EL9343

Data Structure and Algorithm

Lecture 11: Greedy Algorithm, Minimum Spanning Tree

Instructor: Yong Liu

Last Lecture

- Dynamic Programming
  - Rod Cutting Problem
  - Longest Common Subsequence Problem

- Introduction to Greedy Algorithm
  - An Activity-Selection Problem
  - Knapsack Problem

Today

- Greedy Algorithm (cont.)
  - Huffman codes

- Algorithm & It's Application in Network
  - Minimum Spanning Trees
    - Prim's algorithm
    - Kruskal's algorithm

Huffman Coding

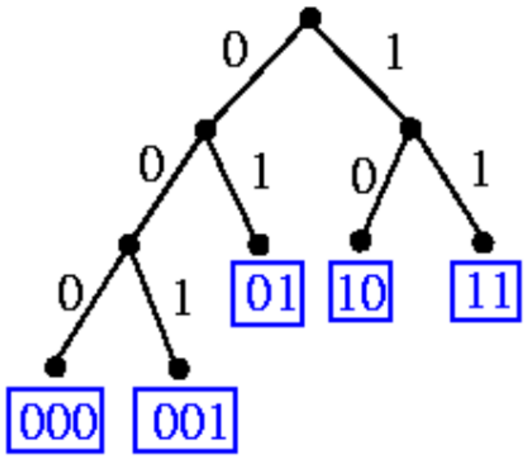
- **Coding** is used for data compression
- **Binary character code:** character is represented by a unique binary string
  - Fixed-length code (block code)
    - a: 000, b: 001, …, f: 101
    - ace: 000 010 100
  - Variable-length code
    - frequent characters: short codeword
    - infrequent characters: long codeword

	a	b	c	d	e	f	cost / 100 characters
Frequency	45	13	12	16	9	5	
Fixed–length codeword	000	001	010	011	100	101	300
Variable–length codeword	0	101	100	111	1101	1100	224

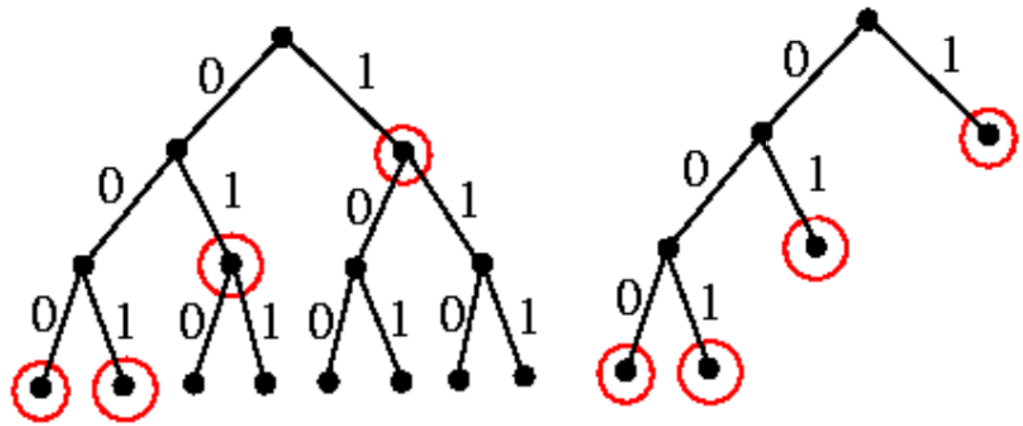
Prefix codes

- Prefix codes
  - one code per input symbol
  - no code is a prefix of another
- Why prefix codes?
  - Easy decoding
  - Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous
  - Identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file

Binary Tree vs. Prefix Code

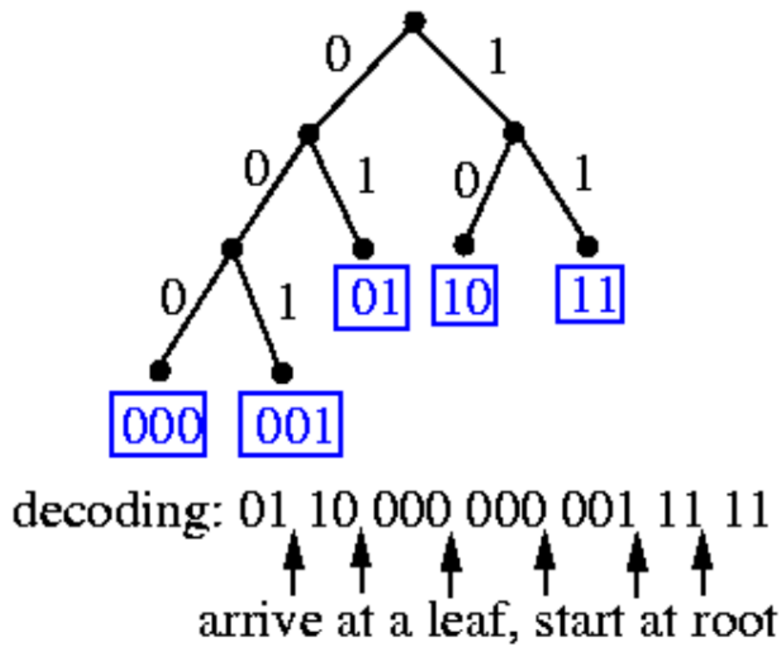


binary tree –> prefix code



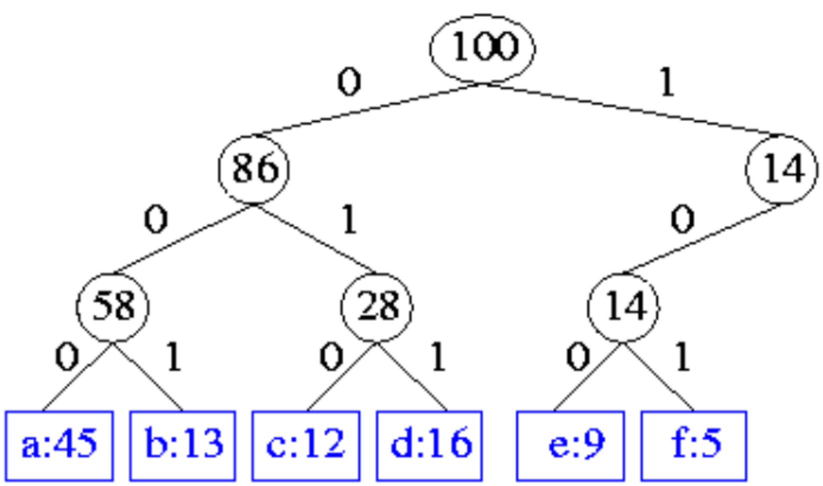
prefix code {1, 01, 000, 001}–> binary tree

- Any binary prefix coding can be described by a binary tree in which the codewords are the leaves of the tree, and where a left branch means “0” and a right branch means “1”.

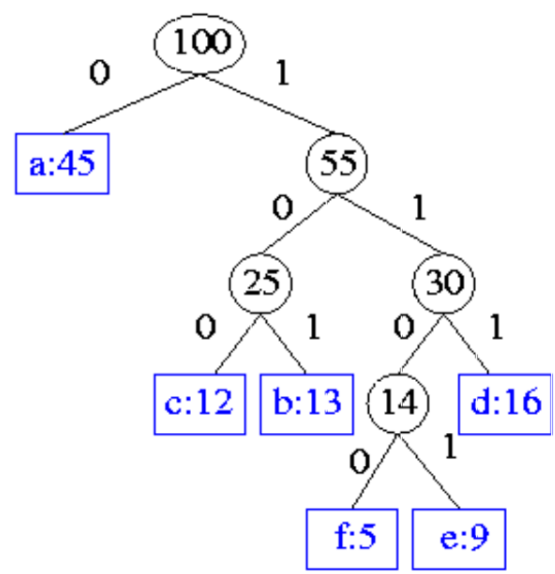


Optimal Prefix Code Design

- Coding Cost of T: $B(T)=\sum_{c \in C} c.freq \cdot d_T(c)$
  - c: character in the alphabet C
  - c.freq: frequency of c
  - $d_T(c)$: depth of c's leaf (length of the codeword of c)
- Code design: Given c_1.freq, c_2.freq, …, c_n.freq, construct a binary tree with n leaves such that B(T) is minimized.
  - Idea: more frequently used characters use shorter depth.



Fixed–length cost: 3 * 100 = 300



Variable–length cost = 224

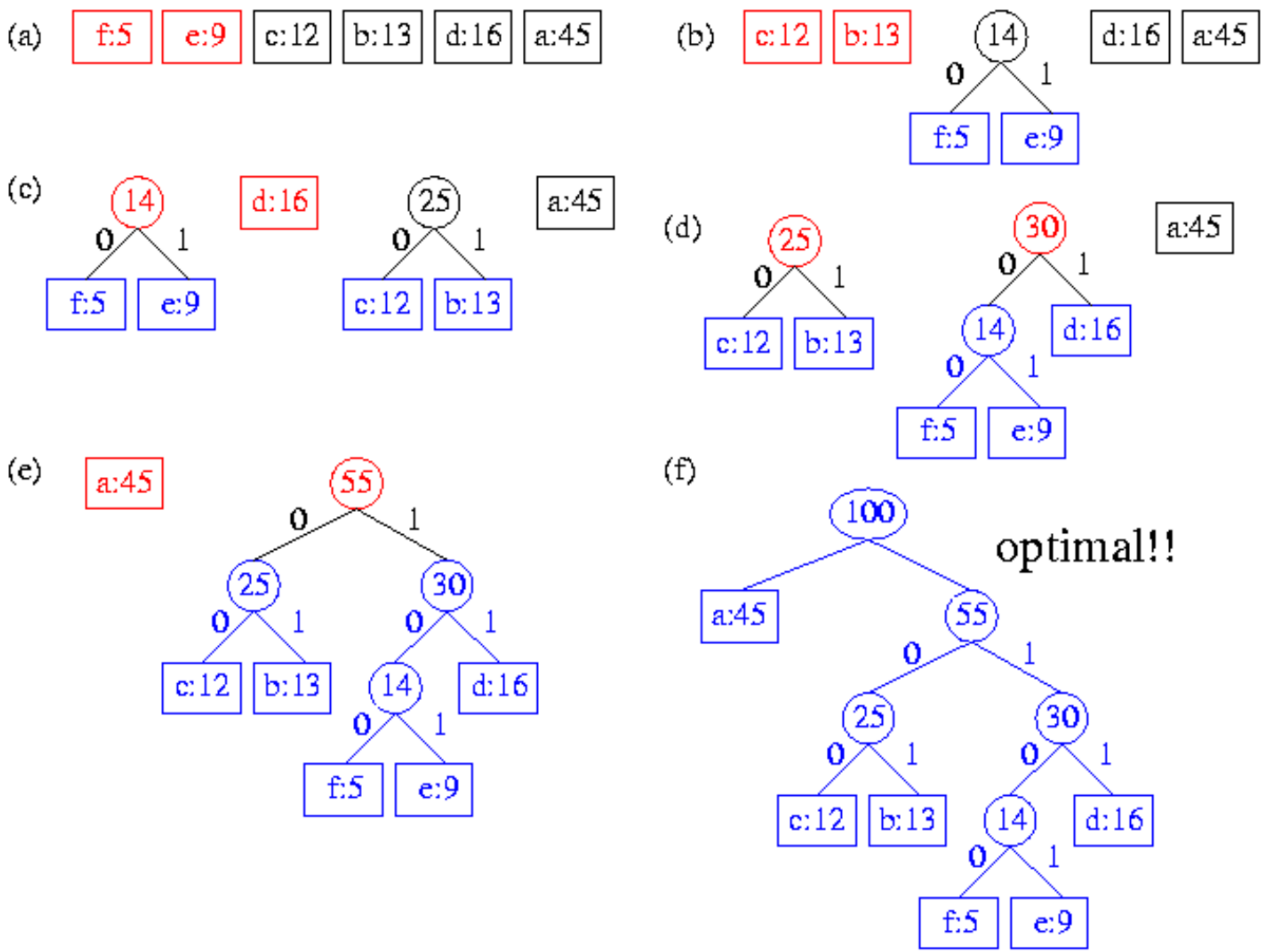
Huffman Codes

Huffman invented a greedy algorithm that constructs an optimal prefix code called a Huffman code. The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner.

- Create tree (leaf) node for each symbol that occurs with nonzero frequency
  - Node weights = frequencies
- Find two nodes with smallest frequency
- Create a new node with these two nodes as children, and with weight equal to the sum of the weights of the two children
- Continue until have a single tree

Huffman's Procedure

- 1. Place the elements into minimum heap (by frequency).
- 2. Remove the first two elements from the heap.
- 3. Combine these two elements into one.
- 4. Insert the new element back into the heap.



