=11$
$f(C_1)=16$

$d(C_2)=1$
$f(C_2)=10$

C_3
$d(C_3)=2$
$f(C_3)=7$

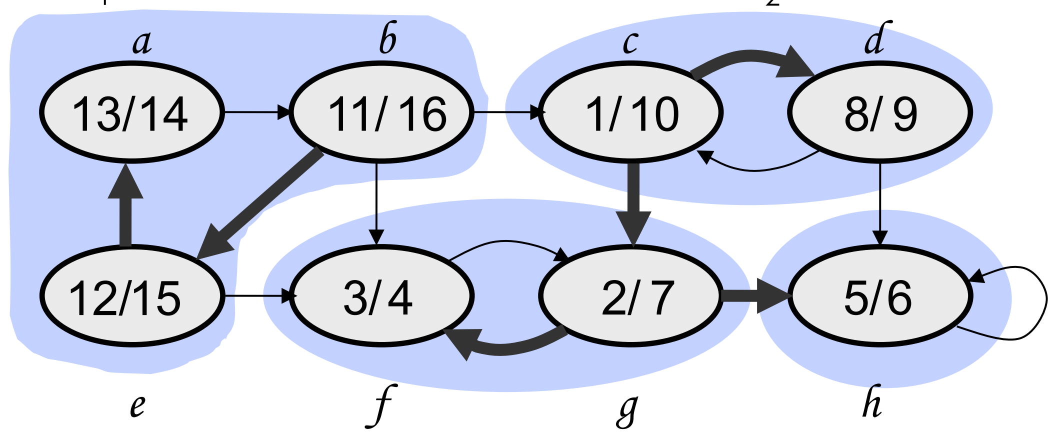
C_4
$d(C_4)=5$
$f(C_4)=6$

Lemma 2

- Let C and C' be distinct SCCs in a directed graph G = (V, E). If there is an edge **(u, v)** ∈ E, where **u** ∈ C and **v** ∈ C' then f(C) > f(C').
- Consider C_1 and C_2, connected by edge **(b, c)**

C_1 C_2

$d(C_1)=11$
$f(C_1)\ =16$



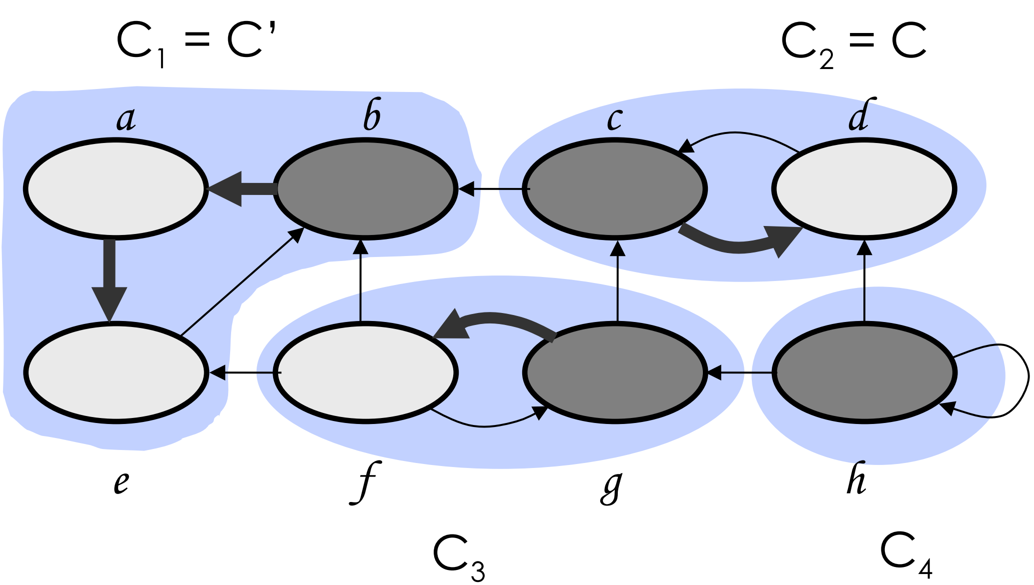
$d(C_2)\ =1$
$f(C_2)\ \ =10$

C_3
$d(C_3)=3$
$f(C_3)\ \ =7$

C_4
$d(C_4)=5$
$f(C_4)\ =6$

Corollary

- Let C and C' be distinct SCCs in a directed graph G = (V, E). If there is an edge **(u, v)** $\in E^T$, where **u** $\in$ C and **v** $\in$ C' then f(C) < f(C').
- Consider C_2 and C_1, connected by edge **(c, b)**



