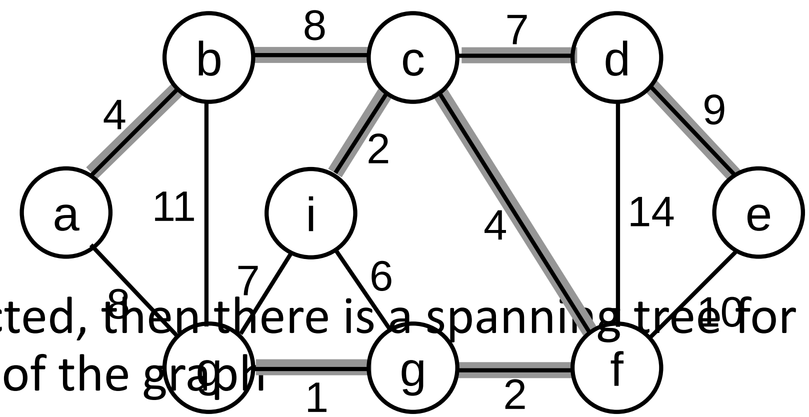
Minimum Spanning Trees (MST)

Minimum Spanning Trees

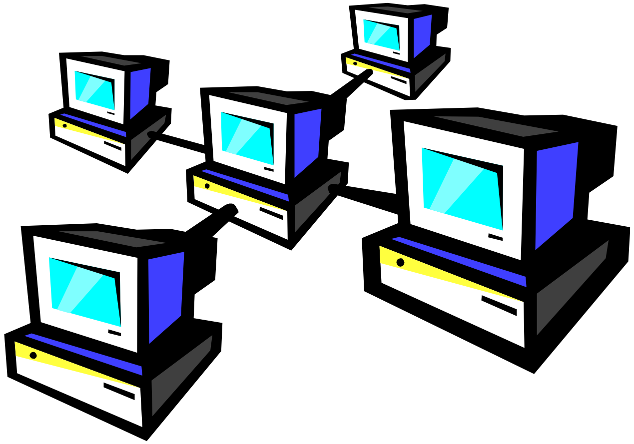
- Spanning Tree
  - A tree (i.e., connected, acyclic graph) which contains all the vertices of the graph
- Minimum Spanning Tree
  - Spanning tree with the **minimum sum of weights**

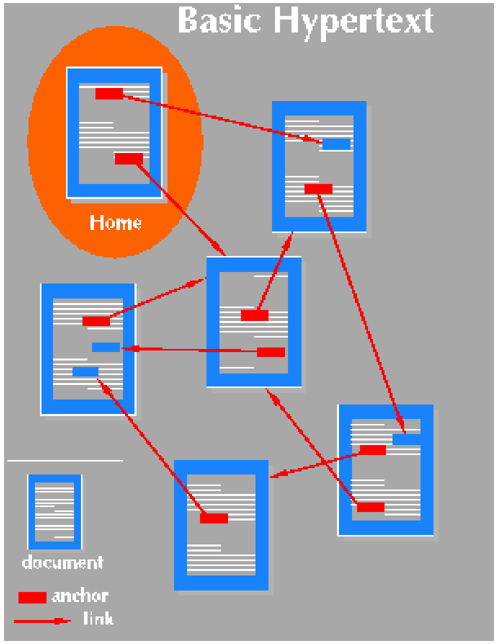


- Spanning forest
  - If a graph is not connected, then there is a spanning tree for each connected component of the graph

Applications of MST

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

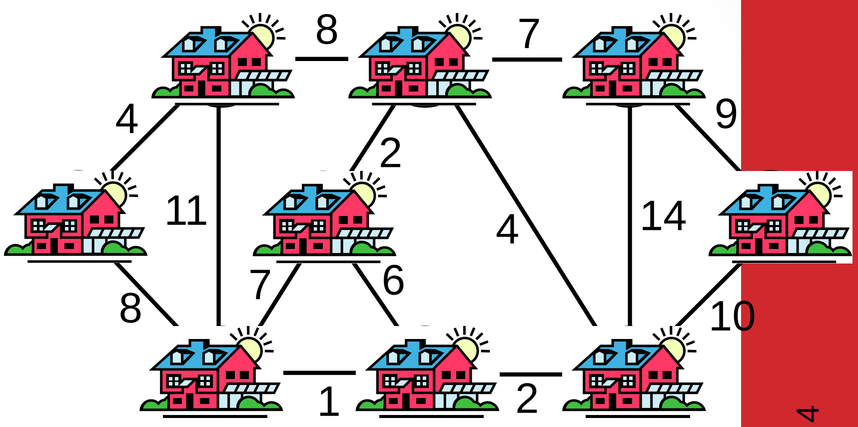




Example

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

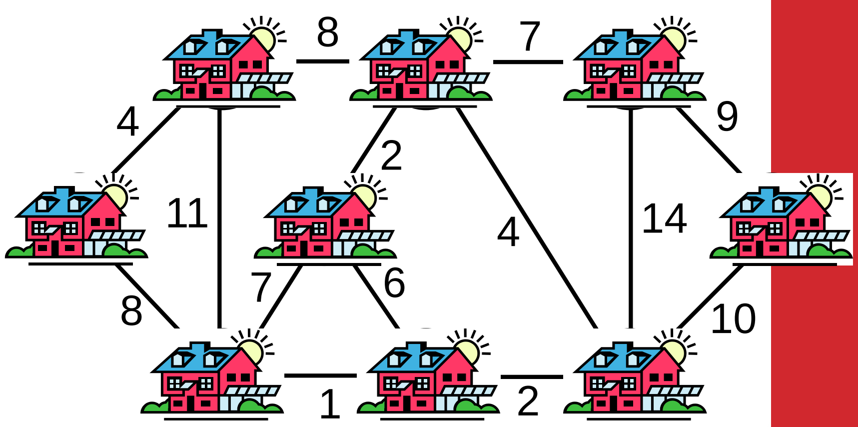
2. Total repair cost is minimum

Minimum Spanning Trees

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

Find $T \subseteq E$ such that:

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



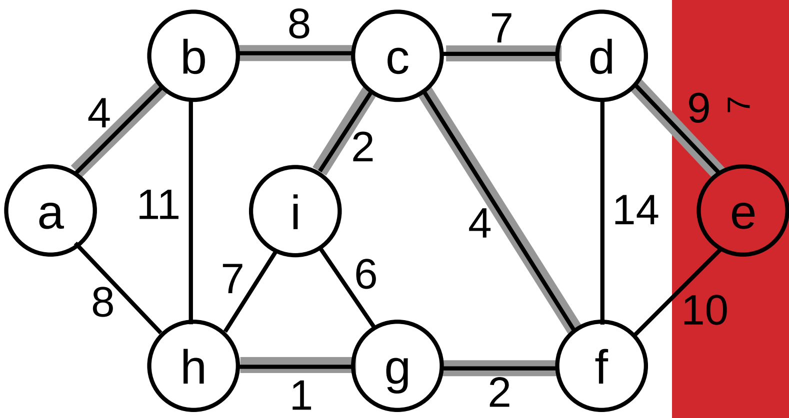
Properties of Minimum Spanning Trees

- Minimum spanning tree is **not** unique

- MST h
  - 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



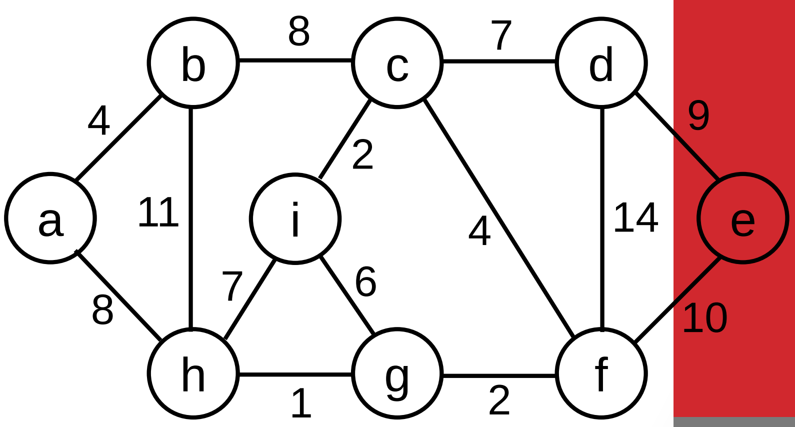
Idea: add only "safe" edges

– An edge (u, v) is **safe** for A if and only if A ∪ {(u, v)} is also a subset of **some** MST

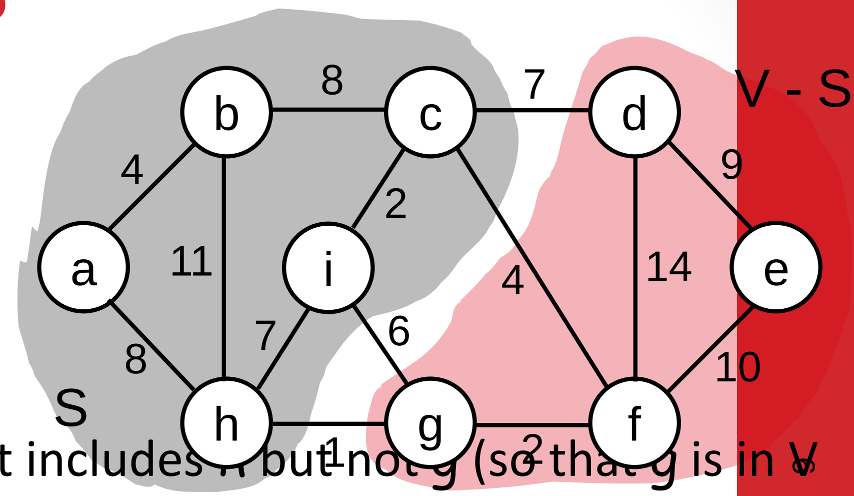
Generic MST algorithm

1. A ← ∅
2. **while** A is not a spanning tree
3. **do** find an edge (u, v) that is safe for A
4. A ← A ∪ {(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*)?

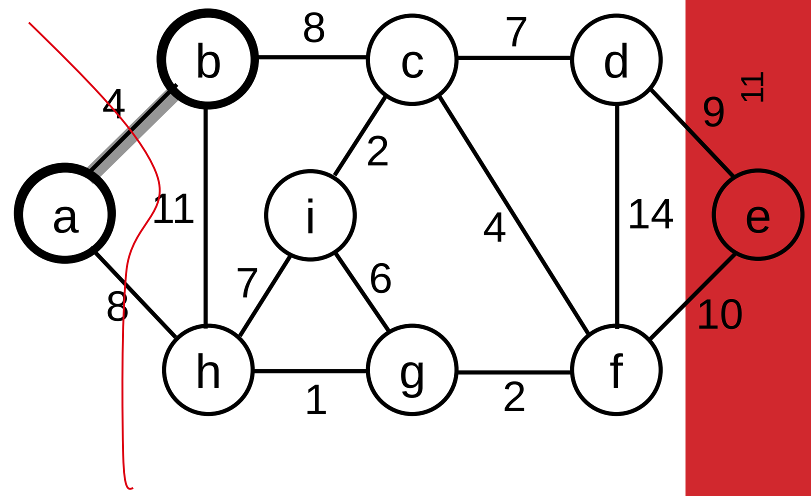
BIRZEIT UNIVERSITY

MST

- Prim’s Algorithm
- Kruskal Algorithm

Prim's Algorithm

- The edges in set A always form a single tree
- 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