(a)
f:5
e:9
c:12
b:13
d:16
a:45
(b)
c:12
b:13
14
0
1
f:5
e:9
d:16
a:45
(c)
14
0
1
f:5
e:9
d:16
25
0
1
c:12
b:13
a:45
(d)
25
0
1
c:12
b:13
30
0
1
14
d:16
0
1
f:5
e:9
a:45
(e)
a:45
55
0
1
25
30
0
1
c:12
b:13
0
1
14
d:16
0
1
f:5
e:9
(f)
100
optimal!!
a:45
55
0
1
25
30
0
1
c:12
b:13
0
1
14
d:16
0
1
f:5
e:9

Huffman's Algorithm

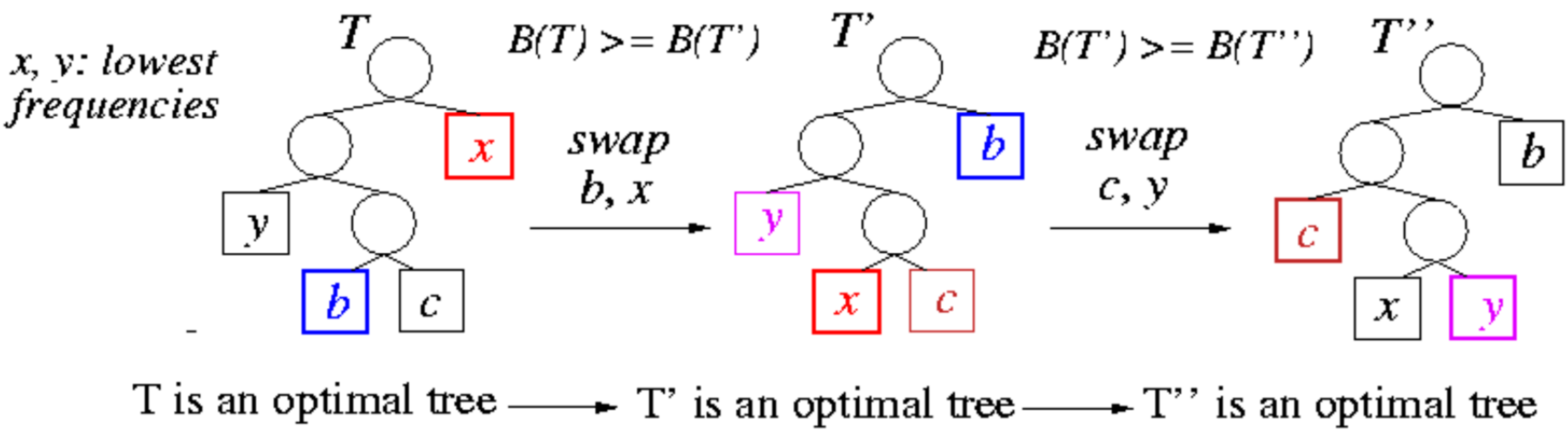
```
Huffman(C)
1. n = |C|
2. Q = C
3. for i = 1 to n -1
4.      Allocate a new node z
5.      z.left = x = Extract-Min(Q)
6.      z.right = y = Extract-Min(Q)
7.      z.freq = x.freq + y.freq
8.      Insert(Q, z)
9. return Extract-Min(Q)  //return the root of the tree
```

Time complexity: O(nlgn).

- Extract-Min(Q) needs O(lg n) by a heap operation.
- Requires initially O(n) time to build a binary heap.

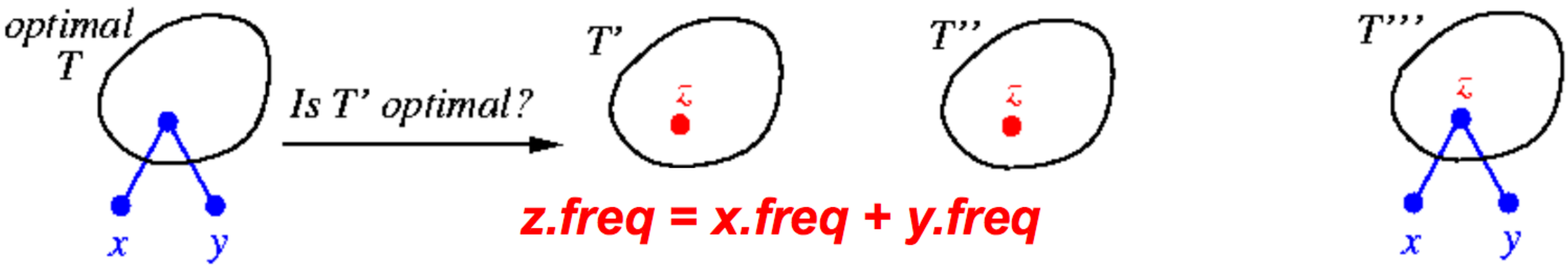
Huffman's Algorithm: Greedy Choice

Greedy choice: The binary tree for optimal prefix code must be full, and the two characters x and y with the lowest frequencies must have the same longest length and differ only in the last bit.



Huffman's Algorithm: Optimum Substructure

Optimal substructure: Let T be a full binary tree for an optimal prefix code over C. Let z be the parent of two leaf characters x and y. If z.freq = x.freq + y.freq, tree T' = T - {x, y} represents an optimal prefix code for C'= C - {x, y} U {z}.



z.freq = x.freq + y.freq

B(T) = B(T')+x.freq+y.freq

$(d_T(x)=d_T(y)=d_{T'}(z)+1)$

If T' is not optimal, find T'' s.t. B(T'')<B(T')

z is in C' => z is a leaf of T''

Add x, y as z's children to form T'''

=> B(T''') = B(T'')+x.freq+y.freq

< B(T')+x.freq+y.freq

= B(t)

Contradiction!!

Minimum Spanning Tree Problem

MST: The subset of edges that connected all vertices in the graph, and has minimum total weight

Greedy algorithm for solving MST

- Prim's algorithm
- Kruskal's algorithm

Minimum Spanning Trees

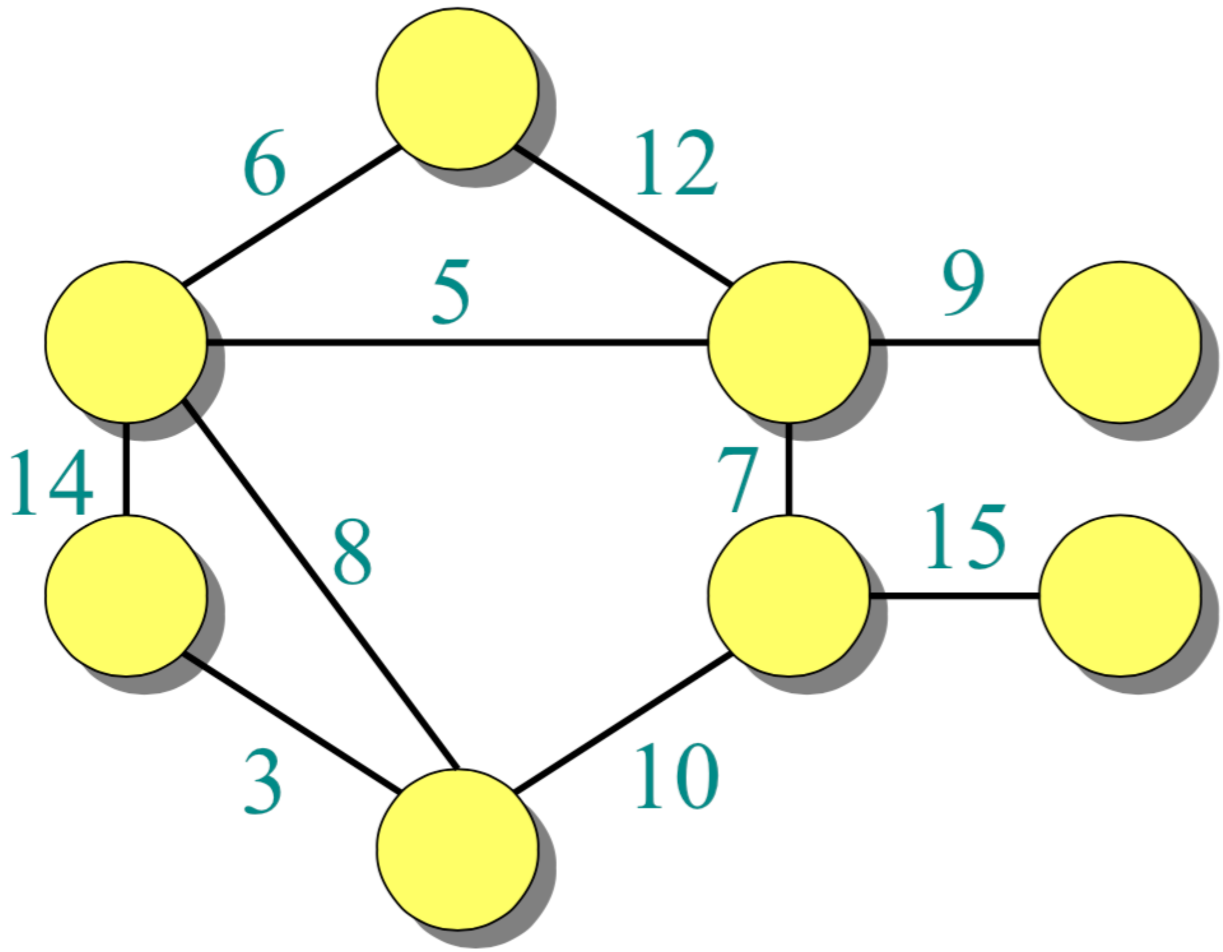
Input: A connected, undirected graph $G = (V, E)$ with weight function $w : E \to R$.

- For simplicity, assume that all edge weights are distinct. (CLRS covers the general case.)

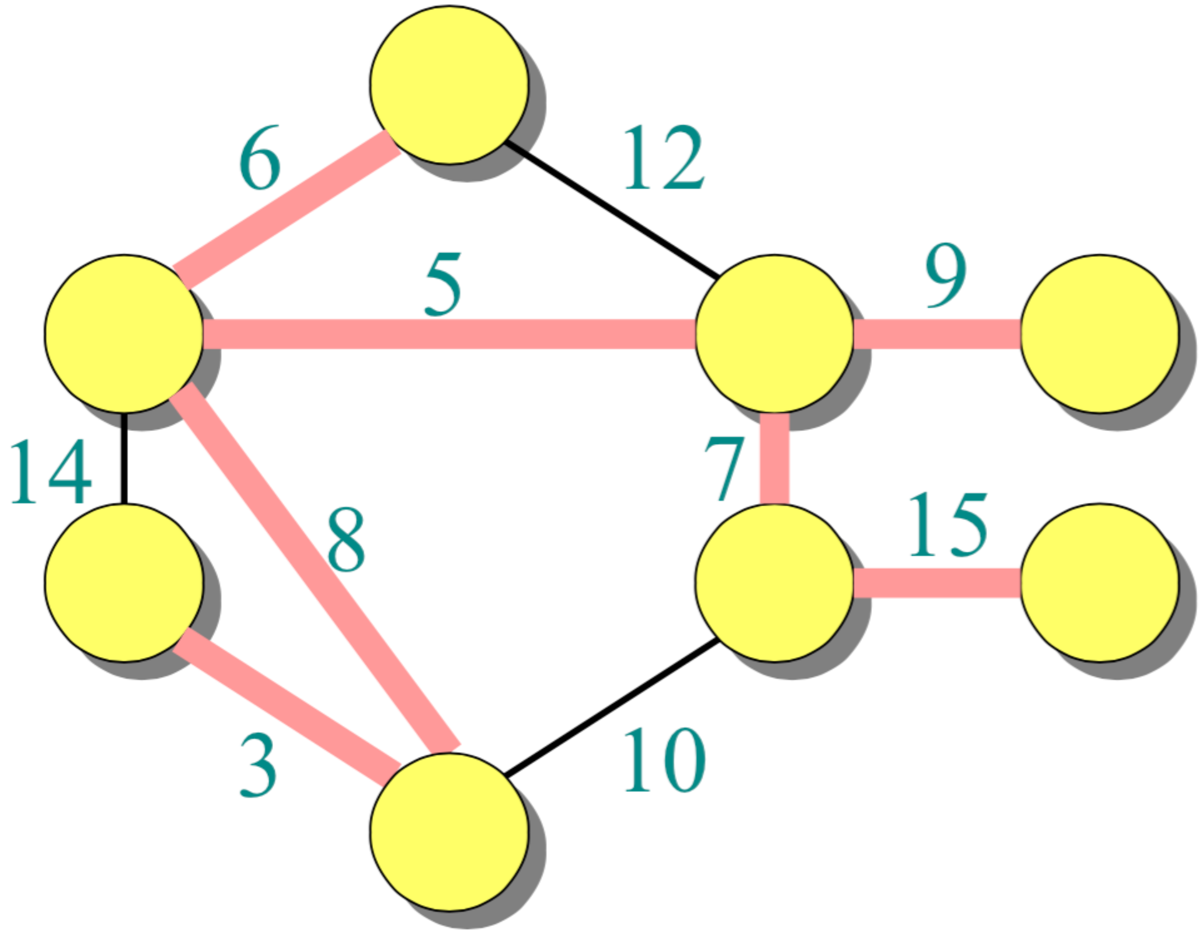
Output: A spanning tree T — a tree that connects all vertices — of minimum weight:

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

Example of MST



Example of MST



MST Algorithms

Greedy Algorithms: Prim, Kruskal

1. Start with initial tree/forest

2. Gradually grow it by adding the lowest weight edge

3. Finish until all nodes are connected in a tree

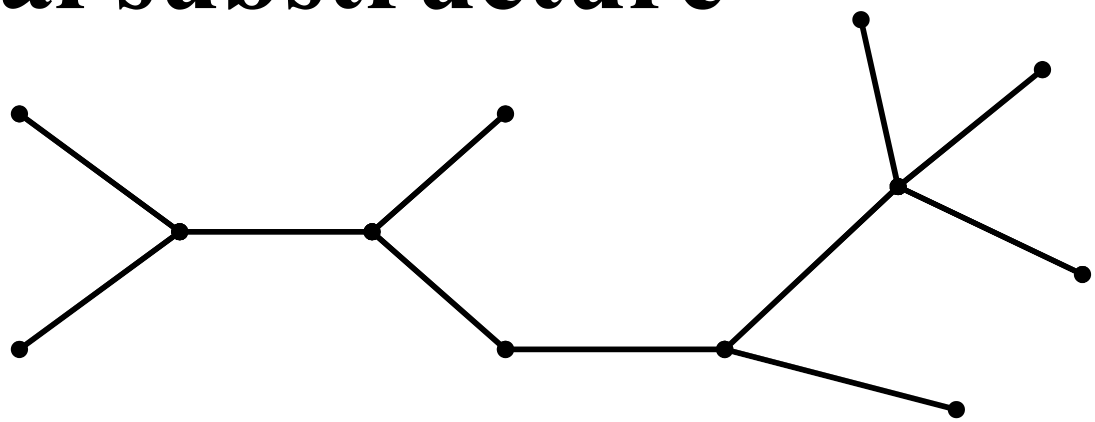
Prim's Algorithm

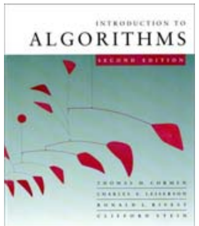
Slides from Demaine and Leiserson

Optimal substructure

MST T:

(Other edges of G are not shown.)