- Since **(c, b)** $\in E^T \Rightarrow$ **(b, c)** $\in$ E
- From previous lemma:

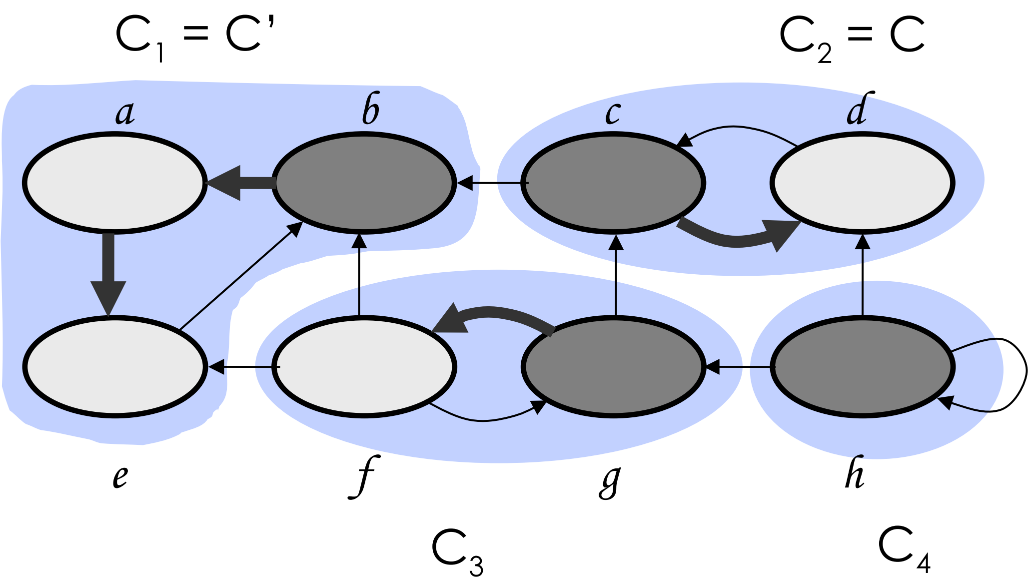
$f(C_1) > f(C_2)$

$f(C') > f(C)$

Discussion

$f(C) < f(C')$

- Each edge in G^T that goes between different components goes from a component with an earlier finish time (in the DFS) to one with a later finish time



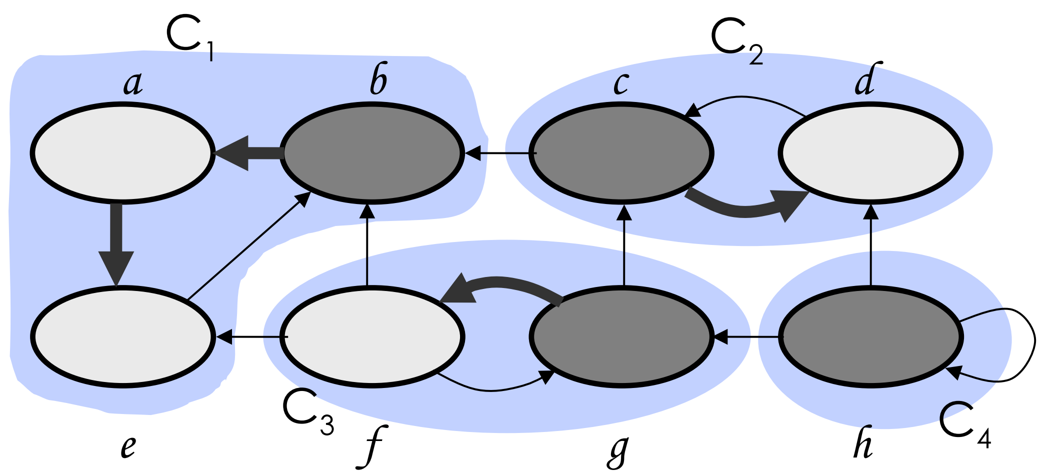
Why does SCC Work?

