Graph Algorithms
COMP4500/7500
Advanced Algorithms & Data Structures

August 10, 2019

Overview

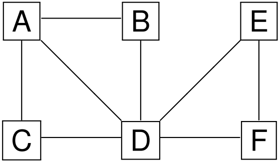
- Graphs recap
- Minimum spanning trees:
    - General
    - Prim's algorithm
    - Kruskal's algorithm

Graphs recap

Graphs are a common way of representing problems.

A graph $G = (V, E)$ is made up of:

- a set V of Vertices, e.g. (A, B, C, D, E, F)
- a set E of Edges, e.g. ((A,B), (A,D), (A,C), (B,D), ...)



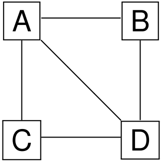
Can be:

- Directed/undirected
- Weighted/unweighted
- Cyclic/acyclic
- Connected/disconnected

Programming with graphs: graph representations

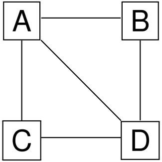
There are two main approaches to representing graphs:

- **Adjacency list.**
  E.g. for an undirected graph:



Node	Connections
A	B, D, C
B	A, D
C	A, D
D	A, B, C

- **Adjacency matrix.**
  E.g. for an undirected graph



	A	B	C	D
A	-	✓	✓	✓
B	✓	-	-	✓
C	✓	-	-	✓
D	✓	✓	✓	-

Algorithms covered

Covered:

- Breadth-first search
- Depth-first search
- Topological sort (DFS as a subroutine)

A minimum spanning tree problem

Consider laying cable (e.g. for the NBN):

- Create a **connected network of houses**
- that uses the **least amount of total cable.**

Assume that the speed along the cable is fast: the distance house-to-house is irrelevant (we aren't looking for shortest paths).

A minimum spanning tree problem ... in other words

We are given an undirected, weighted graph $G = (V, E)$ with weights w such that:

- vertices V represent houses in the NBN
- for each edge $(u, v) \in E$, the weight $w(u, v)$ is the cost of laying cable from house u to house v.

We want to find an acyclic subset $T \subseteq E$ that

- connects all of the vertices in G such that
- the total weight of T, i.e. $\sum_{(u,v)\in E} w(u, v)$, is minimised.

The minimum spanning tree problem

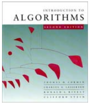
Inputs

- G a connected, undirected, weighted graph $G = (V, E)$
- w weights w, where $w(u, v)$ is the weight of the edge from u to v

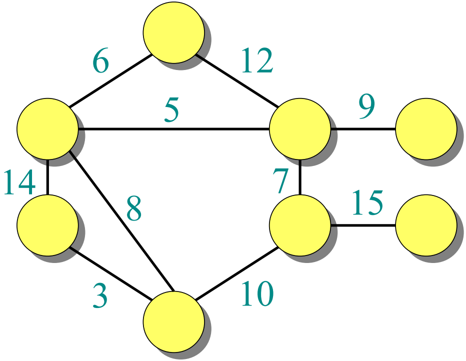
Output

- T A subset of E that forms a **spanning tree** T:
  a tree (connected acyclic subgraph of G) that
  contains all vertices of G (spanning)
  and is of **minimal weight**

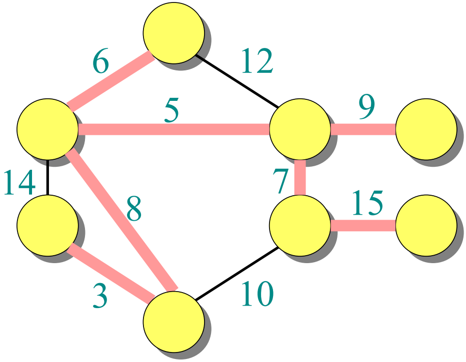
$$weight(T) = \sum_{(u,v) \in T} w(u, v)$$



Example of MST



Example of MST



Generic construction of an MST

Approach: incrementally construct T, which is a set of edges, and which will eventually become an MST of G.

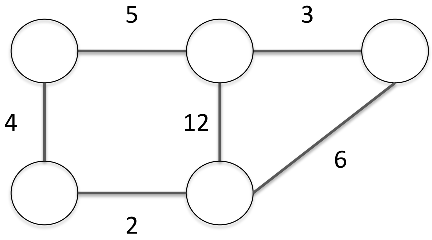
GENERIC-MST(G, w)

```
T = ∅
while T is not a spanning tree
        // invariant: T is a subset of some MST of G
        find an edge (u, v) that is safe for T
        T = T ∪ {(u, v)}
return T
```

Prim's algorithm

T **is always a tree** (a connected acyclic sub-graph of *G*):

- Initially *T* is chosen to contain any one vertex from $G.V$.
- At each step, **the least-weight edge leaving** *T* **is added.**
- The algorithm stops when *T* is **spanning**.



Prim's algorithm

How do we (efficiently) find the least-weight edge leaving T?

- Maintain a **priority queue** Q containing vertices $V - T$.
- For each $v \in V - T$:
  - $\boldsymbol{v.key}$: least weight of an edge connecting v to T
  - $\boldsymbol{v.\pi}$: the vertex adjacent to v on that least-weight edge

Represent

$$T = \{(v, v.\pi) : v \in V - \{r\} - Q\}$$

where r is the first vertex chosen for T.

Priority Queue

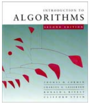
A **priority queue** Q maintains a set S of elements, each associated with a **key**, denoting its priority.