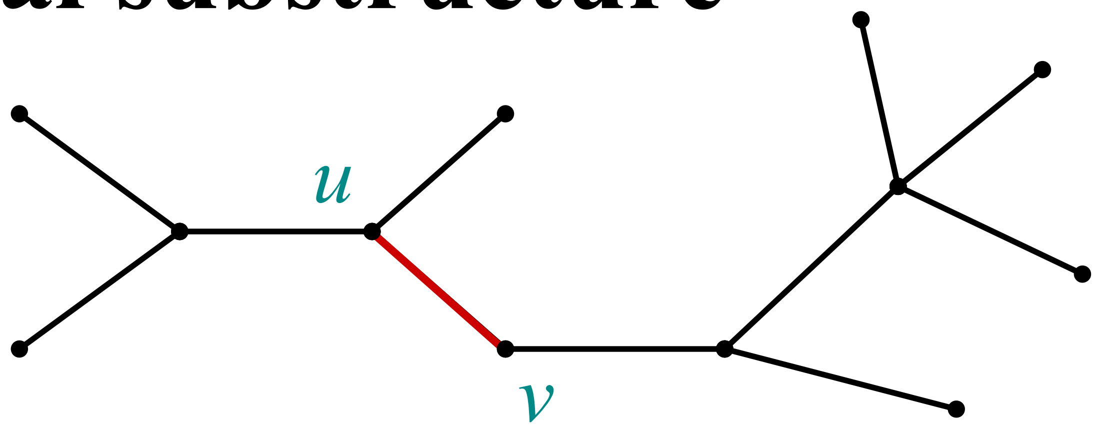




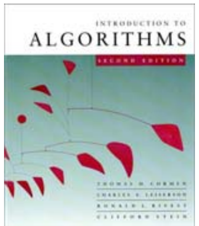
Optimal substructure

MST T:

(Other edges of G are not shown.)



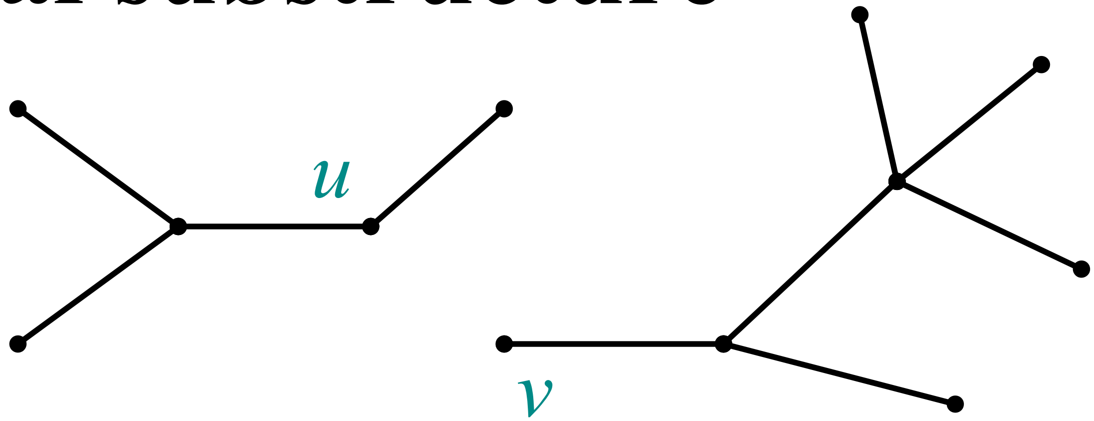
Remove any edge $(u, v) \in T$.



Optimal substructure

MST T:

(Other edges of G are not shown.)

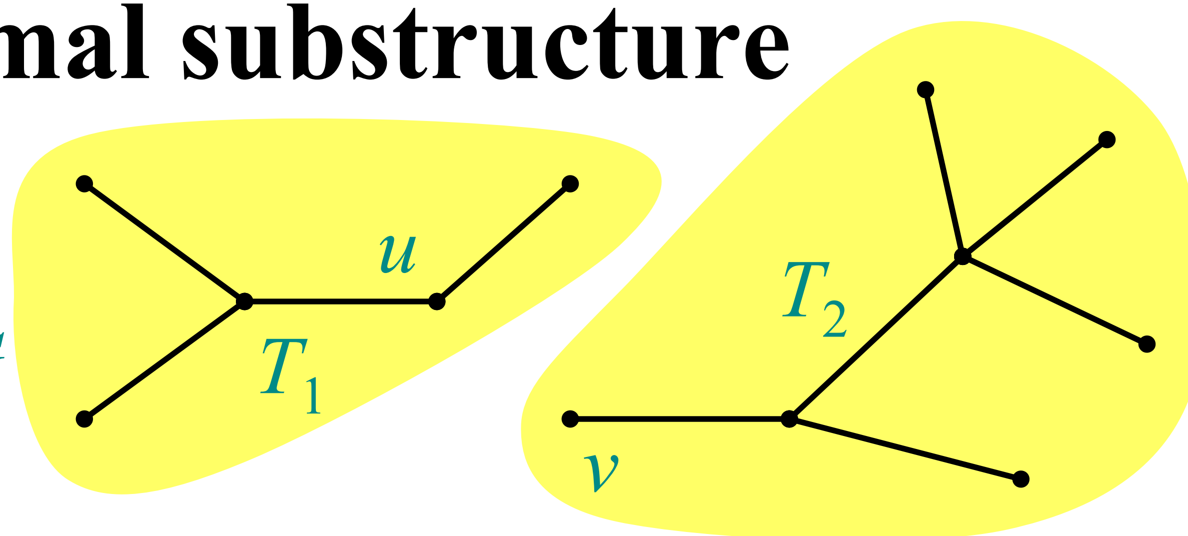


Remove any edge $(u, v) \in T$.

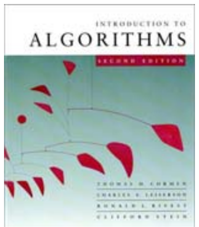
Optimal substructure

MST T:

(Other edges of G are not shown.)



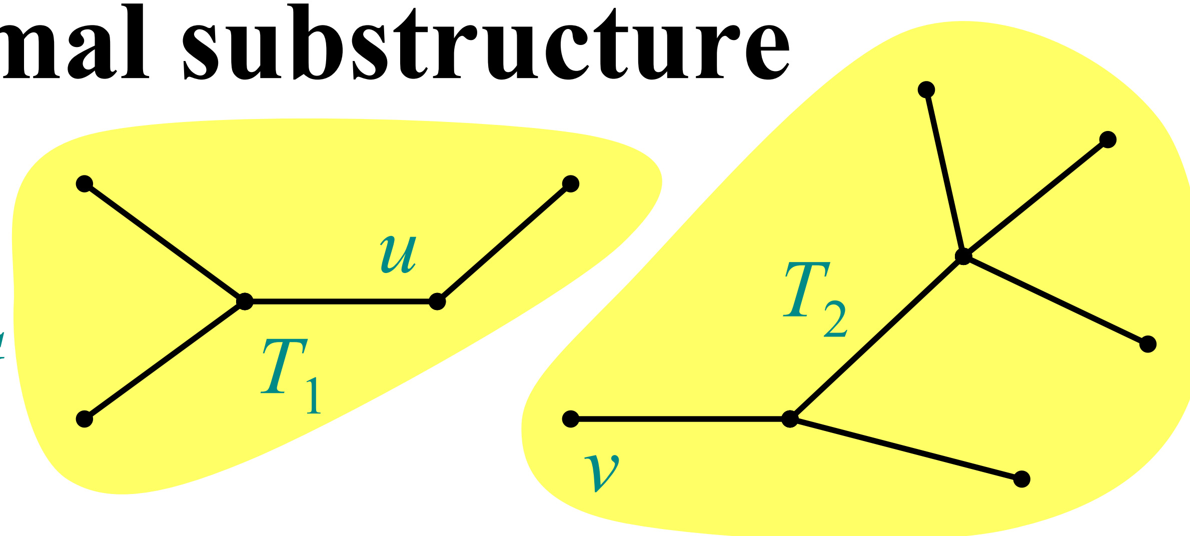
Remove any edge $(u, v) \in T$. Then, T is partitioned into two subtrees T_1 and T_2.



Optimal substructure

MST T:

(Other edges of G are not shown.)

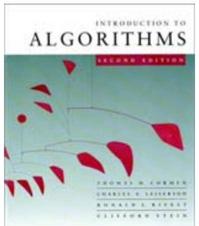


Remove any edge $(u, v) \in T$. Then, T is partitioned into two subtrees T_1 and T_2.

Theorem. The subtree T_1 is an MST of $G_1 = (V_1, E_1)$, the subgraph of G ***induced*** by the vertices of T_1:

$$V_1 = \text{vertices of } T_1,$$
$$E_1 = \{ (x, y) \in E : x, y \in V_1 \}.$$

Similarly for T_2.



Proof of optimal substructure

Proof. Cut and paste:

$$w(T) = w(u, v) + w(T_1) + w(T_2).$$

If T_1' were a lower-weight spanning tree than T_1 for G_1, then $T' = \{(u, v)\} \cup T_1' \cup T_2$ would be a lower-weight spanning tree than T for G. $\square$

Proof of optimal substructure

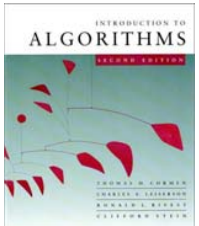
Proof. Cut and paste:

$$w(T) = w(u, v) + w(T_1) + w(T_2).$$

If T_1' were a lower-weight spanning tree than T_1 for G_1, then $T' = \{(u, v)\} \cup T_1' \cup T_2$ would be a lower-weight spanning tree than T for G. $\square$

Do we also have overlapping subproblems?

- Yes.



Proof of optimal substructure

Proof. Cut and paste:

$$w(T) = w(u, v) + w(T_1) + w(T_2).$$

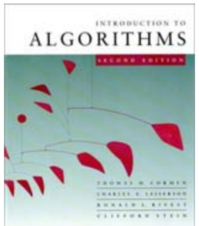
If T_1' were a lower-weight spanning tree than T_1 for G_1, then $T' = \{(u, v)\} \cup T_1' \cup T_2$ would be a lower-weight spanning tree than T for G. $\square$

Do we also have overlapping subproblems?

- Yes.

Great, then dynamic programming may work!

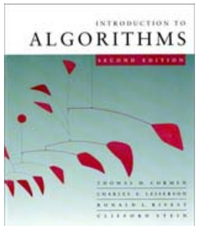
- Yes, but MST exhibits another powerful property which leads to an even more efficient algorithm.



Hallmark for “greedy” algorithms

Greedy-choice property

A locally optimal choice is globally optimal.

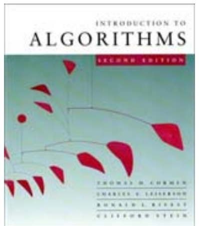


Hallmark for “greedy” algorithms

Greedy-choice property

A locally optimal choice is globally optimal.