- When we do the second DFS, on G^T, we start with a component C such that f(C) is maximum (b, in our case)
- We start from **b** and visit all vertices in C_1
- From corollary: f(C) > f(C') for all C ≠ C' ⇒ there are no edges from C to any other SCCs in G^T

⇒ DFS will visit only vertices in C_1

⇒ The depth-first tree rooted at **b** contains exactly the vertices of C_1

b	e	a	c	d	g	h	f
16	15	14	10	9	7	6	4



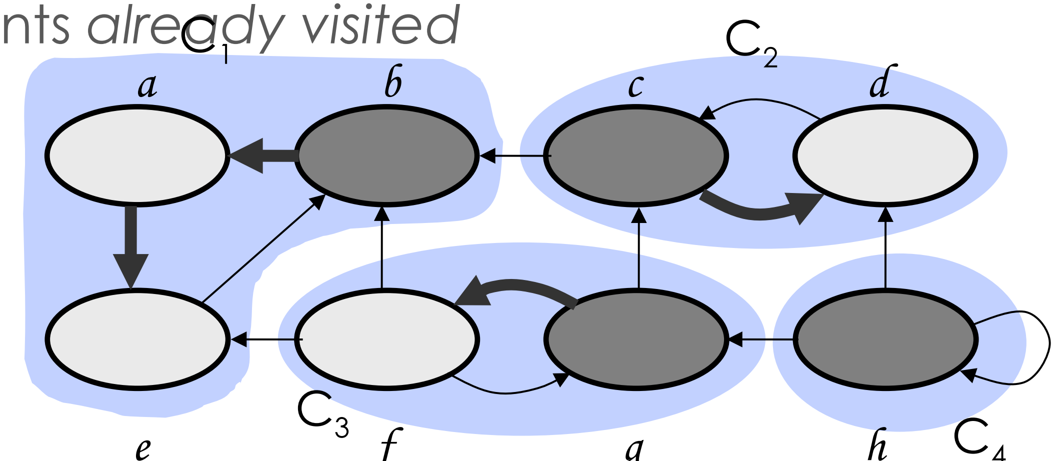
Why does SCC Work? (cont.)

- The next root chosen in the second DFS is in SCC C_2 such that f(C) is maximum over all SCC's other than C_1
- DFS visits all vertices in C_2
  - the only edges out of C_2 go to C_1, which we already visited

⇒ The only tree edges will be to vertices in C_2

- Each time we choose a new root it can reach only:
  - vertices in its own component
  - vertices in components *already visited*

b	e	a	c	d	g	h	f
16	15	14	10	9	7	6	4



Lemma 1

Let C and C' be distinct SCC's in G

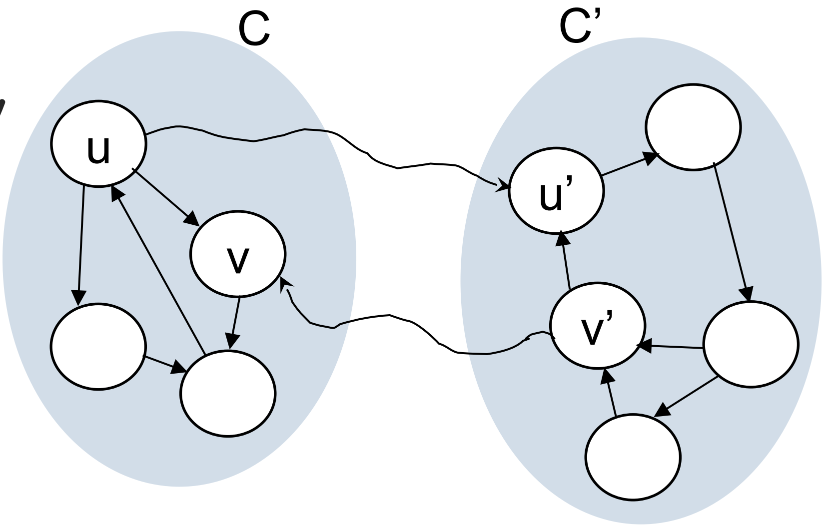
Let **u**, **v** ∈ C, and **u'**, **v'** ∈ C'

Suppose there is a path **u** ⇨ **u'** in G

Then there cannot also be a path **v'** ⇨ **v** in G.