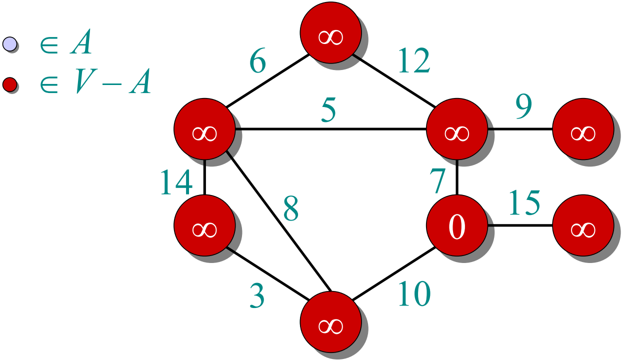
- In a **min-priority queue** an element with the smallest key has the highest priority.
- Operations are available to:
  - **insert(Q,x)**
    inserts an element x with key $x.key$ into Q
  - **extract-min(Q)**
    removes and returns the element of Q with the smallest key
  - **decrease-key(Q,x,k)**
    decreases the key of x in Q to the value k.

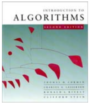
Prim's algorithm

```
MST-PRIM(G, w, r)
for each u ∈ G.V
        u.key = ∞
        u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
        // invariant: T is a subset of some MST of G
        //                 where T = {(v, v.π) : v ∈ V−{r}−Q}
        u = EXTRACT-MIN(Q)
        for each v ∈ G.Adj[u]
                if v ∈ Q and w(u, v) < v.key
                        v.π = u
                        v.key = w(u, v)   // Decrease key
```