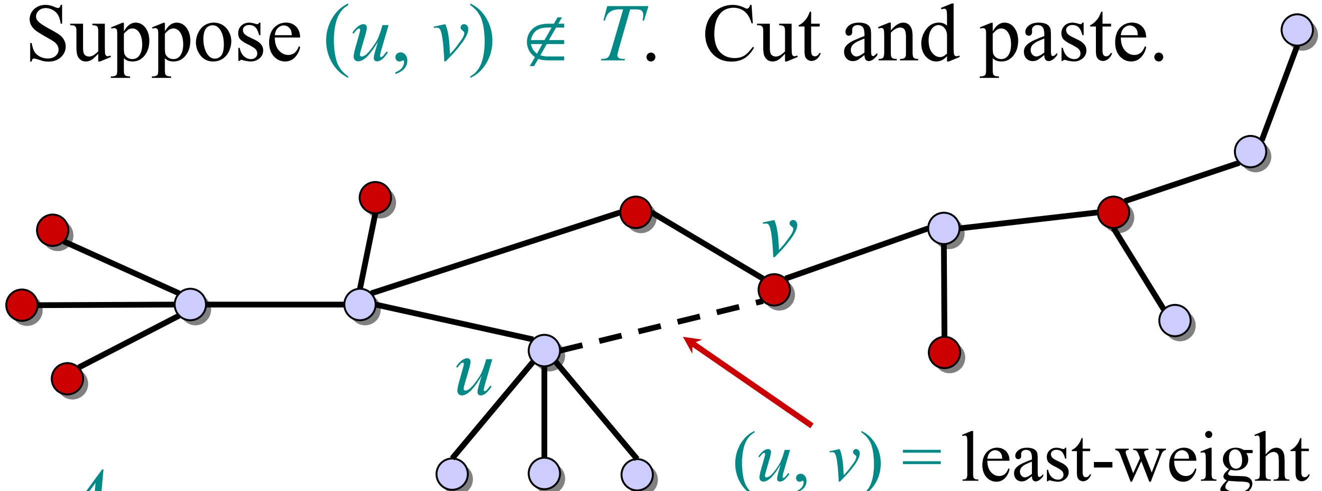
Theorem. Let T be the MST of $G = (V, E)$, and let $A \subseteq V$. Suppose that $(u, v) \in E$ is the least-weight edge connecting A to $V - A$. Then, $(u, v) \in T$.



Proof of theorem

Proof. Suppose $(u, v) \notin T$. Cut and paste.

T:



$\in A$

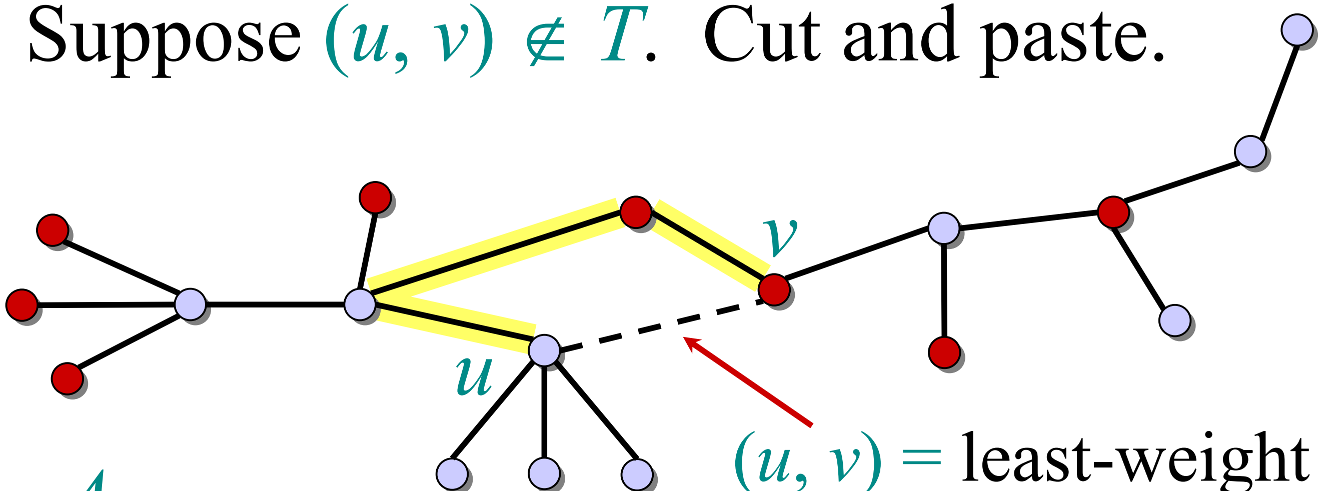
$\in V - A$

(u, v) = least-weight edge connecting A to $V - A$

Proof of theorem

Proof. Suppose $(u, v) \notin T$. Cut and paste.

T:

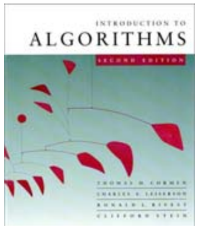


$\in A$

$\in V - A$

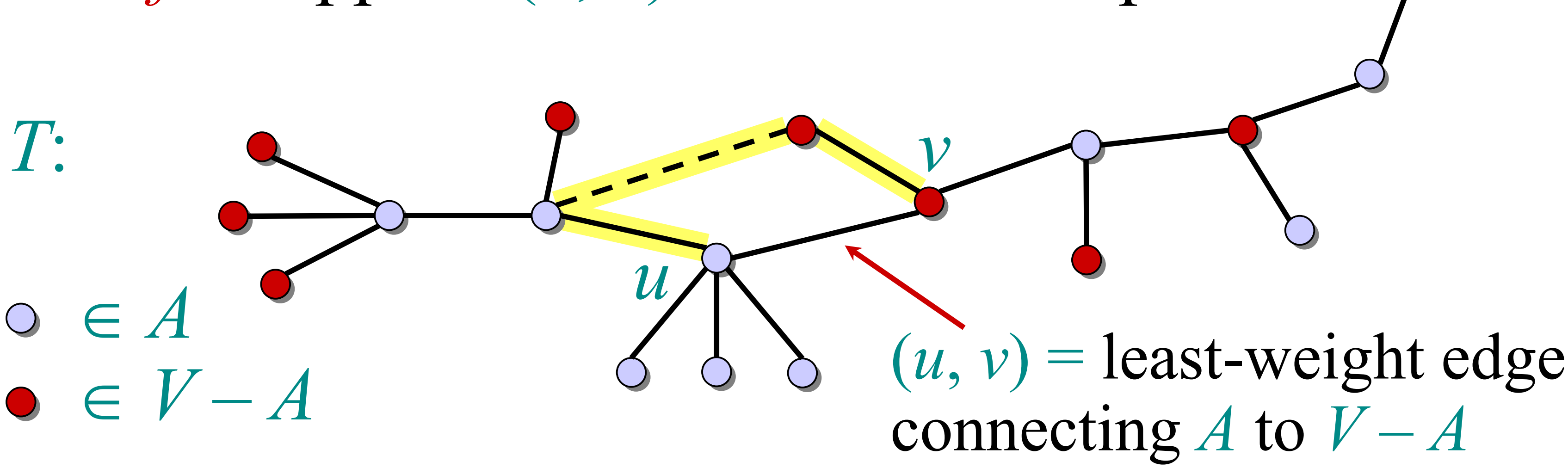
$(u, v) =$ least-weight edge connecting A to $V - A$

Consider the unique simple path from u to v in T.



Proof of theorem

Proof. Suppose $(u, v) \notin T$. Cut and paste.

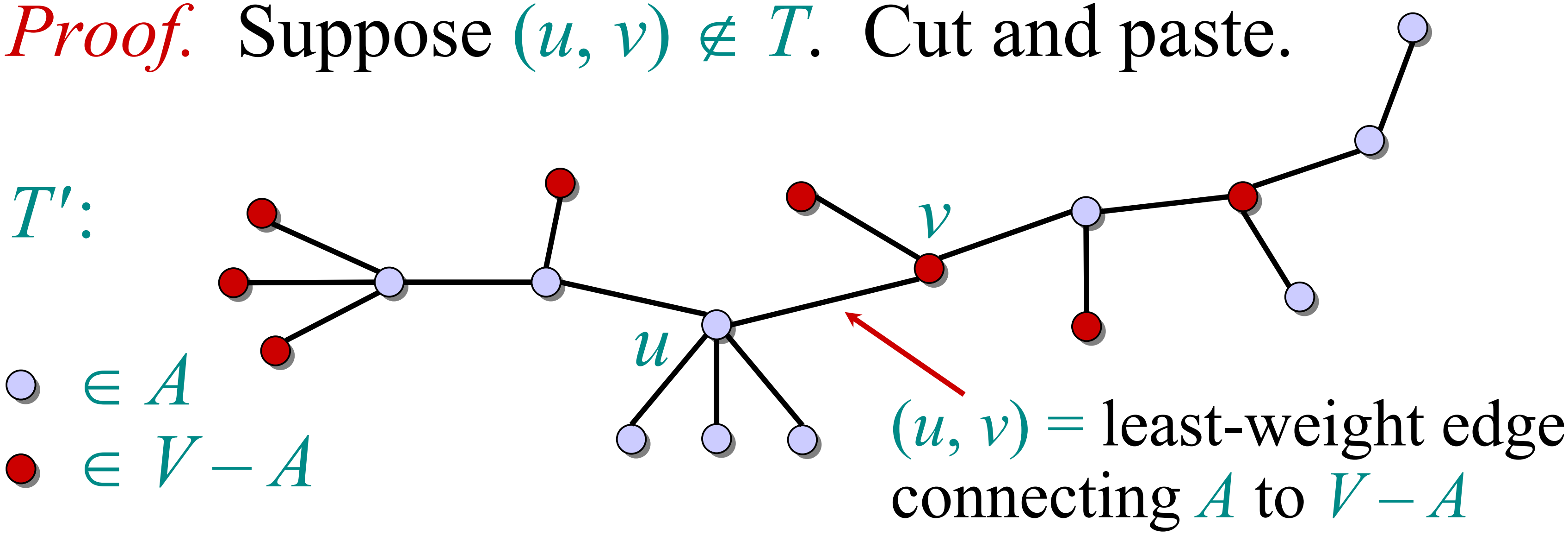


Consider the unique simple path from u to v in T.

Swap (u, v) with the first edge on this path that connects a vertex in A to a vertex in $V - A$.

Proof of theorem

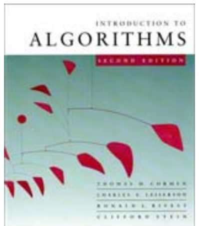
Proof. Suppose $(u, v) \notin T$. Cut and paste.



Consider the unique simple path from u to v in T.

Swap (u, v) with the first edge on this path that connects a vertex in A to a vertex in $V - A$.

A lighter-weight spanning tree than T results. $\square$



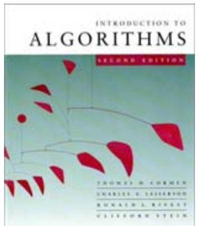
Prim's algorithm

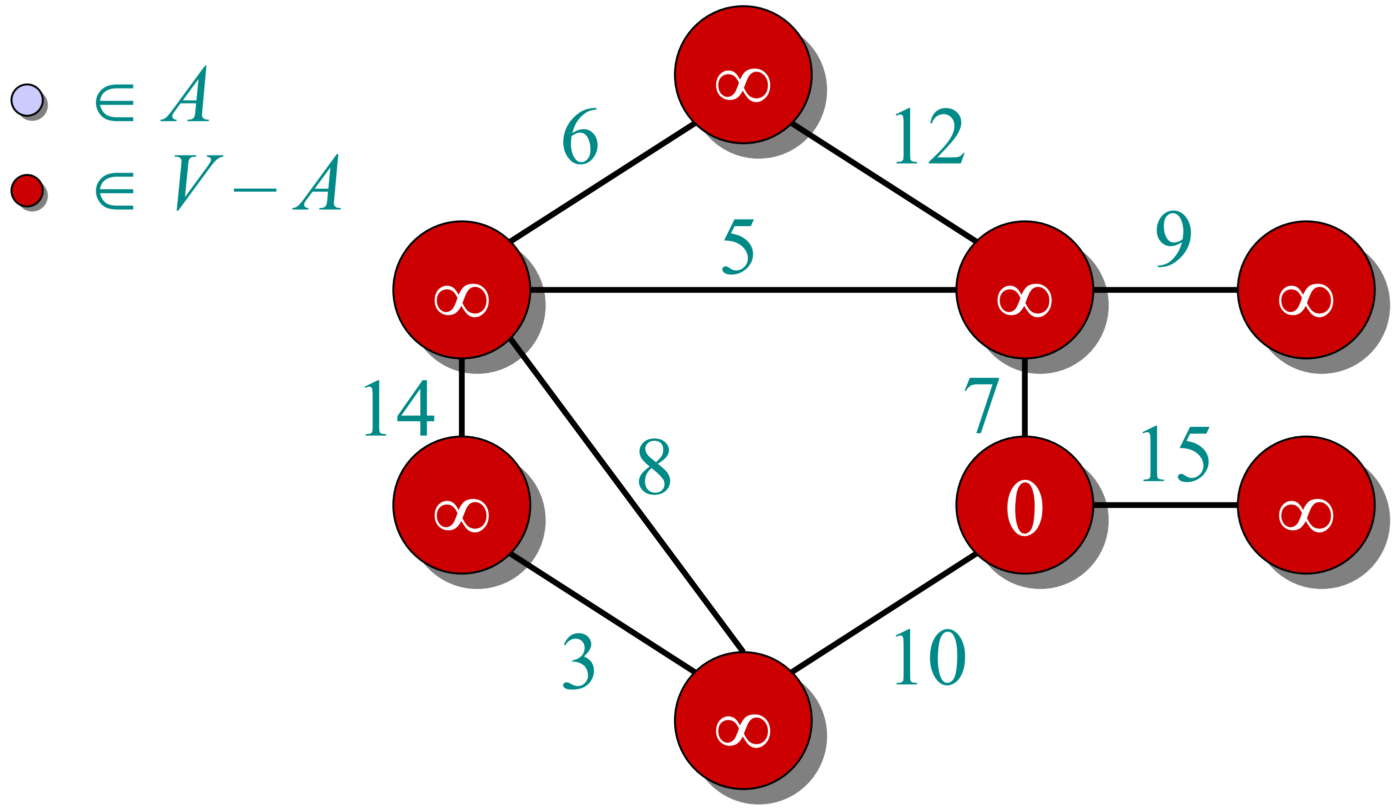
IDEA: Maintain $V - A$ as a priority queue Q. Key each vertex in Q with the weight of the least-weight edge connecting it to a vertex in A.