Proof

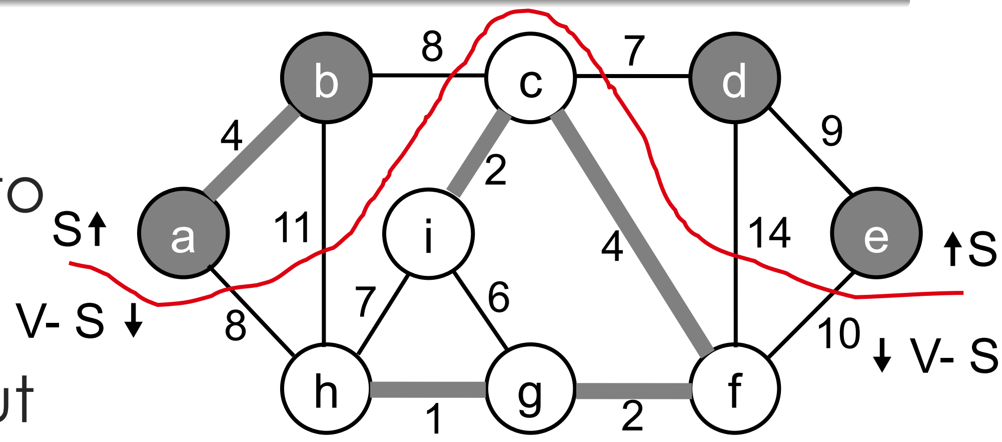
- Suppose there is a path **v'** ⇨ **v**
- There exists **u** ⇨ **u'** ⇨ **v'**
- There exists **v'** ⇨ **v** ⇨ **u**
- u and v' are reachable from each other, so they are not in separate SCC's: contradiction!



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

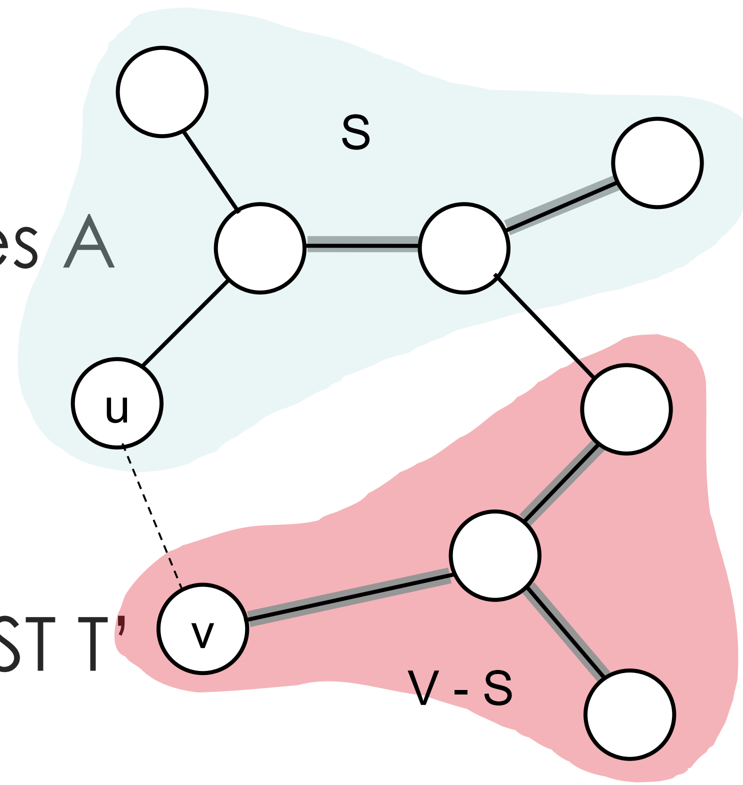
- A cut **respects** a set *A* of edges ⟺ no edge in A crosses the cut
- An edge is a **light edge** crossing a cut ⟺ its weight is minimum over all edges crossing the cut
  - For a given cut, there can be several light edges crossing it