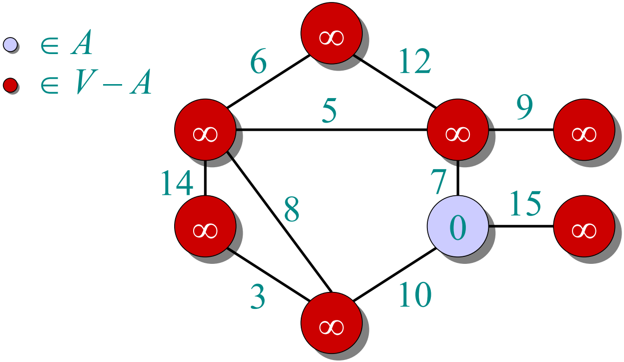
Example of Prim's algorithm



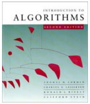


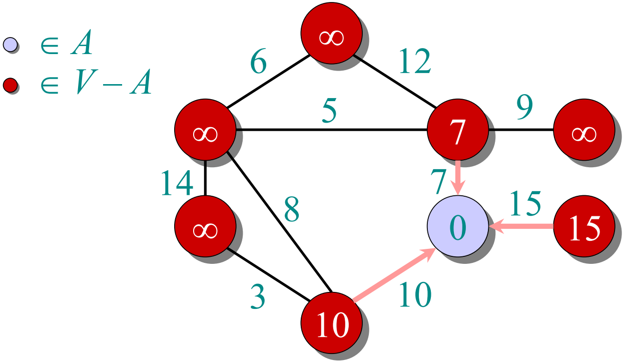


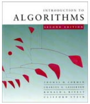
Example of Prim's algorithm



Example of Prim's algorithm



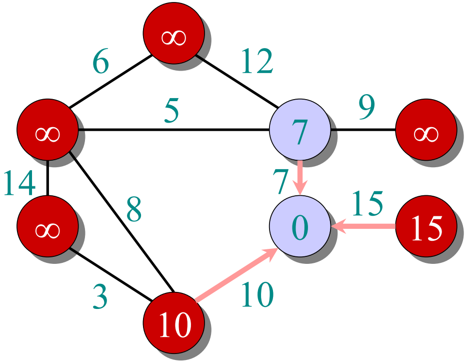


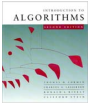


Example of Prim's algorithm

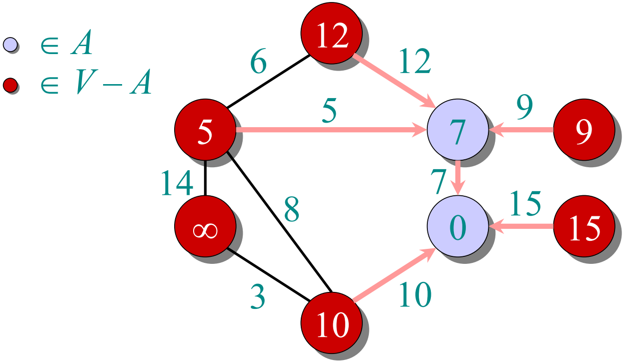
$\circ \in A$

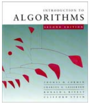
$\bullet \in V - A$



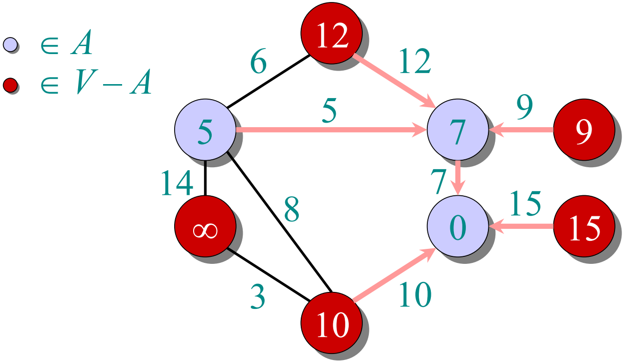


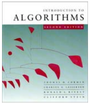
Example of Prim's algorithm



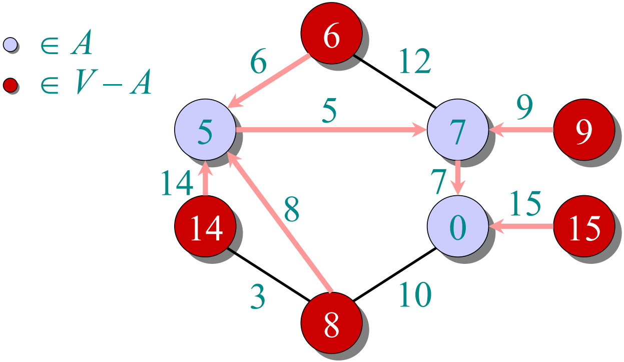


Example of Prim's algorithm

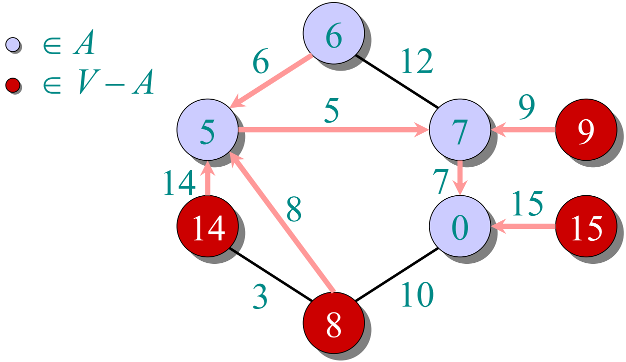


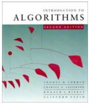


Example of Prim's algorithm

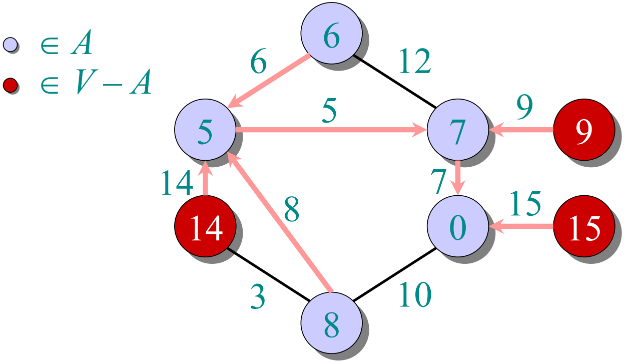


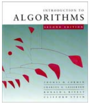
Example of Prim's algorithm





Example of Prim's algorithm

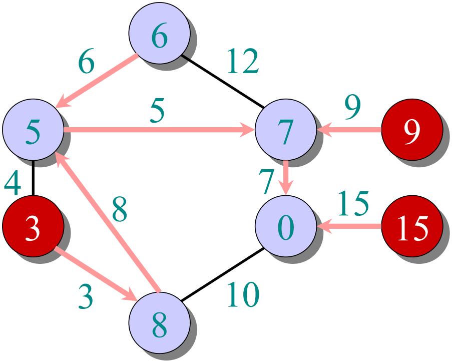


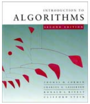


Example of Prim's algorithm

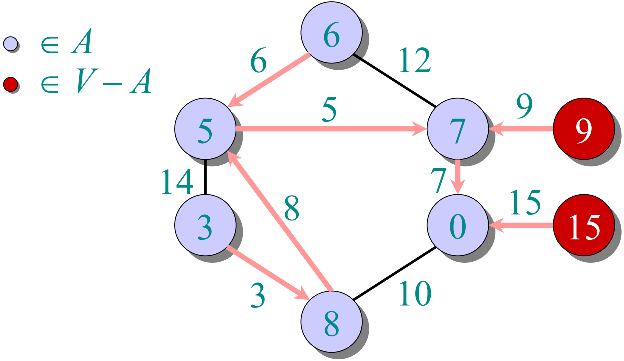
● $\in A$

● $\in V - A$

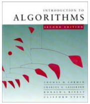


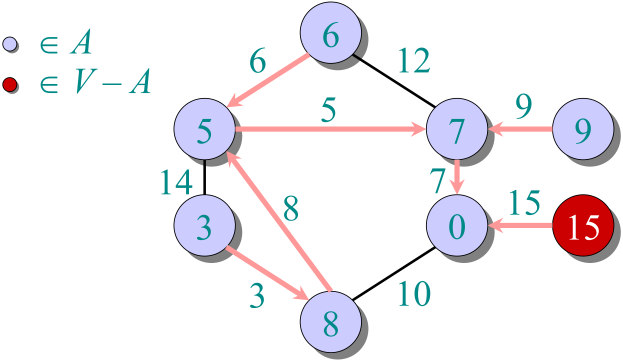


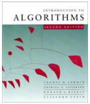
Example of Prim's algorithm



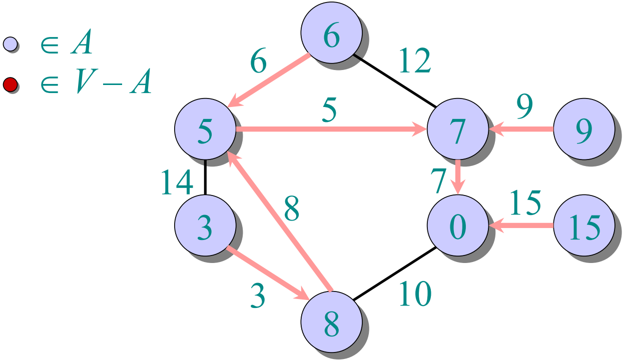
Example of Prim's algorithm

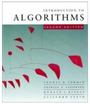






Example of Prim's algorithm





Analysis of Prim

$Q \leftarrow V$

$key[v] \leftarrow \infty$ for all $v \in V$

$key[s] \leftarrow 0$ for some arbitrary $s \in V$

while $Q \neq \emptyset$

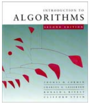
  do $u \leftarrow$ EXTRACT-MIN(Q)

    for each $v \in Adj[u]$

      do if $v \in Q$ and $w(u, v) < key[v]$

        then $key[v] \leftarrow w(u, v)$

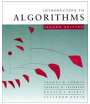
          $\pi[v] \leftarrow u$



Analysis of Prim

$$
\left.\begin{array}{c}\Theta(V)\\ \text{total}\end{array}\right\{
\begin{array}{l}
Q \leftarrow V\\
key[v] \leftarrow \infty \text{ for all } v \in V\\
key[s] \leftarrow 0 \text{ for some arbitrary } s \in V
\end{array}
$$

while $Q \neq \emptyset$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    for each $v \in Adj[u]$
      do if $v \in Q$ and $w(u, v) < key[v]$
        then $key[v] \leftarrow w(u, v)$
          $\pi[v] \leftarrow u$



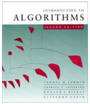
Analysis of Prim

$\Theta(V)$ total:

$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$

$|V|$ times:

while $Q \neq \emptyset$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    for each $v \in Adj[u]$
      do if $v \in Q$ and $w(u, v) < key[v]$
        then $key[v] \leftarrow w(u, v)$
          $\pi[v] \leftarrow u$



Analysis of Prim

$\Theta(V)$ total

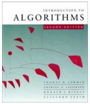
$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$

$|V|$ times

while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN(Q)

$degree(u)$ times

        for each $v \in Adj[u]$
            do if $v \in Q$ and $w(u, v) < key[v]$
                then $key[v] \leftarrow w(u, v)$
                    $\pi[v] \leftarrow u$



Analysis of Prim

$\Theta(V)$ total

$Q \leftarrow V$

$key[v] \leftarrow \infty$ for all $v \in V$

$key[s] \leftarrow 0$ for some arbitrary $s \in V$

$|V|$ times

while $Q \neq \emptyset$

do $u \leftarrow$ EXTRACT-MIN(Q)

$degree(u)$ times

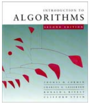
for each $v \in Adj[u]$

do if $v \in Q$ and $w(u, v) < key[v]$

then $key[v] \leftarrow w(u, v)$

$\pi[v] \leftarrow u$

Handshaking Lemma $\Rightarrow \Theta(E)$ implicit DECREASE-KEY's.



Analysis of Prim

$\Theta(V)$ total:

- $Q \leftarrow V$
- $key[v] \leftarrow \infty$ for all $v \in V$
- $key[s] \leftarrow 0$ for some arbitrary $s \in V$

$|V|$ times:

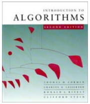
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN(Q)

$degree(u)$ times:

        for each $v \in Adj[u]$
            do if $v \in Q$ and $w(u, v) < key[v]$
                then $key[v] \leftarrow w(u, v)$
                    $\pi[v] \leftarrow u$

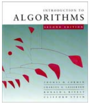
Handshaking Lemma $\Rightarrow \Theta(E)$ implicit DECREASE-KEY's.

Time $= \Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$



Analysis of Prim (continued)

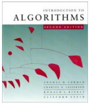
Time = $\Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$



Analysis of Prim (continued)

Time = $\Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$

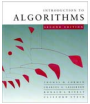
Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total



Analysis of Prim (continued)

Time = $\Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

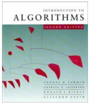
Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$



Analysis of Prim (continued)

Time = $\Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$



Analysis of Prim (continued)

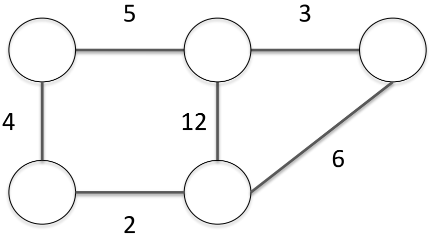
Time = $\Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$ amortized	$O(1)$ amortized	$O(E + V \lg V)$ worst case

Kruskal’s approach

T is always a spanning acyclic subgraph(a **forest of trees**):

- Initially T contains all vertices $G.V$, but no edges.
- At each step, the least-weight edge that connects any two trees in the forest T is added to T.
- The algorithm stops when T is connected.



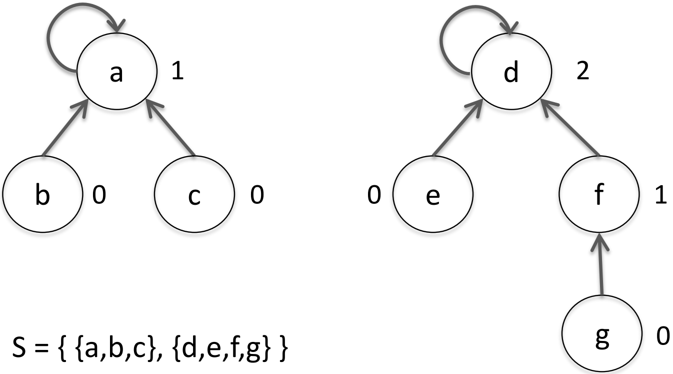
Disjoint sets

The trees in T form disjoint sets of $G.V$.

- A **disjoint-set data structure** maintains a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets.
- Each set is identified by a **representative** element from that set.
- Operations are available to:
  - **Make-Set(x)**:
    adds a new set with element x.
    Requires that x is not already a member of another set.
  - **Find-Set(x)**:
    returns the representative element for the set containing x.
  - **Union(x,y)**:
    merge the set that contain x with the set that contains y.
    (Uses: **Link(x,y)** sub-routine.)

Disjoint set implementation: disjoint-set forests

- Sets are represented by rooted trees.
- The root of each tree is its representative element.
- Each element x stores:
  - $x.p$: the parent of x in its tree (or itself if it is the root).
  - $x.rank$: an upper bound on the height of x in its tree.



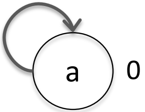
S = { {a,b,c}, {d,e,f,g} }

Disjoint set forests: Make-set(x)

MAKE-SET(x)

$x.p = x$
$x.rank = 0$

E.g. Make-set(a) makes a set of size 1, containing only a:



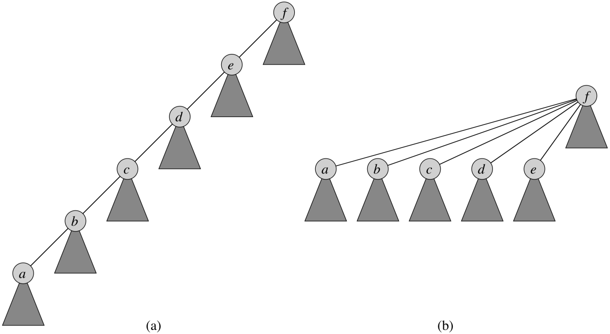
Disjoint set forests: Find-set(x)

```
FIND-SET(x)
if x ≠ x.p
    x.p = FIND-SET(x.p)
return x.p
```

- Find-set(x) returns the *top node* in the set
  - The top node in the set is the "identifier" for that set
  - It is the only node whose parent is itself.
- It applies a **path compression heuristic.**

Disjoint set forests: path compression heuristic

As it traverses the parent links, it collapses them, making them point directly to the top node.