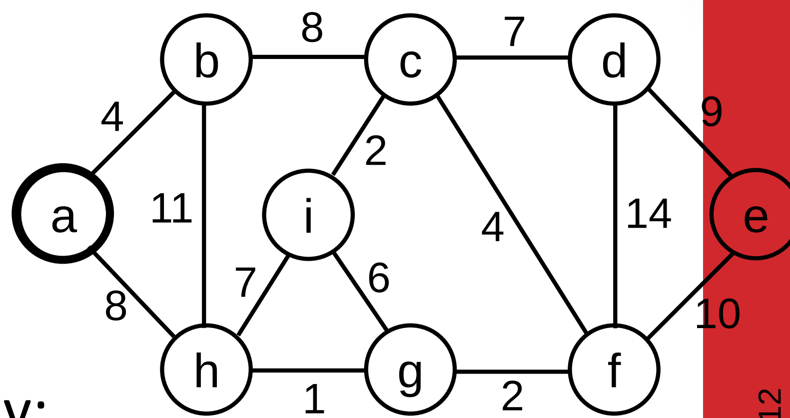


How to Find Light Edges Quickly?

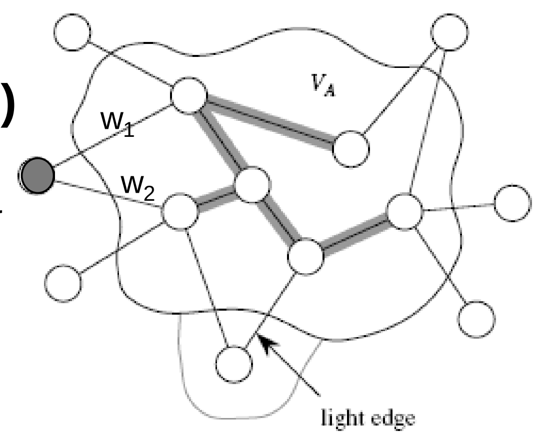
Use a priority queue Q:

- 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[a]=min(w_1,w_2)

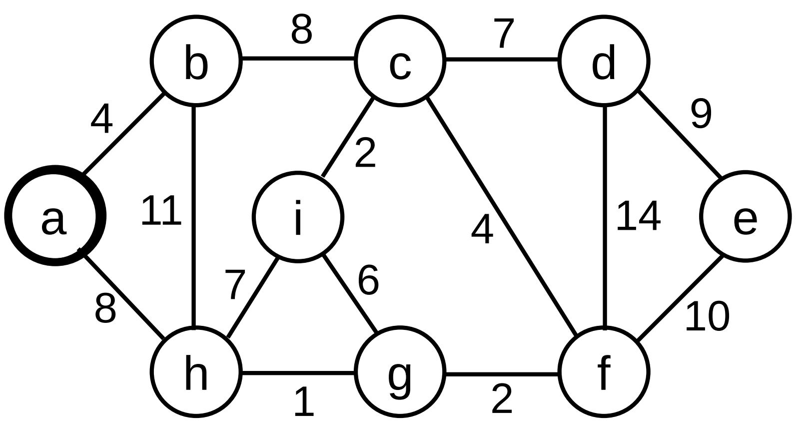


How to Find Light Edges Quickly? (cont.)

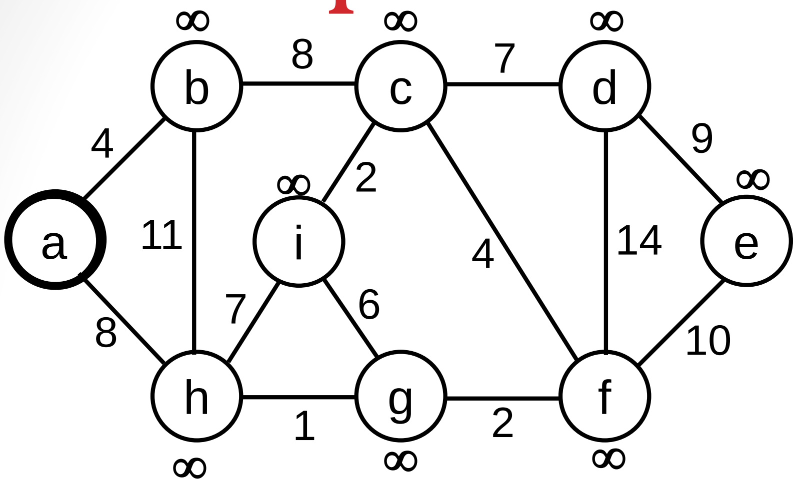
- After adding a new node to V_A we update the weights of all the nodes adjacent to it