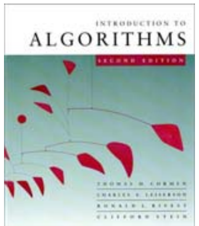
$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN(Q)
        for each $v \in Adj[u]$
            do if $v \in Q$ and $w(u, v) < key[v]$
                then $key[v] \leftarrow w(u, v)$ $\quad\triangleright$ DECREASE-KEY
                    $\pi[v] \leftarrow u$

At the end, $\{(v, \pi[v])\}$ forms the MST.

Example of Prim's algorithm

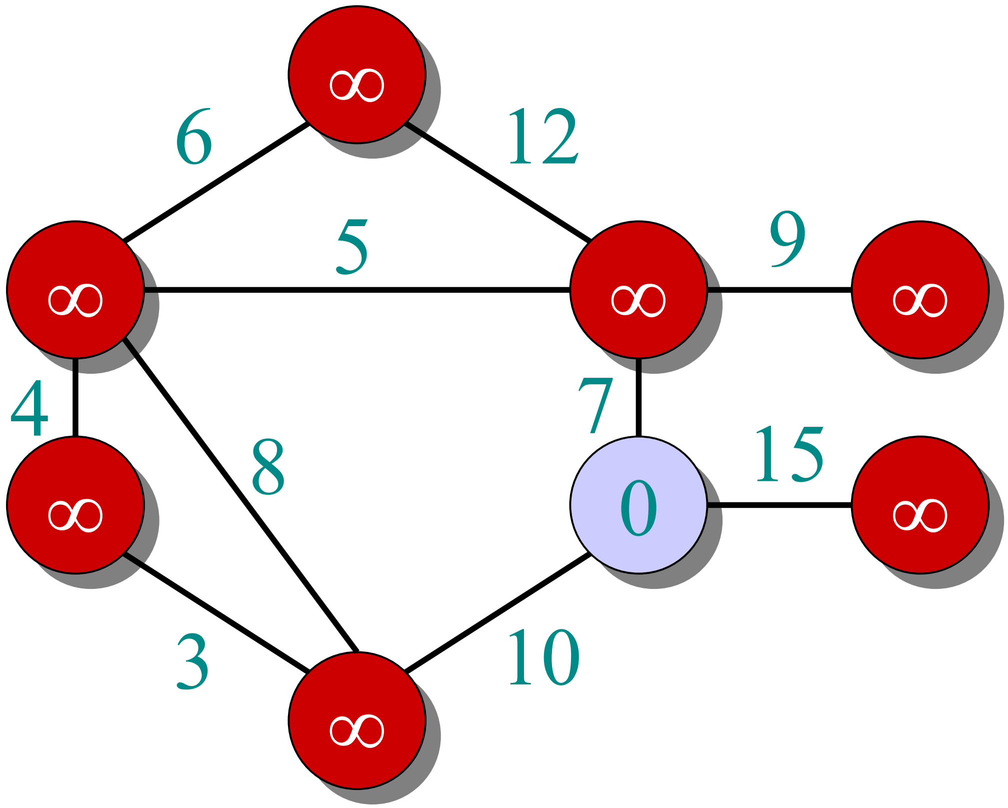




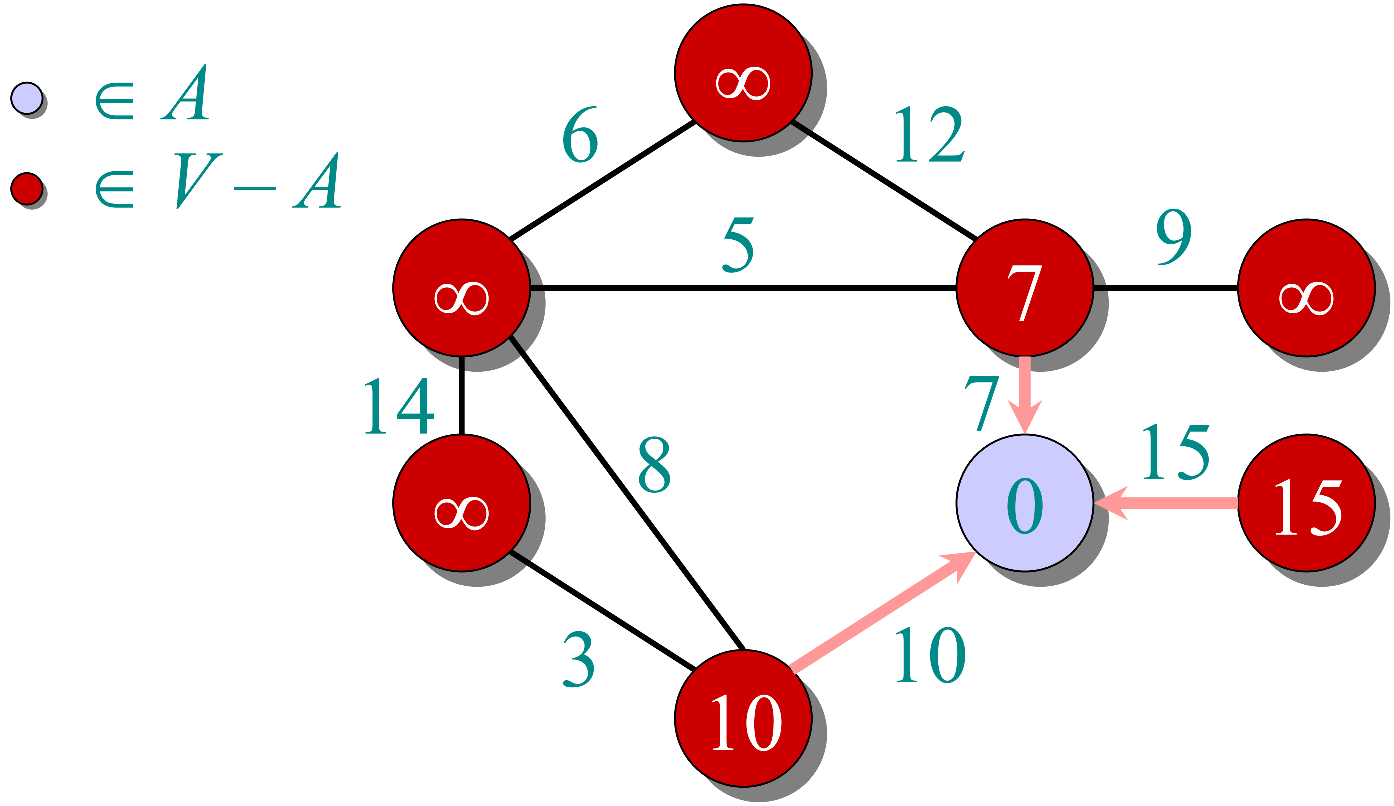


Example of Prim's algorithm

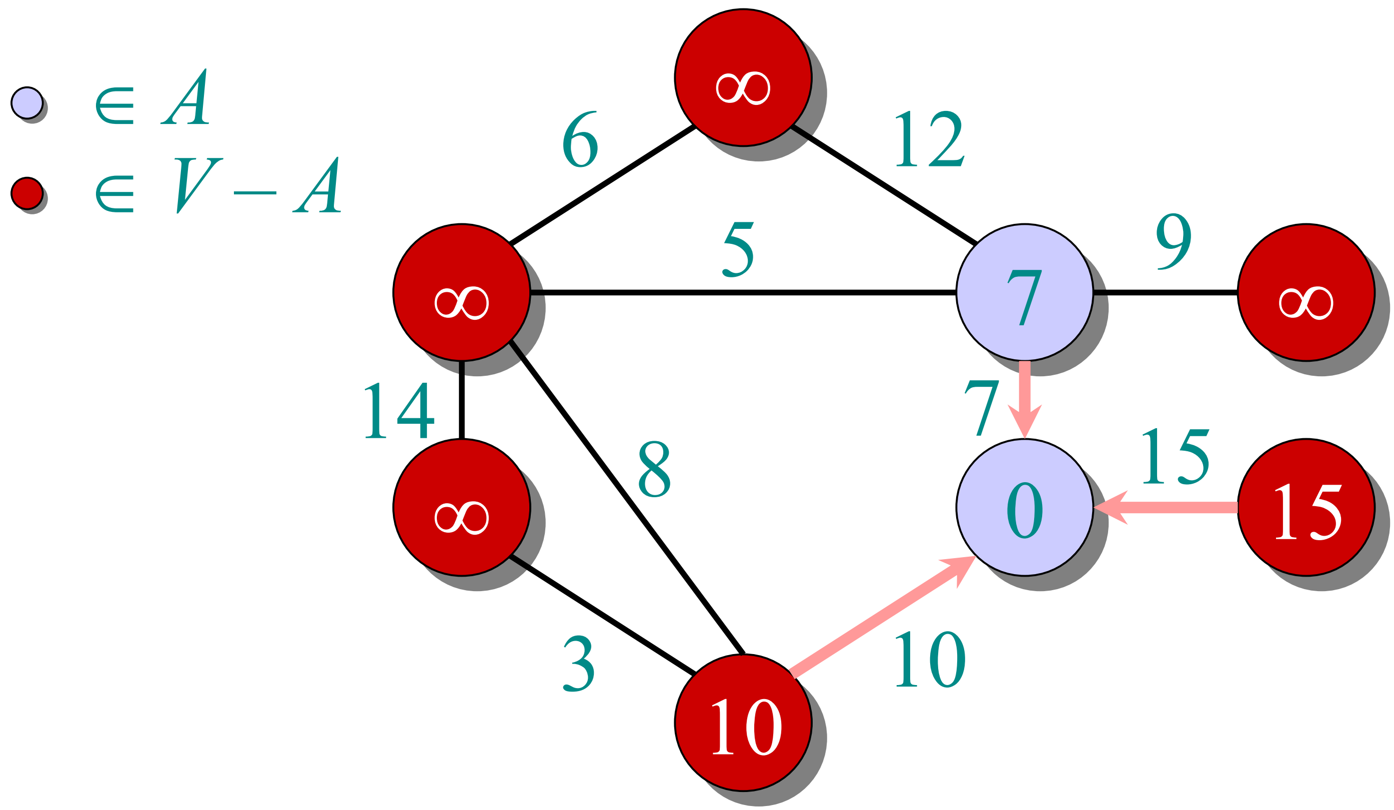
- $\in A$
- $\in V - A$

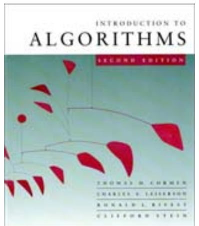


Example of Prim's algorithm

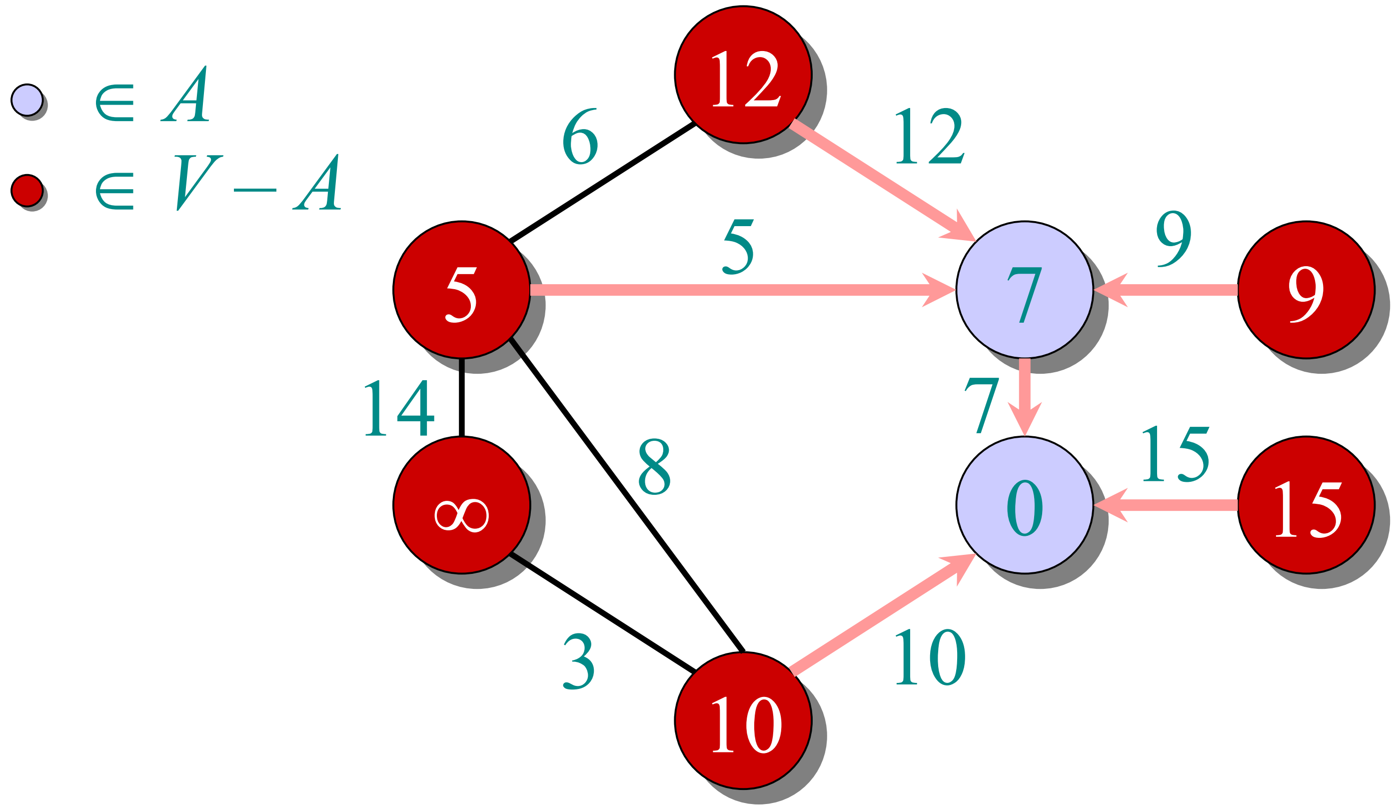


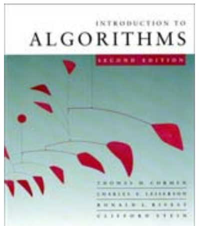
Example of Prim's algorithm



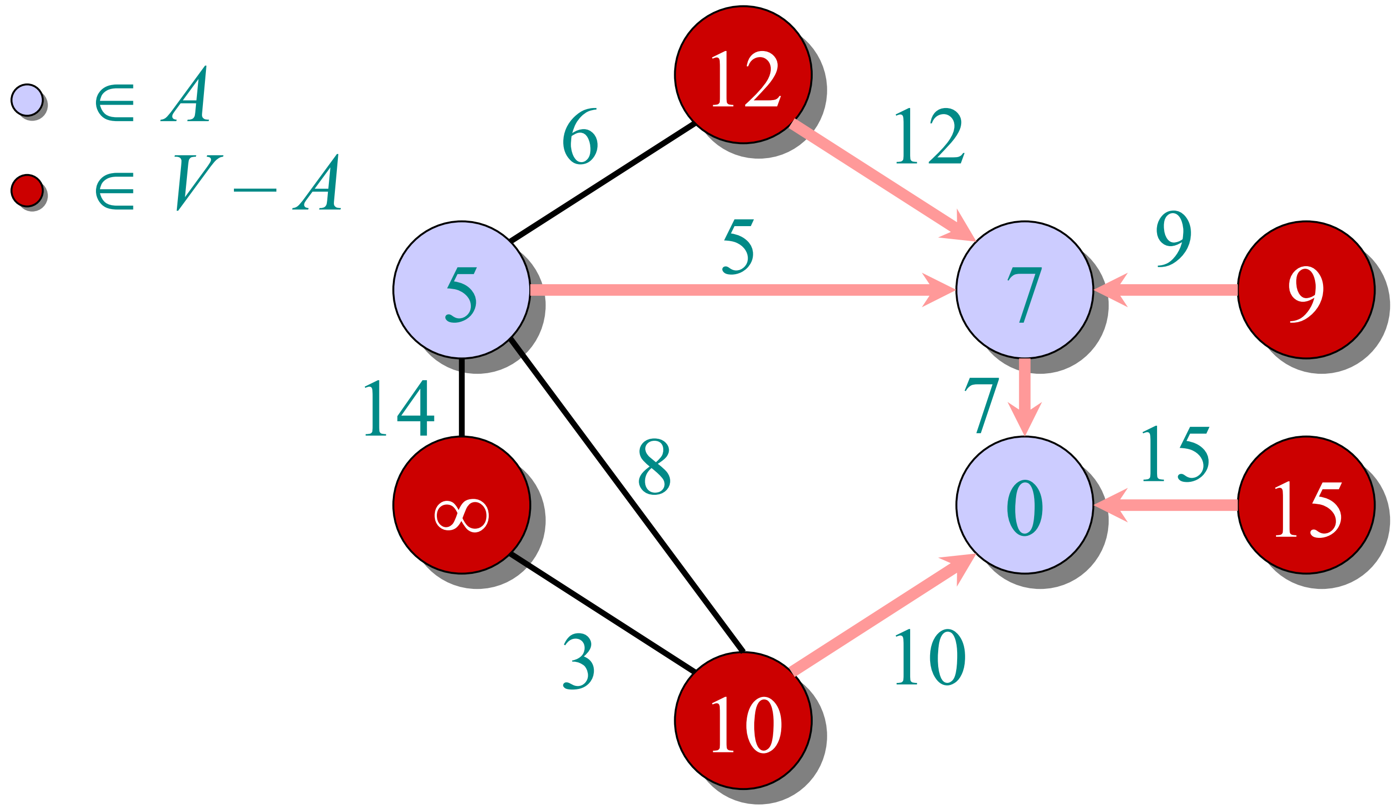


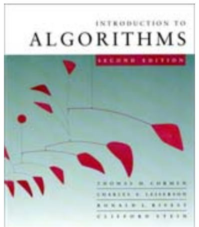
Example of Prim's algorithm



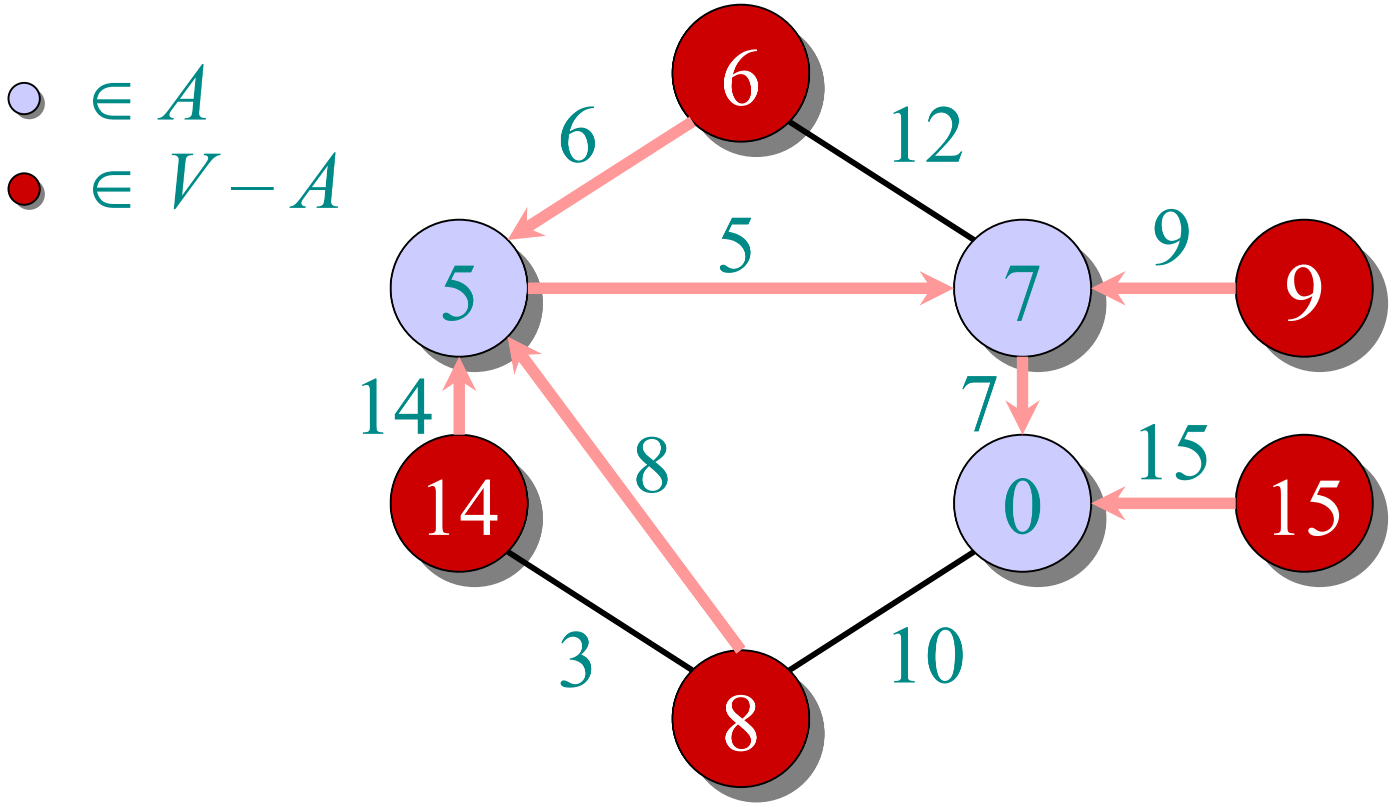


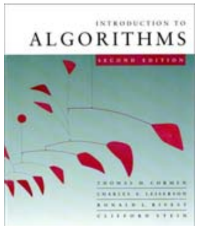
Example of Prim's algorithm