(a) (b)

Disjoint set forests: Union(x,y)

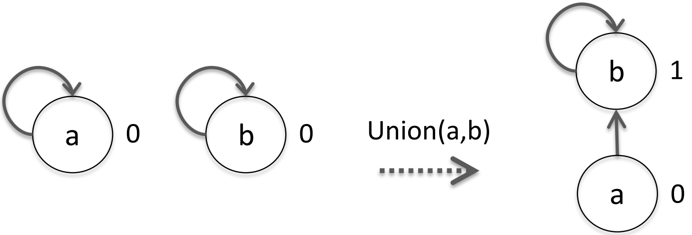
```
UNION(x, y)
  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
  if x.rank > y.rank
      y.p = x
  else
      x.p = y
      // If equal rank, choose y as parent and increment its rank
      if x.rank == y.rank
          y.rank = y.rank + 1
```

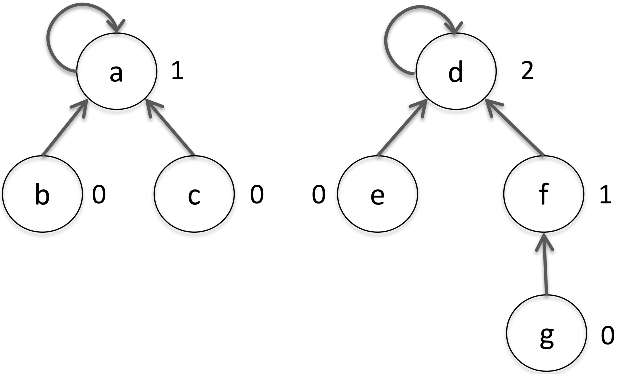
It applies a **union by rank** heuristic:

- the tree with fewer nodes is made to point to the tree with more nodes.

Disjoint set forests: Union(x,y)

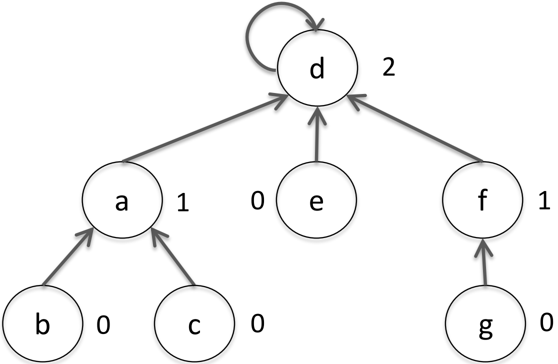


Disjoint set forests: Union(x,y)



Union(a,d)

Disjoint set forests: Union(x,y)



Union(a,d)

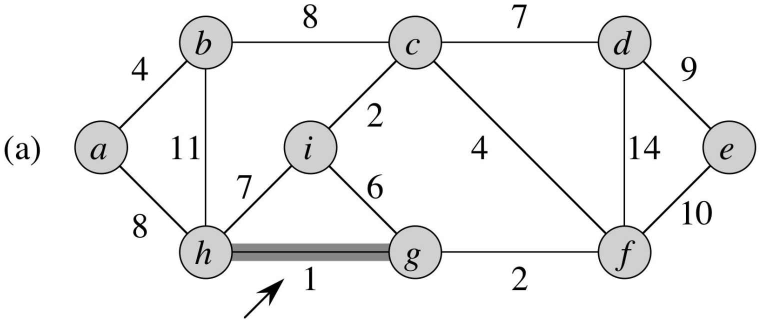
Kruskal's approach

A set (of edges) T represents a tree

MST-KRUSKAL(G, w)

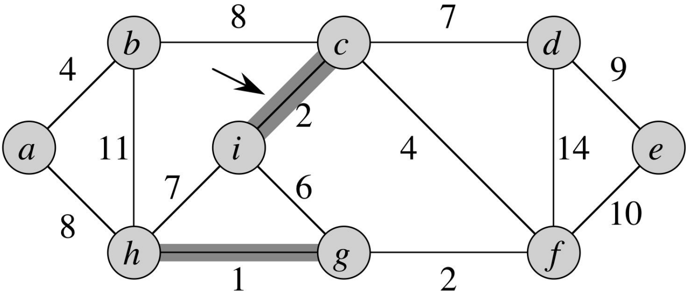
$T = \emptyset$
for each vertex $v \in G.V$
    MAKE-SET(v)
sort the edges of $G.E$ into non-decreasing order by weight w
for each (u, v) taken from the sorted list
    if FIND-SET$(u) \neq$ FIND-SET(v)
        $T = T \cup \{(u, v)\}$
        UNION(u, v)
return T

Kruskal's MST algorithm walkthrough (CLRS Fig 23.4)



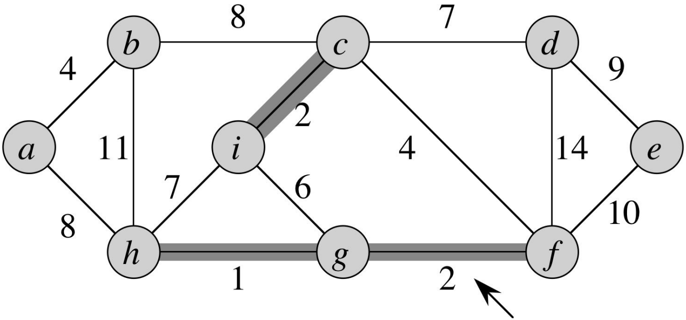
Kruskal's MST algorithm walkthrough (CLRS Fig 23.4)

(b)