  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

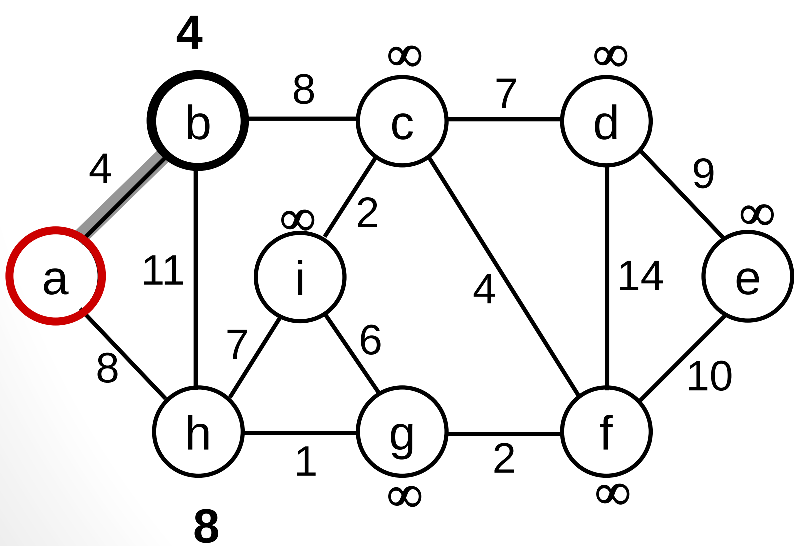


0 ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞

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

$V_A = \varnothing$

Extract-MIN(Q) $\Rightarrow$ a



key [b] = 4 π [b] = a

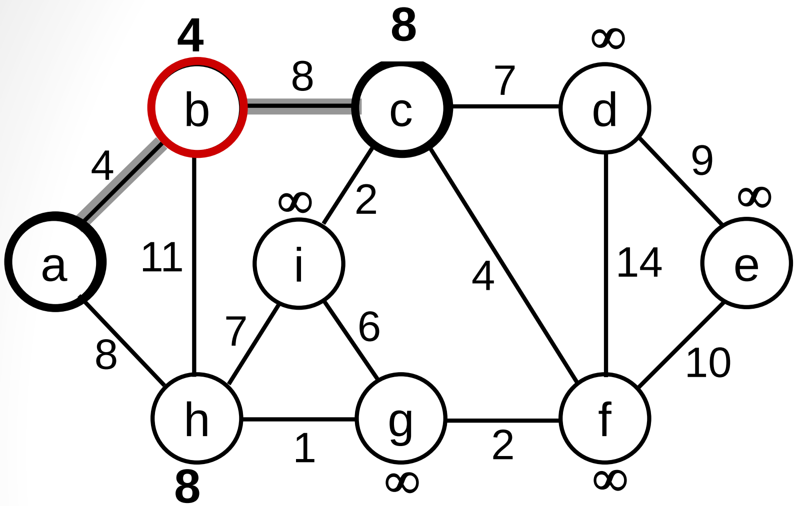
key [h] = 8 π [h] = a

4 ∞ ∞ ∞ ∞ ∞ **8** ∞

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

Extract-MIN(Q) $\Rightarrow$ b

Example



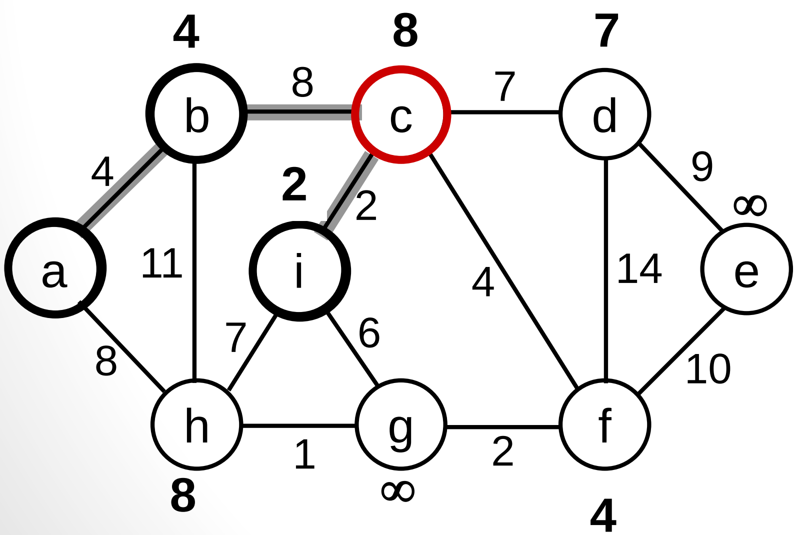
key [c] = 8 π [c] = b

key [h] = 11 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

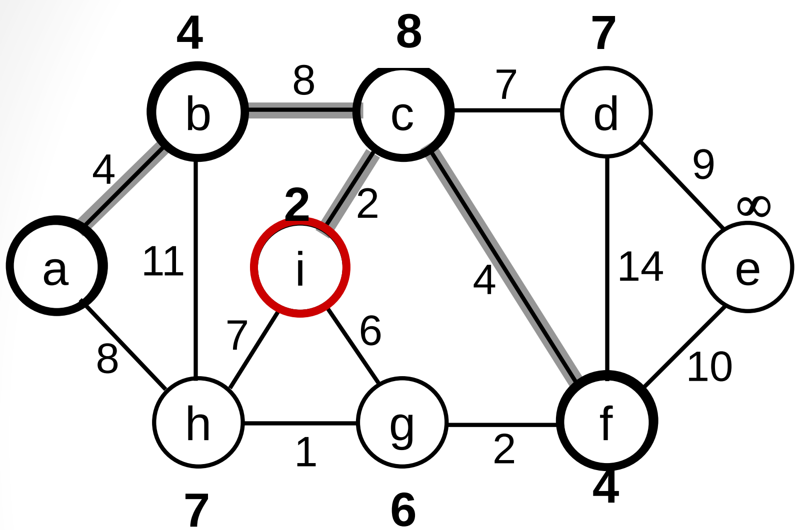
key [i] = 2 π [i] = c

7 ∞ 4 ∞ 8 2

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

Extract-MIN(Q) ⇒ i

Example