Theorem

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

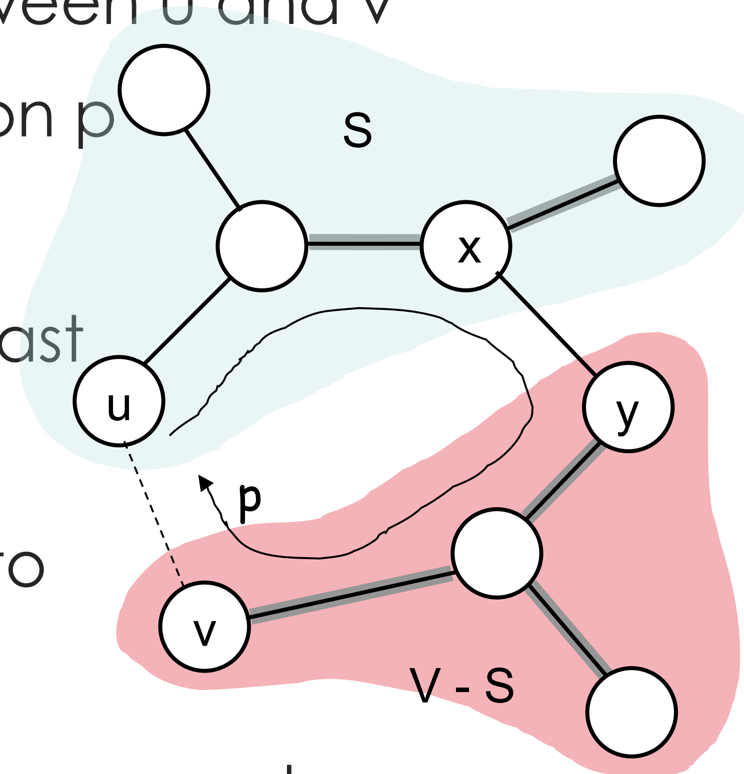
Proof:

- Let T be a MST that includes A
  - Edges in A are shaded
- Assume 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
- **(u, v)** forms a cycle with edges on p
- **(u, v)** crosses the cut ⇒ path p must cross the cut (S, V - S) at least once: let **(x, y)** be that edge
- Let's remove (x, y) ⇒ breaks T into two components.
- Adding (u, v) reconnects the components



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

Theorem – Proof (cont.)

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

Have to show that T' is a MST:

- (u, v) is a light edge

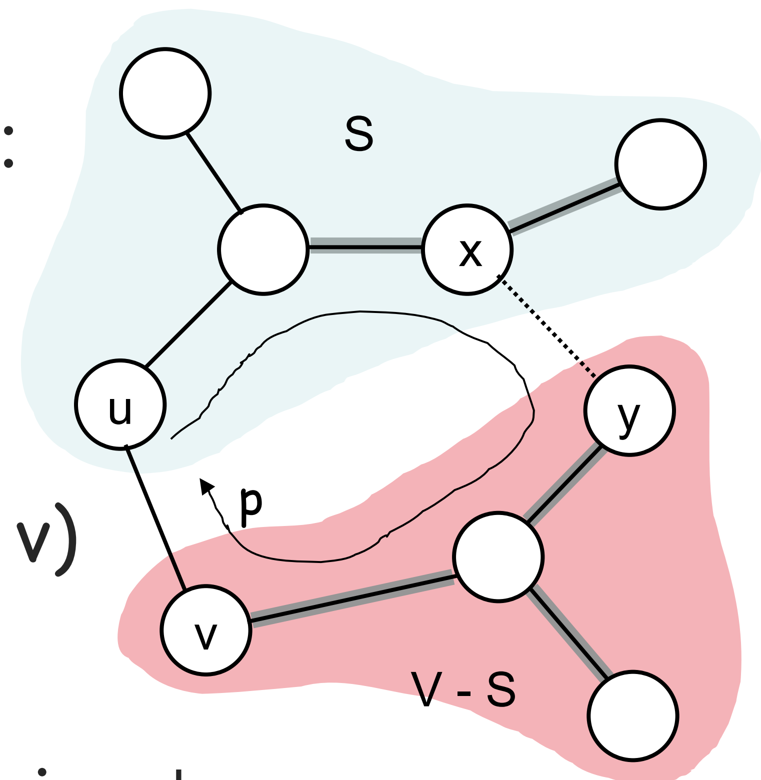
  $\Rightarrow w(u, v) \leq w(x, y)$

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

  $\leq w(T)$

- Since T is a minimum 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 a MST

- $A \subseteq T$ and **(x, y)** $\notin A \Rightarrow A \subseteq T'$
- $A \cup \{$**(u, v)**$\} \subseteq T'$
- Since T' is an MST

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