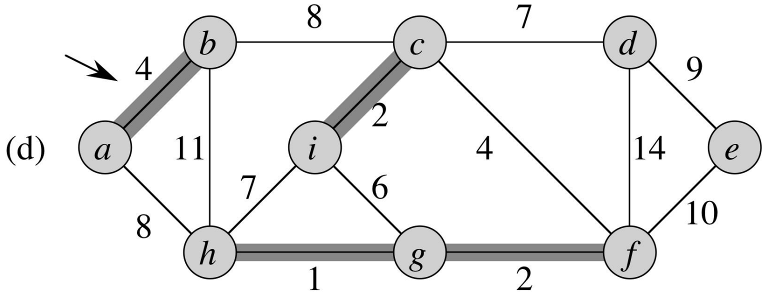


Kruskal's MST algorithm walkthrough (CLRS Fig 23.4)

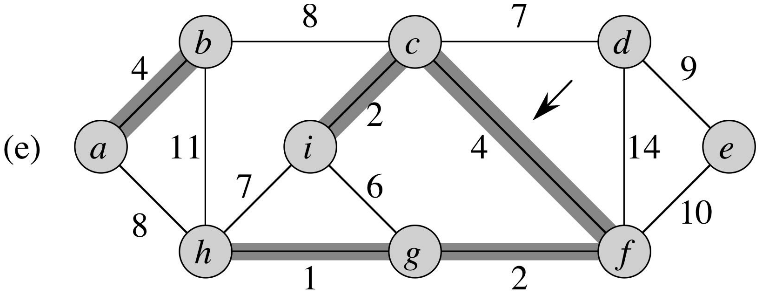
(c)



Kruskal's MST algorithm walkthrough (CLRS Fig 23.4)

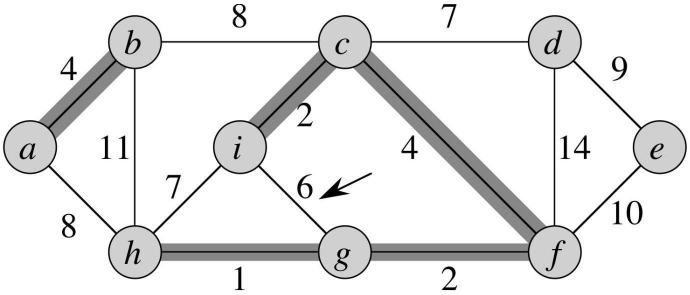


Kruskal's MST algorithm walkthrough (CLRS Fig 23.4)



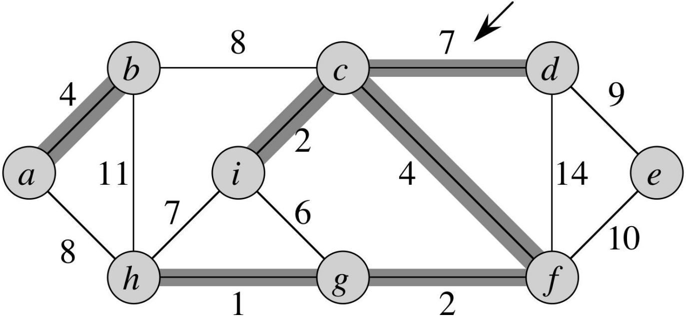
Kruskal's MST algorithm walkthrough (CLRS Fig 23.4)

(f)

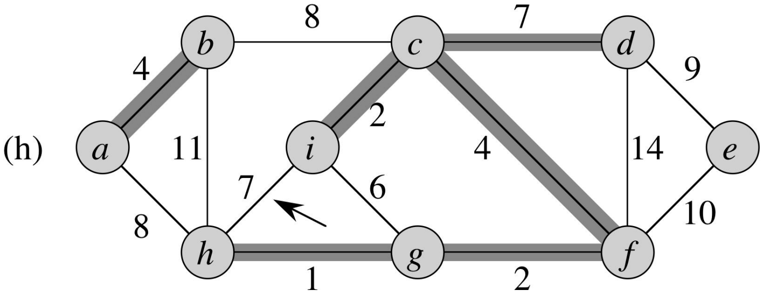


Kruskal's MST algorithm walkthrough (CLRS Fig 23.4)

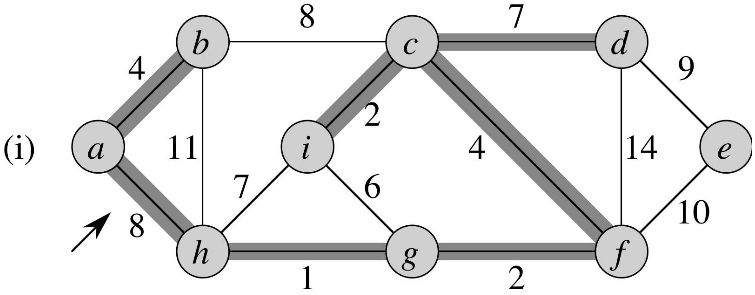
(g)



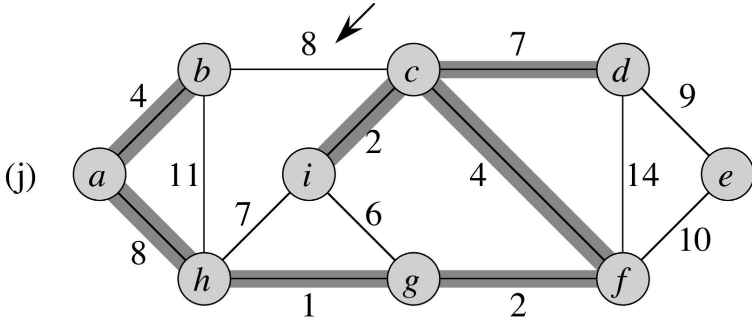
Kruskal's MST algorithm walkthrough (CLRS Fig 23.4)