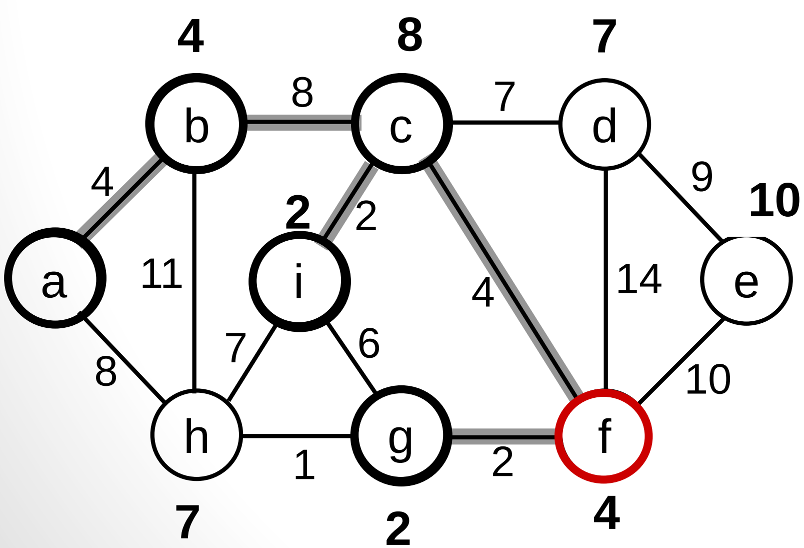
key [h] = 7   π [h] = i

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

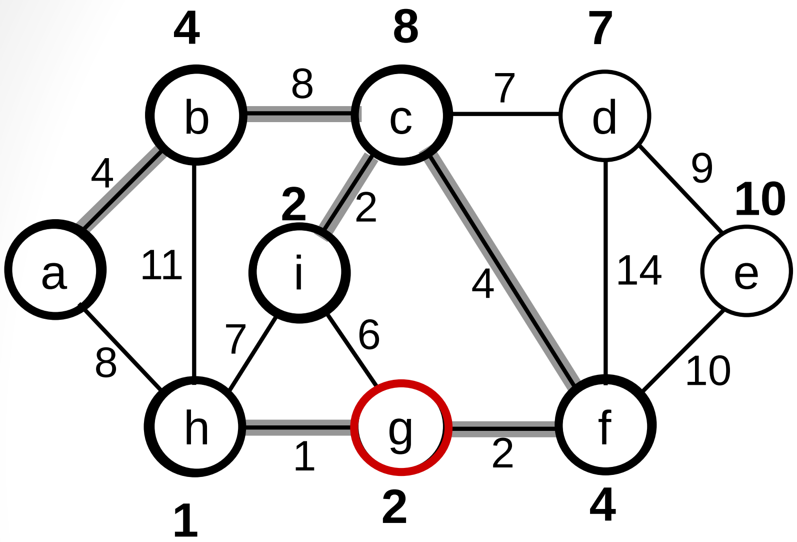
key [e] = 10   π [e] = f

7 10 2 7

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

Extract-MIN(Q) ⇒ g

Example

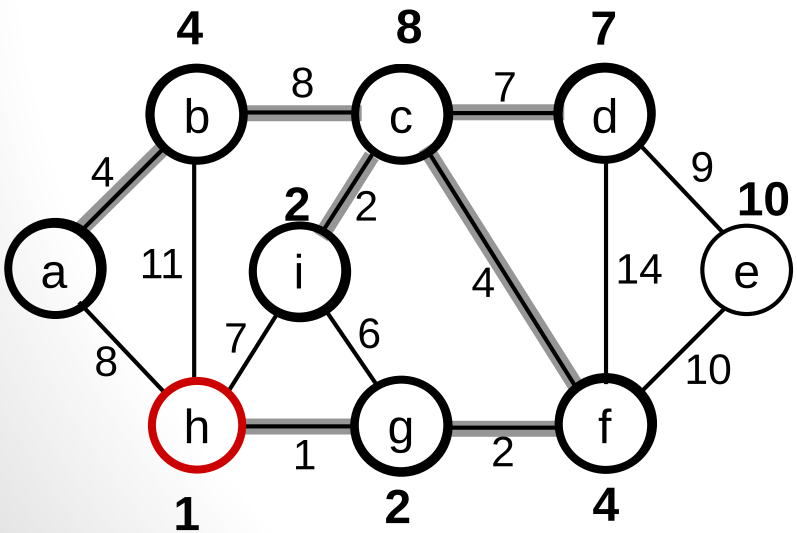


key [h] = 1 π [h] = g

7 10 1

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

Extract-MIN(Q) ⇒ h

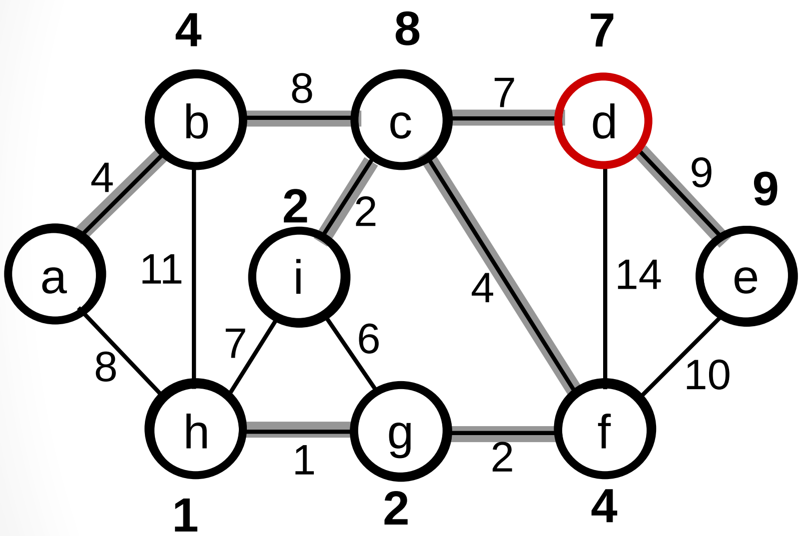


7 10

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))
```

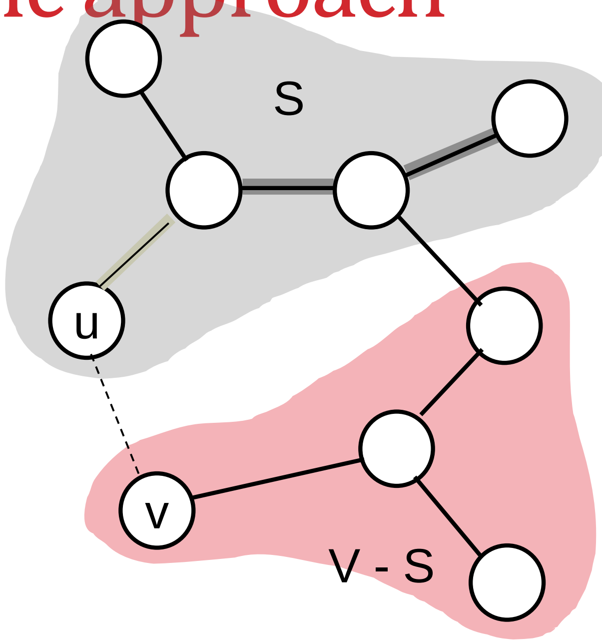
- Initialization (Q ← ∅ through INSERT(Q, u)): $O(V)$ if Q is implemented as a min-heap
- DECREASE-KEY(Q, r, 0): $O(\lg V)$
- while Q ≠ ∅: Executed |V| times
- u ← EXTRACT-MIN(Q): Takes $O(\lg V)$
- Min-heap operations: $O(V\lg V)$
- for each v ∈ Adj[u]: Executed O(E) times total
- if v ∈ Q and w(u, v) < key[v]: Constant
- DECREASE-KEY(Q, v, w(u, v)): Takes $O(\lg V)$
- Overall for the inner loop: $O(E\lg V)$

Prim's Algorithm

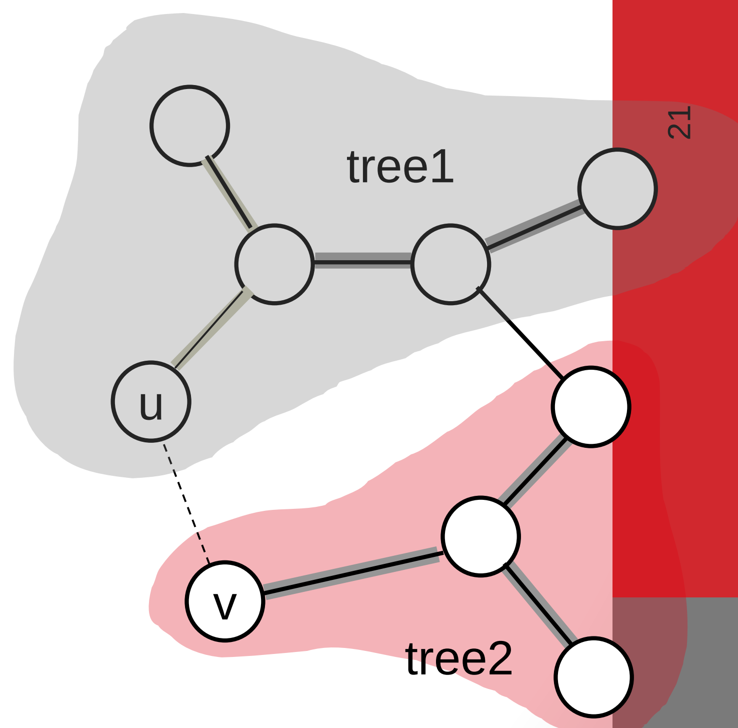
- Prim's algorithm is a **"greedy"** algorithm
  - Greedy algorithms find solutions based on a sequence of choices which are **"locally"** optimal at each step.
- Nevertheless, Prim's greedy strategy produces a globally optimum solution!

A different instance of the generic approach

(instance 1)



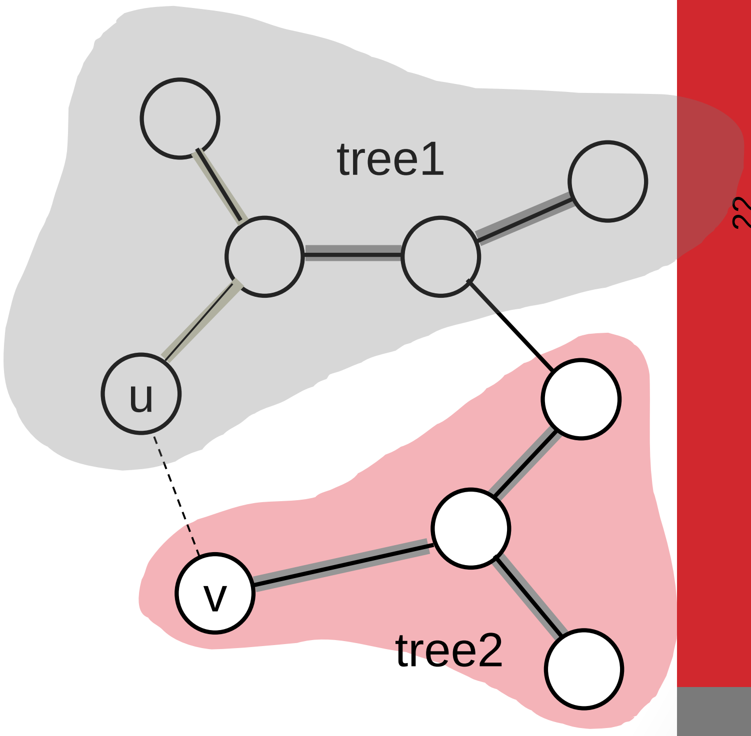
(instance 2)



- *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

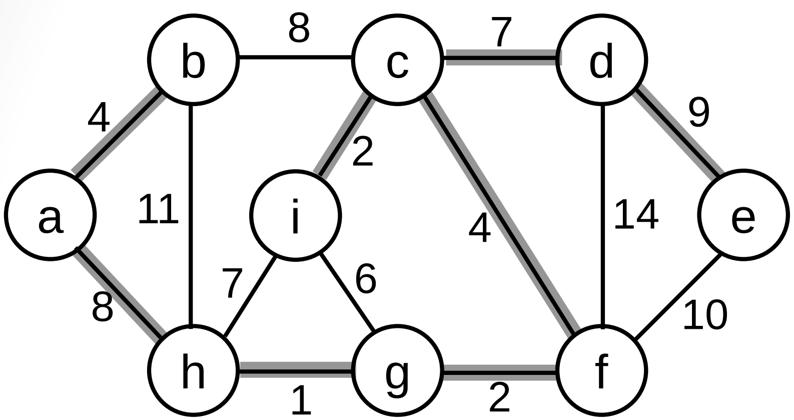
- How is it different from Prim's algorithm?