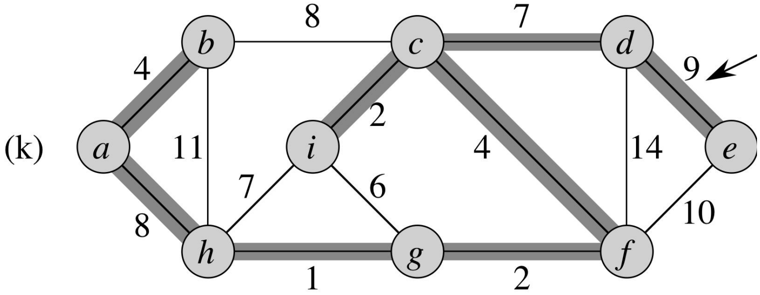
Kruskal's MST algorithm walkthrough (CLRS Fig 23.4)



Kruskal's MST algorithm walkthrough (CLRS Fig 23.4)

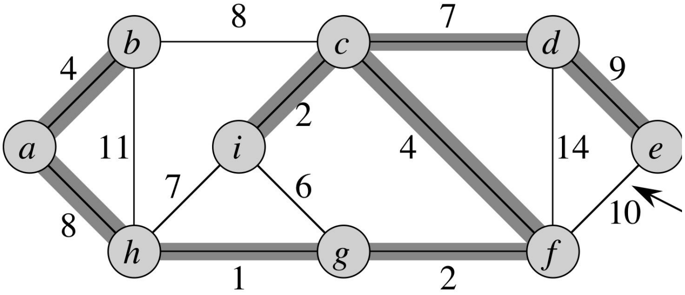


Kruskal’s MST algorithm walkthrough (CLRS Fig 23.4)



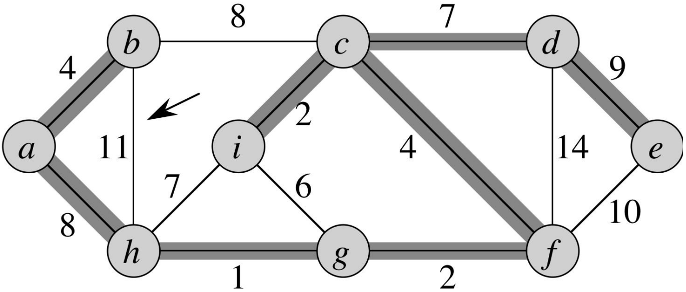
Kruskal's MST algorithm walkthrough (CLRS Fig 23.4)

(l)



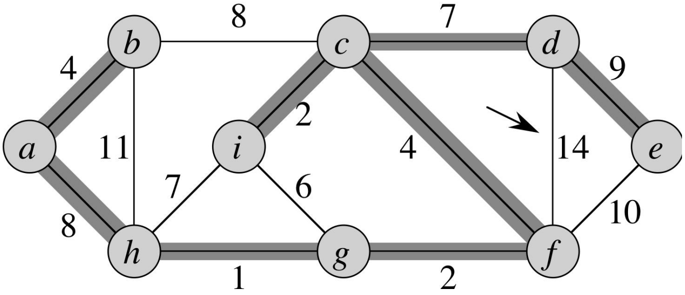
Kruskal's MST algorithm walkthrough (CLRS Fig 23.4)

(m)



Kruskal's MST algorithm walkthrough (CLRS Fig 23.4)

(n)



Analysis of Kruskal

```
MST-KRUSKAL(G, w)
T = ∅
for each vertex v ∈ G.V
      MAKE-SET(v)
sort the edges of G.E into non-decreasing order by weight w
for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
            T = T ∪ {(u, v)}
            UNION(u, v)
return T
```

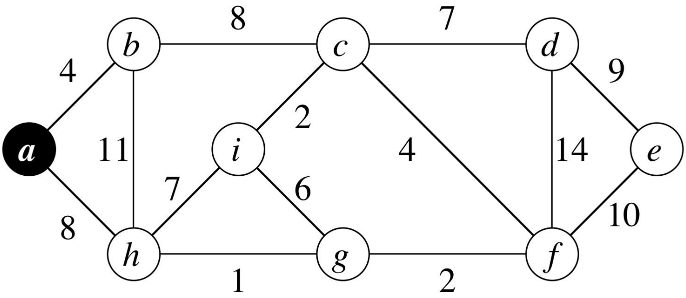
1. MAKE-SET is constant time. Hence the first loop is $\Theta(V)$.
2. Sorting the edges is $\Theta(E \lg E)$.
3. The main loop is executed once per edge ($|E|$ times). There are four calls to FIND-SET. CLRS contains a sophisticated argument that this is $O(\lg E)$.

Hence Kruskal's algorithm using the disjoint-set forest implementation is $\Theta(E \lg E)$.

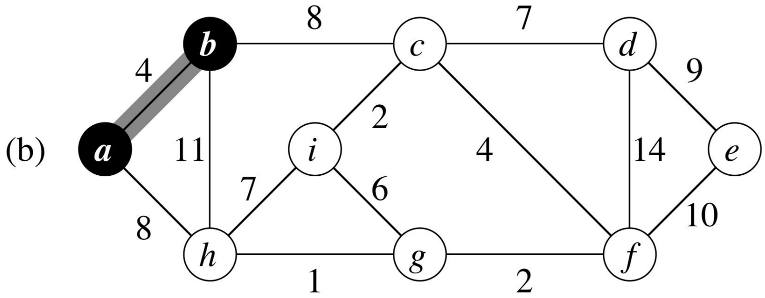
Prim’s MST algorithm walkthrough (CLRS Figure 23.5)

Prim's MST algorithm walkthrough (CLRS Figure 23.5)

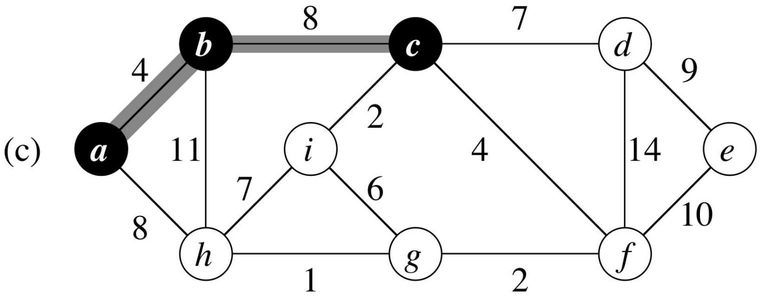
(a)