  - Prim's algorithm grows one tree all the time
  - Kruskal's algorithm grows multiple trees (i.e., a forest) at the same time.
  - Trees are merged together using **safe** edges
  - Since an MST has exactly |V| - 1 edges, after |V| - 1 merges, we would have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

Example



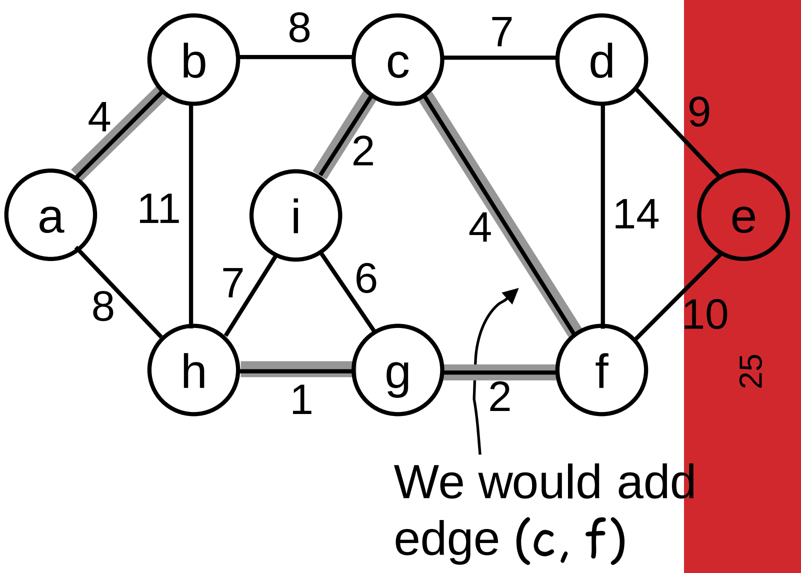
1: (h, g)
2: (c, i), (g, f)
4: (a, b), (c, f)
6: (i, g)
7: (c, d), (i, h)
8: (a, h), (b, c)
9: (d, e)
10: (e, f)
11: (b, h)
14: (d, f)

{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 (see **Chapter 21**) 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.
- Can be shown to be **α(n)=O(lgn)** where **α()** is a very slowly growing function (see **Chapter 21**)

KRUSKAL(V, E, w)

```
A ← ∅
for each vertex v ∈ V
        do MAKE-SET(v)
sort E into non-decreasing order by 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
```

O(V)

O(ElgE)

O(E)

O(lgV)

Running time: O(V+ElgE+ElgV)=O(ElgE) – dependent on the implementation of the disjoint-set data structure

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

```
A ← ∅
for each vertex v ∈ V
        do MAKE-SET(v)
sort E into non-decreasing order by 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
```

Lines 2–3: **O(V)**

Lines 4–5: **O(ElgE)**

Line 6: ← **O(E)**

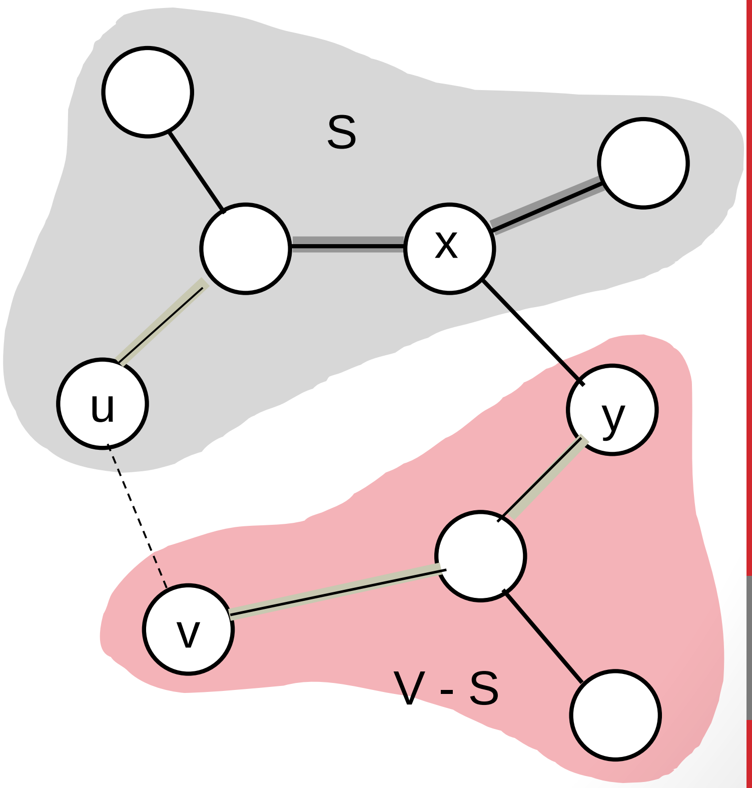
Lines 7–8: ← **O(lgV)**

O(ElgV)

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

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



Problem 1

- **(Exercise 23.2-3, page 573**) Compare Prim's algorithm with and Kruskal's algorithm assuming:

(a) sparse graphs:

In this case, E=O(V)

Kruskal:

O(ElgE)=O(VlgV)

Prim:

- binary heap: O(ElgV)=O(VlgV)
- Fibonacci heap: O(VlgV+E)=O(VlgV)

Problem 1 (cont.)

(b) dense graphs

In this case, $E=O(V^2)$

Kruskal:

$O(ElgE)=O(V^2lgV^2)=O(2V^2lgV)=O(V^2lgV)$

Prim:

- binary heap: $O(ElgV)=O(V^2lgV)$
- Fibonacci heap: $O(VlgV+E)=O(VlgV+V^2)=O(V^2)$

Problem 2

(Exercise 23.2-4, page 574): Analyze the running time of Kruskal's algorithm when weights are in the range **[1 ... V]**

Problem 2 (cont.)

```
A ← ∅
for each vertex v ∈ V
        do MAKE-SET(v)
sort E into non-decreasing order by 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
```

- Lines 2–3: O(V)
- Line 4: O(ElgE)
- O(E)
- O(lgV)

- Sorting can be done in O(E) time (e.g., using counting sort)
- However, overall running time will not change, i.e, O(ElgV)

Problem 3

- Suppose that some of the weights in a connected graph G are negative. Will Prim's algorithm still work? What about Kruskal's algorithm? Justify your answers.
  - Yes, both algorithms will work with negative weights. Review the proof of the generic approach; there is no assumption in the proof about the weights being positive.

Problem 4

- **(Exercise 23.2-2, page 573)** Analyze Prim's algorithm assuming:

(a) an adjacency-list representation of G

$O(ElgV)$

(b) an adjacency-matrix representation of G

$O(ElgV+V^2)$

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

- Lines 1–5 (Q ← ∅ … INSERT(Q, u)): $O(V)$ if Q is implemented as a min-heap
- DECREASE-KEY(Q, r, 0): $O(\lg V)$
- while Q ≠ ∅: Executed |V| times
- u ← EXTRACT-MIN(Q): Takes $O(\lg V)$
- Min-heap operations: $O(V\lg V)$
- for each v ∈ Adj[u]: Executed O(E) times
- if v ∈ Q and w(u, v) < key[v]: Constant
- DECREASE-KEY(Q, v, w(u, v)): Takes $O(\lg V)$
- Together: $O(E\lg V)$

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 (j=0; j<|V|; j++)
           if (A[u][j]=1)
              if v ∈ Q and w(u, v) < key[v]
                 then π[v] ← u
                      DECREASE-KEY(Q, v, w(u, v))
```

Total time: $O(V\lg V + E\lg V + V^2) = O(E\lg V + V^2)$

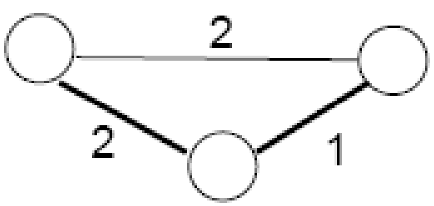
- Lines 1–5: $O(V)$ if Q is implemented as a min-heap
- DECREASE-KEY(Q, r, 0) ← $O(\lg V)$
- while Q ≠ ∅ ← Executed |V| times
- EXTRACT-MIN(Q) ← Takes $O(\lg V)$
  - Min-heap operations: $O(V\lg V)$
- for (j=0; j<|V|; j++) ← Executed $O(V^2)$ times total
- if (A[u][j]=1) ← Constant
- then π[v] ← u / DECREASE-KEY ← Takes $O(\lg V)$
  - $O(E\lg V)$

Problem 5

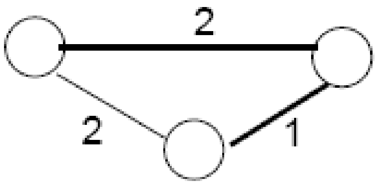
- Find an algorithm for the "maximum" spanning tree. That is, given an undirected weighted graph G, find a spanning tree of G of maximum cost. Prove the correctness of your algorithm.
  - Consider choosing the "heaviest" edge (i.e., the edge associated with the largest weight) in a cut. The generic proof can be modified easily to show that this approach will work.
  - Alternatively, multiply the weights by -1 and apply either Prim's or Kruskal's algorithms without any modification at all!

Problem 6

- **(Exercise 23.1-8, page 567)** Let T be a MST of a graph G, and let L be the sorted list of the edge weights of T. Show that for any other MST T' of G, the list L is also the sorted list of the edge weights of T'



T, L={1,2}



T', L={1,2}

- Special thanks to Dr. George Bebis, University of Nevada Reno