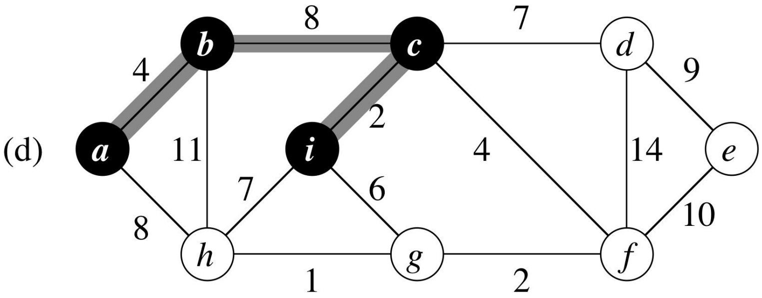
Prim's MST algorithm walkthrough (CLRS Figure 23.5)



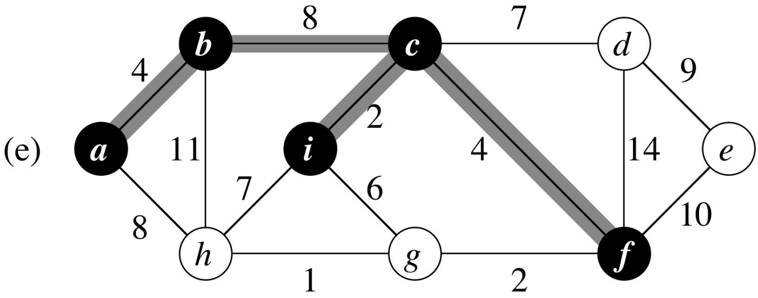
Prim's MST algorithm walkthrough (CLRS Figure 23.5)



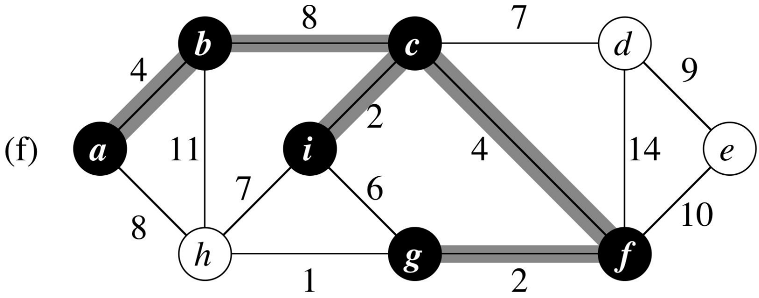
Prim's MST algorithm walkthrough (CLRS Figure 23.5)



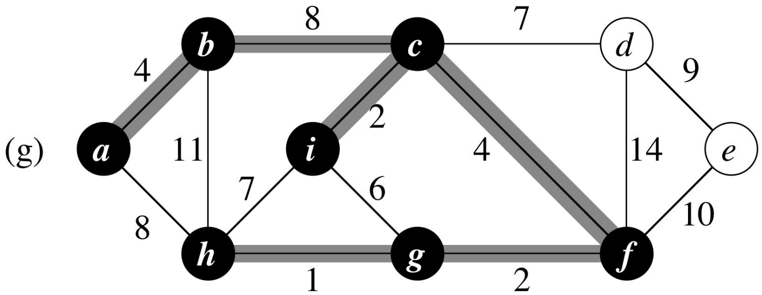
Prim's MST algorithm walkthrough (CLRS Figure 23.5)



Prim's MST algorithm walkthrough (CLRS Figure 23.5)

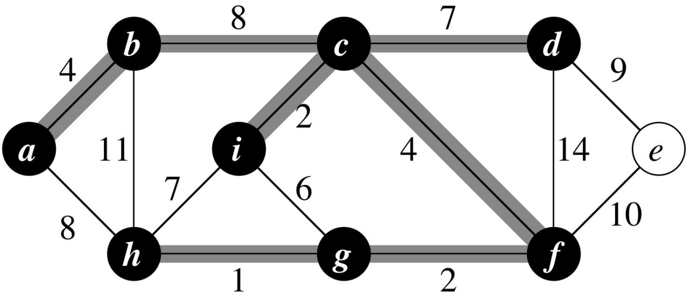


Prim's MST algorithm walkthrough (CLRS Figure 23.5)

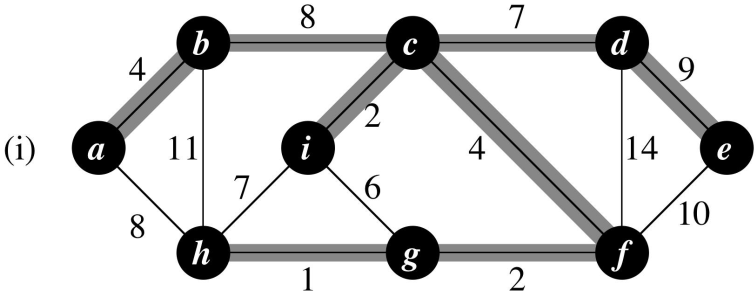


Prim's MST algorithm walkthrough (CLRS Figure 23.5)

(h)



Prim's MST algorithm walkthrough (CLRS Figure 23.5)



Implementing MST algorithms

Data structures for incrementally building/maintaining the MST

Prim: a priority queue implemented using a heap

Kruskal: a disjoint set implemented using a disjoint-set forest

MST recap

- An MST gives the shortest total physical distance required to connect every node
- Kruskal and Prim are two different approaches to constructing an MST
- Kruskal's algorithm requires an implementation of disjoint sets;
  Prim's algorithm requires an implementation of a priority queue

Recap of this week

1. Minimum spanning trees
    - Kruskal's algorithm,
        - $O(E \lg E)$ using union-find trees
    - Prim's algorithm,
        - $O(E \lg V)$ using binary heap or
        - $O(E + V \lg V)$ using a Fibonacci heap