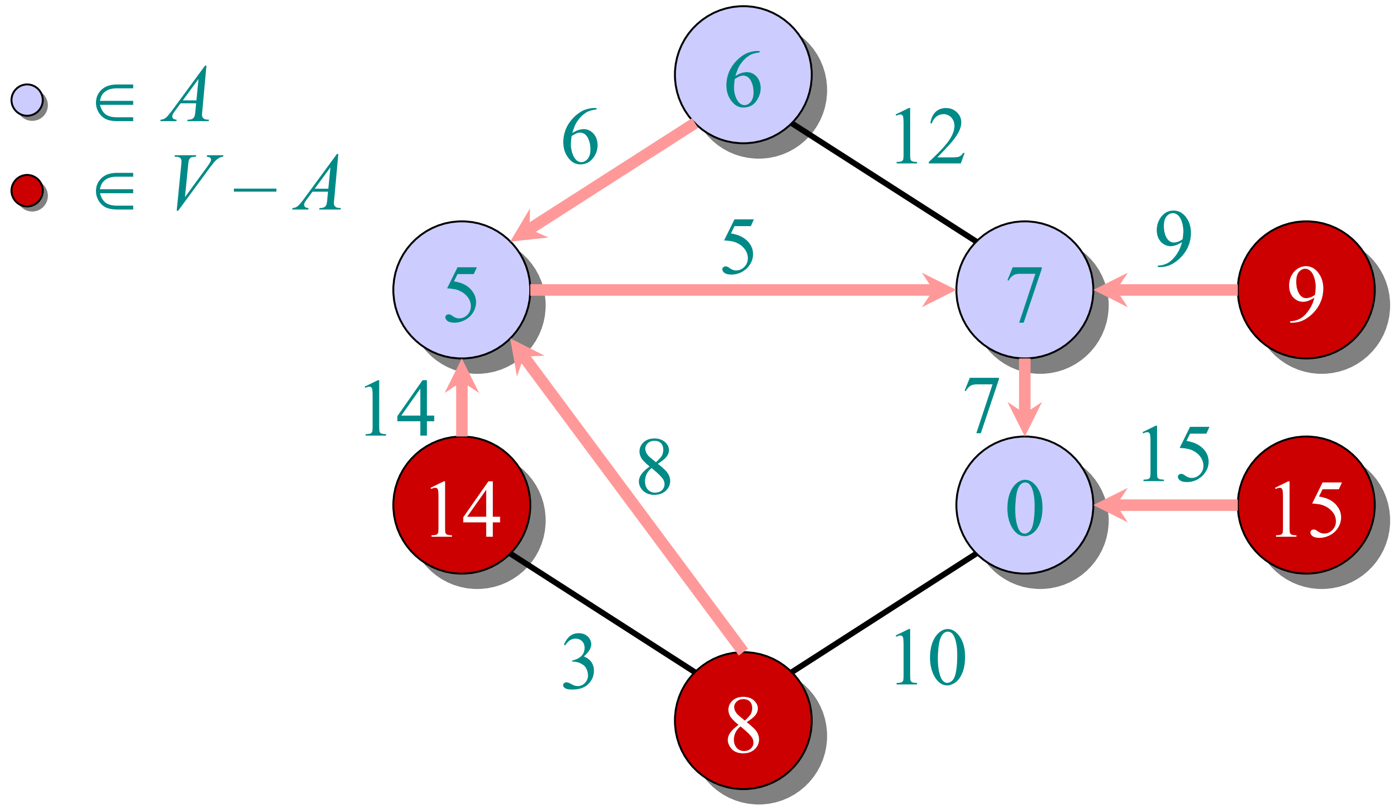


Example of Prim's algorithm

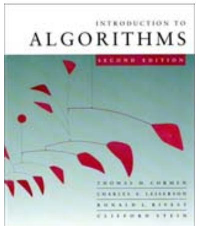


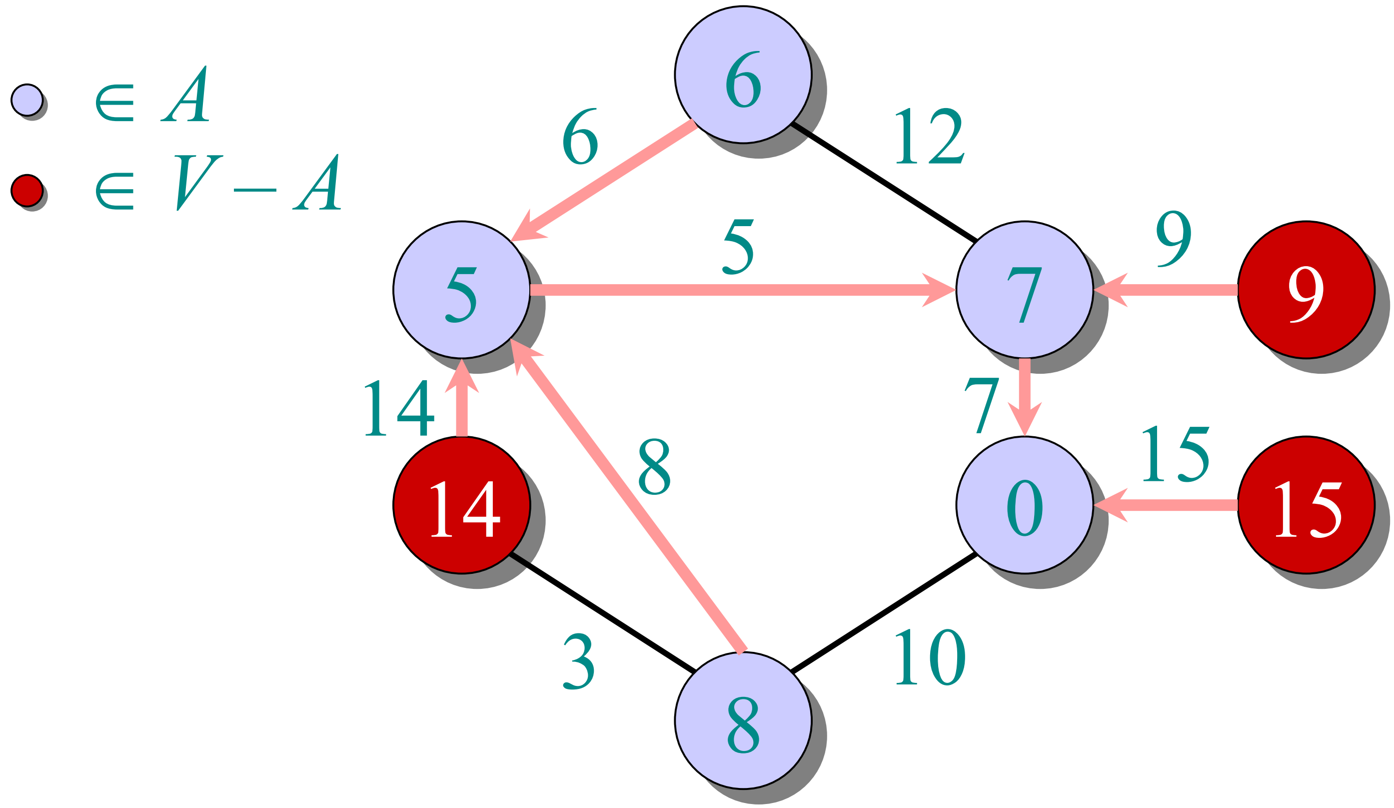


Example of Prim's algorithm

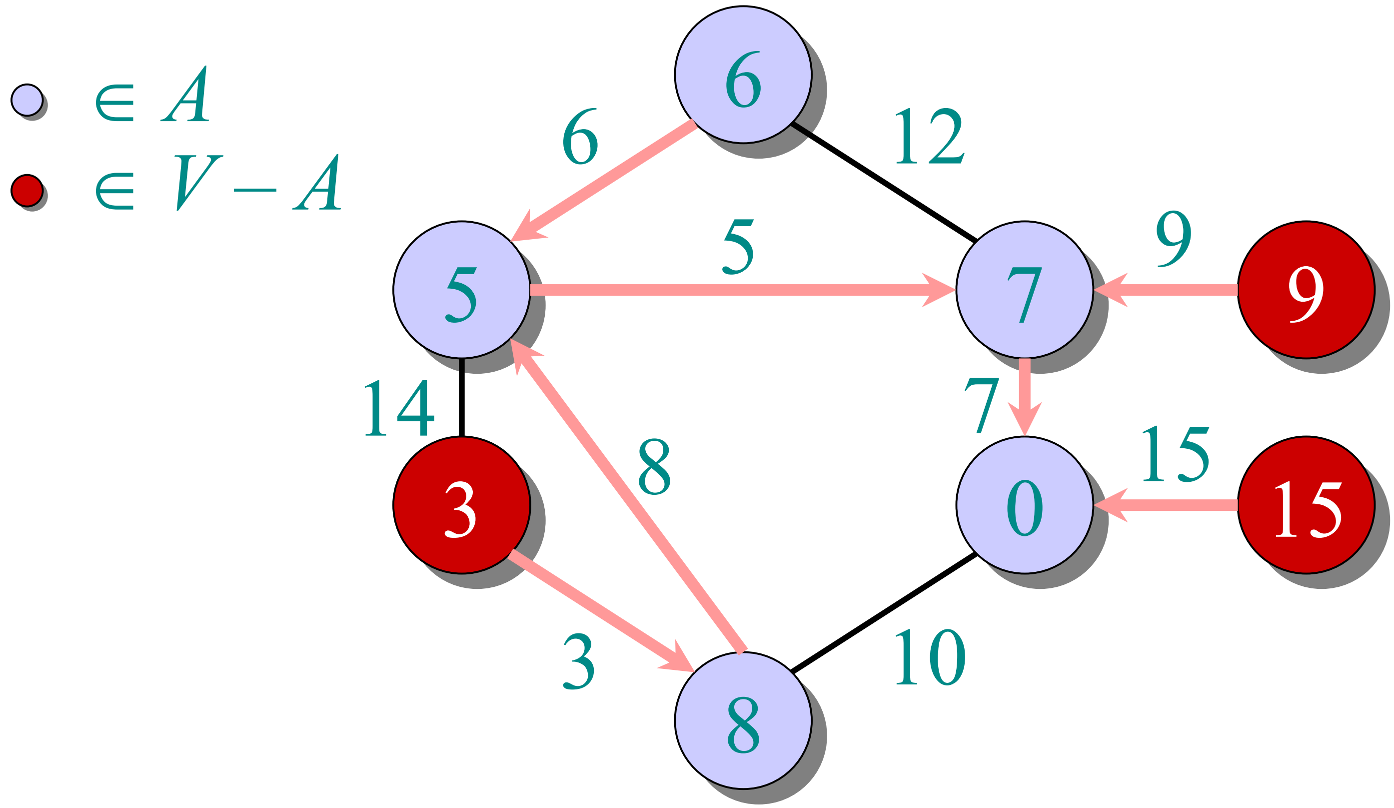


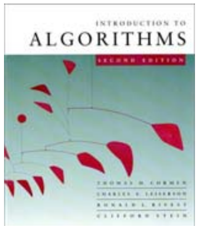
Example of Prim's algorithm

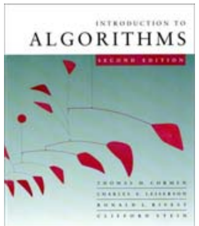




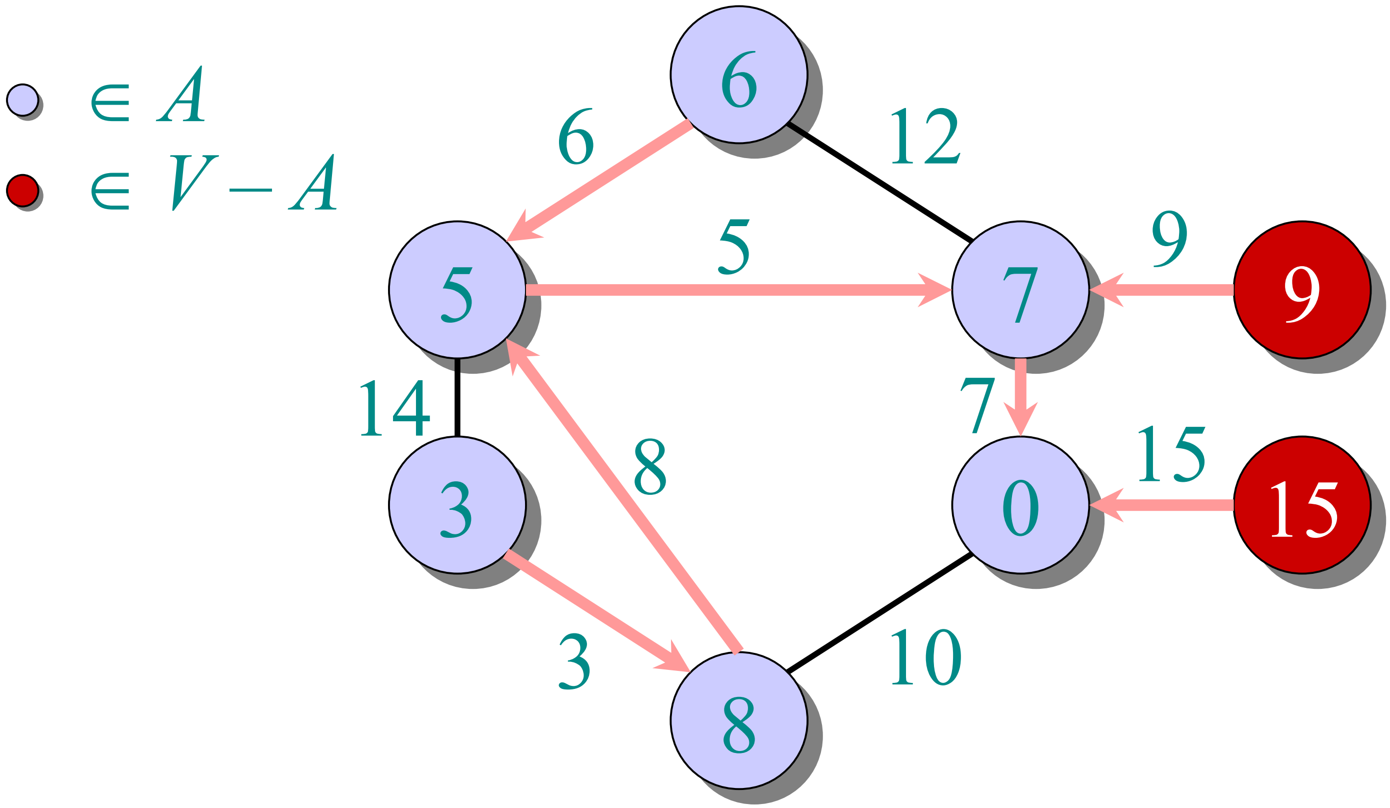
Example of Prim's algorithm

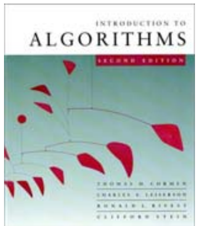






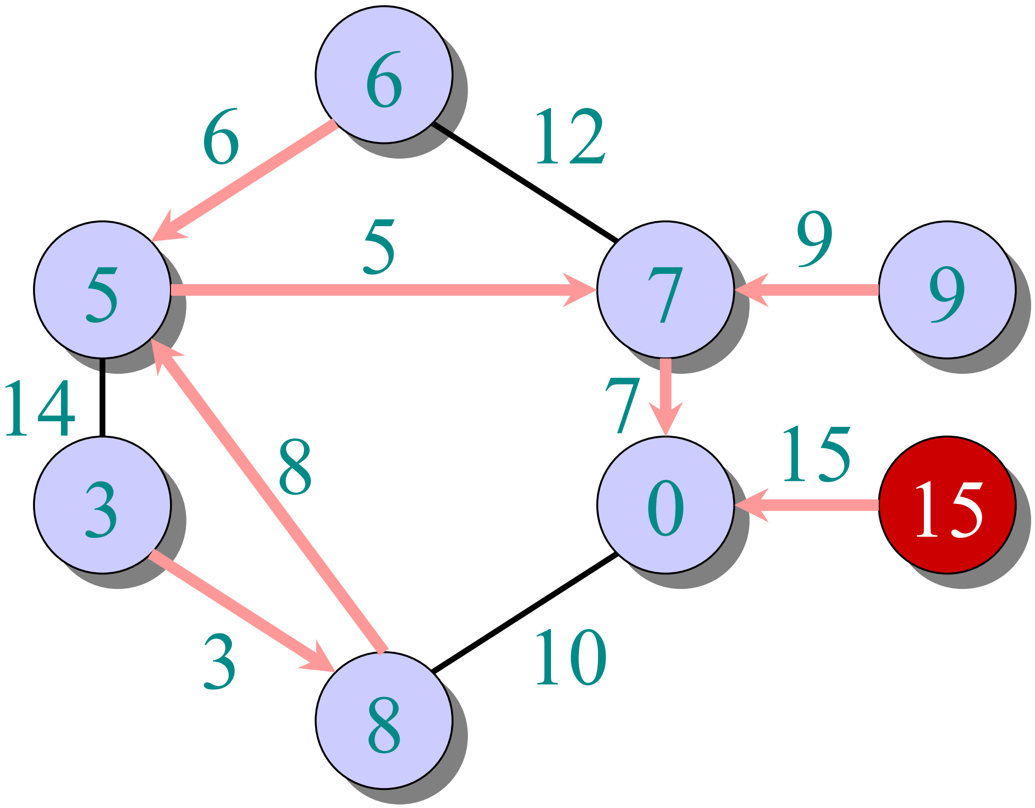
Example of Prim's algorithm

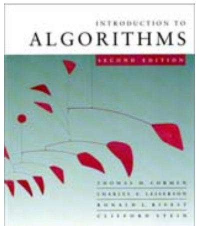




Example of Prim's algorithm

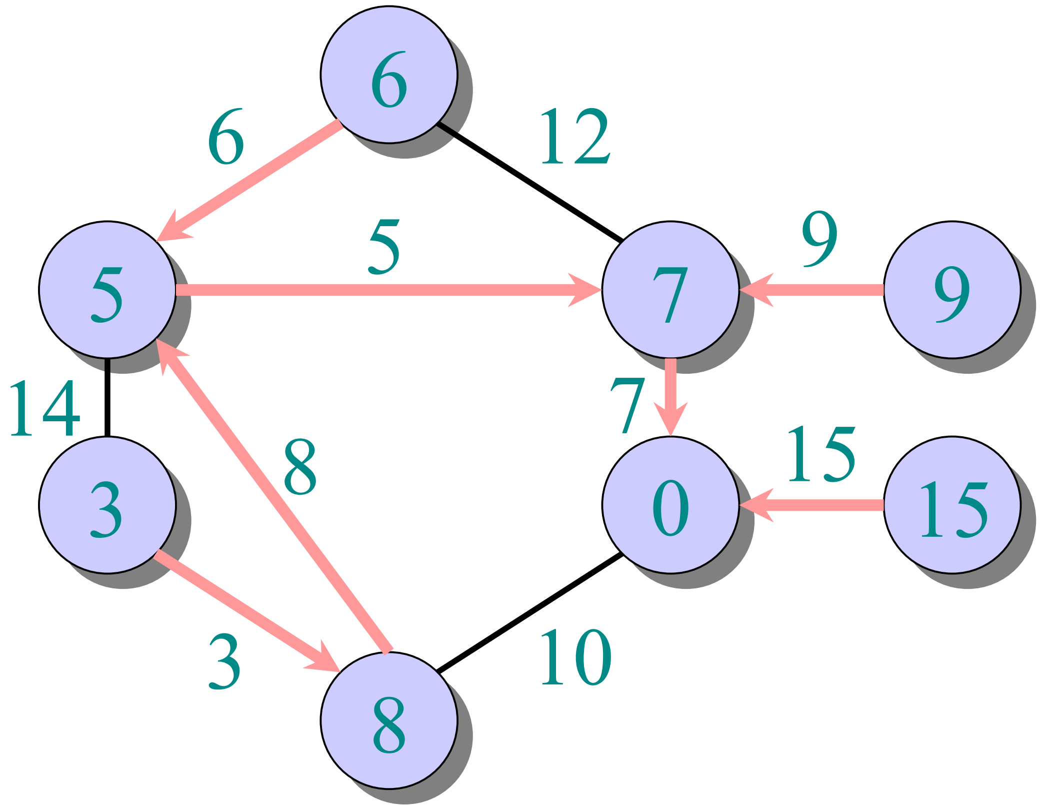
- $\in A$
- $\in V - A$

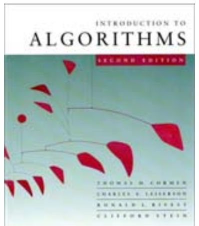




Example of Prim's algorithm

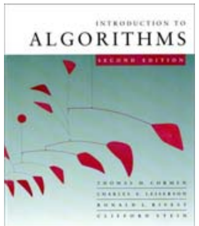
- $\in A$
- $\in V - A$





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

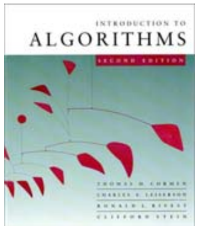


Analysis of Prim

$\Theta(V)$ total:

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

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



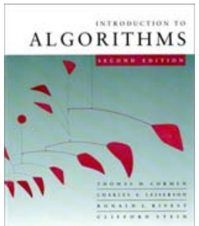
Analysis of Prim

$\Theta(V)$ 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:

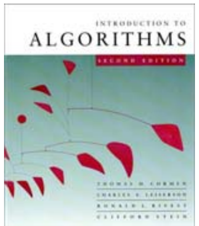
$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$

$|V|$ 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 $s \in V$

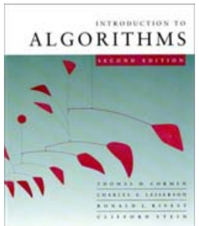
$|V|$ times:

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:

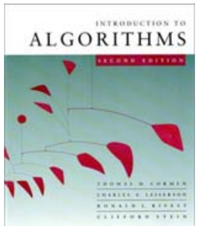
   for each $v \in Adj[u]$
    do if $v \in Q$ and $w(u, v) < key[v]$
     then $key[v] \leftarrow w(u, v)$
      $\pi[v] \leftarrow u$

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)

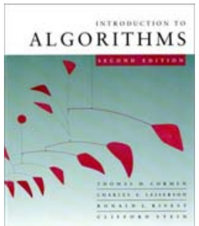
Time = $\Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$



Analysis of Prim (continued)

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

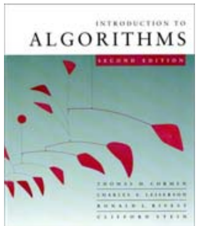
Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total



Analysis of Prim (continued)

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

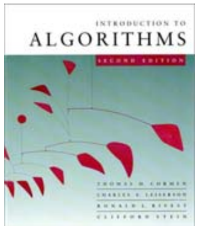
Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$



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

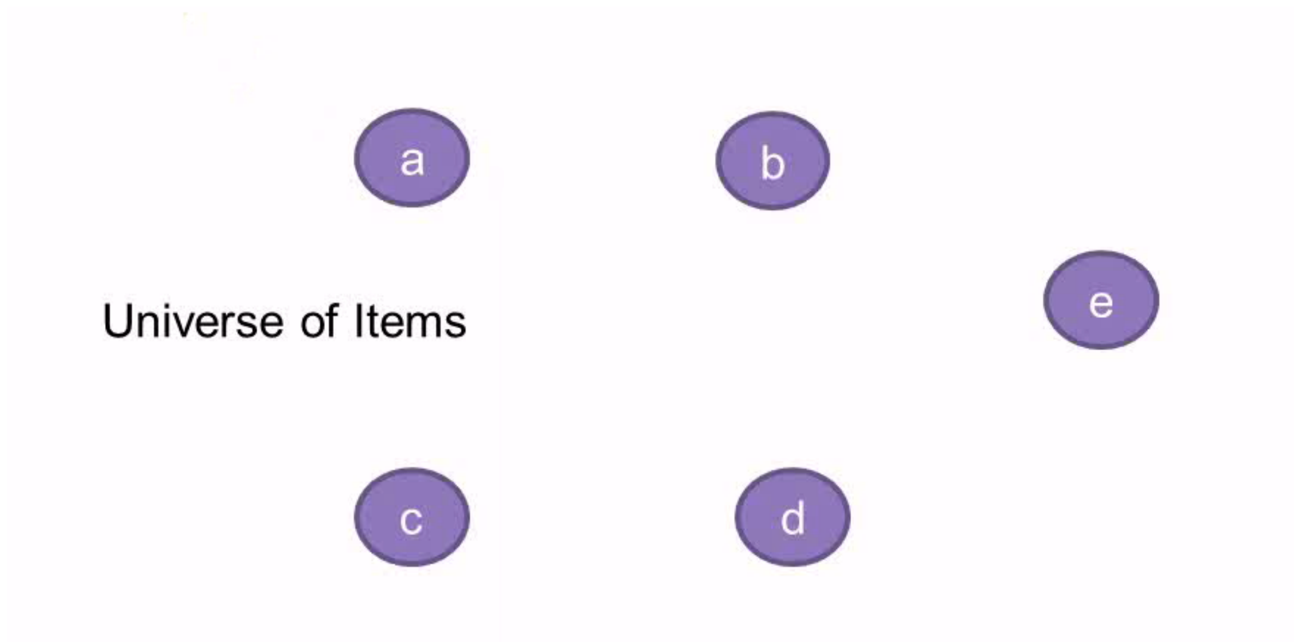
Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$ amortized	$O(1)$ amortized	$O(E + V \lg V)$ worst case

MST Algorithms

- Kruskal's algorithm (see CLRS):
  - Uses the disjoint-set data structure
  - Running time = $O(E \lg V)$

Disjoint Set

A group of sets where no item can be in more than one set

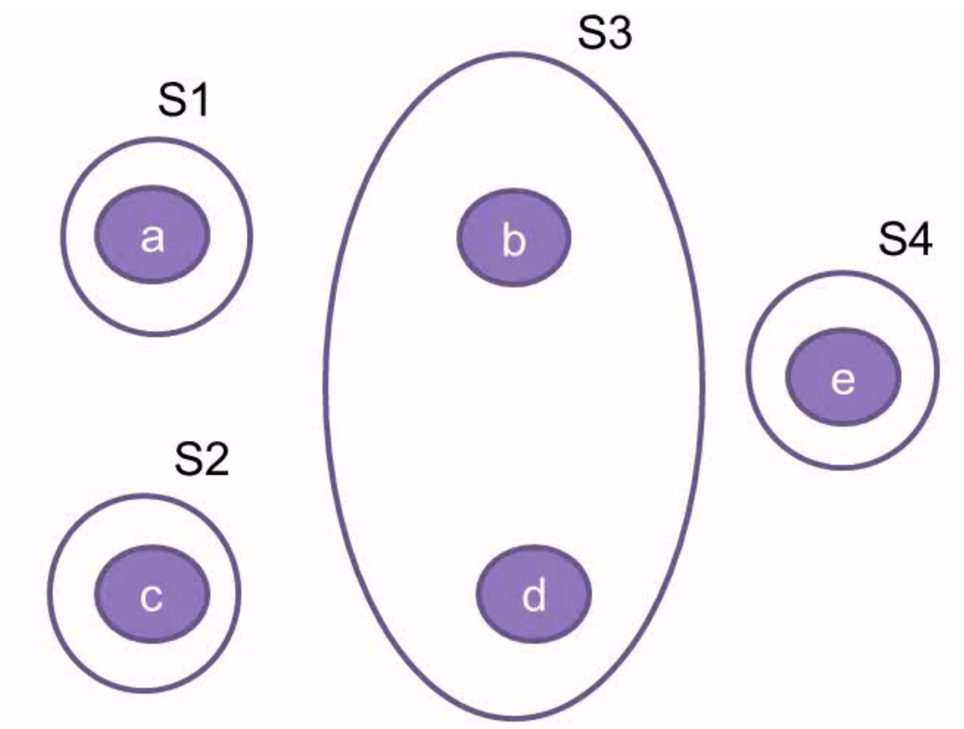


Disjoint Set

A group of sets where no item can be in more than one set

Supported Operations:
Find()
Union()

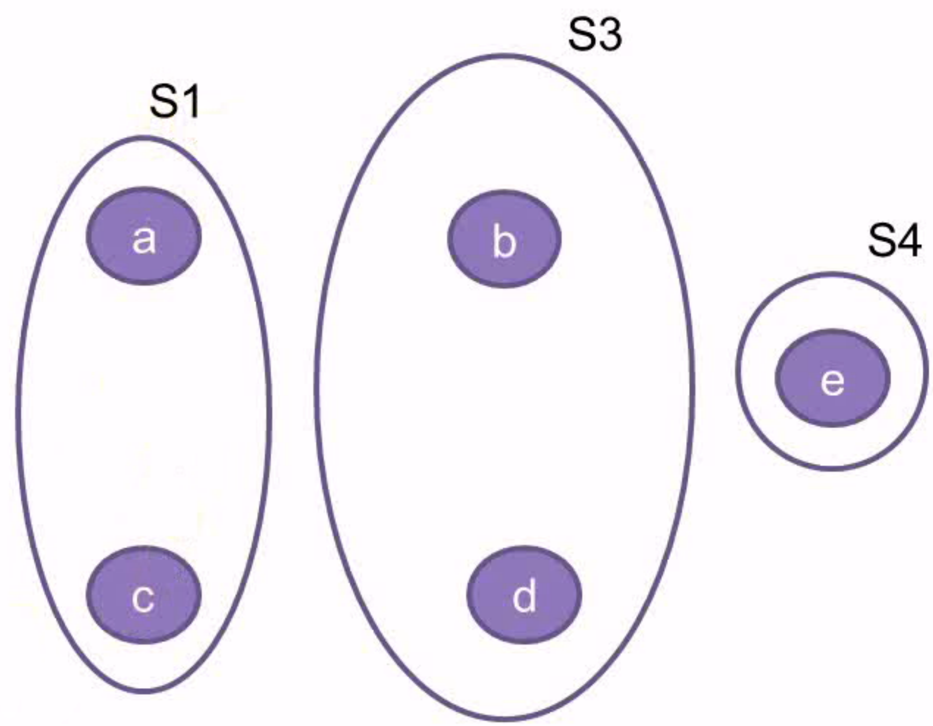
Find(d) => S3
Union(S2, S1)



Disjoint Set

A group of sets where no item can be in more than one set

Supported Operations:
Find()
Union()



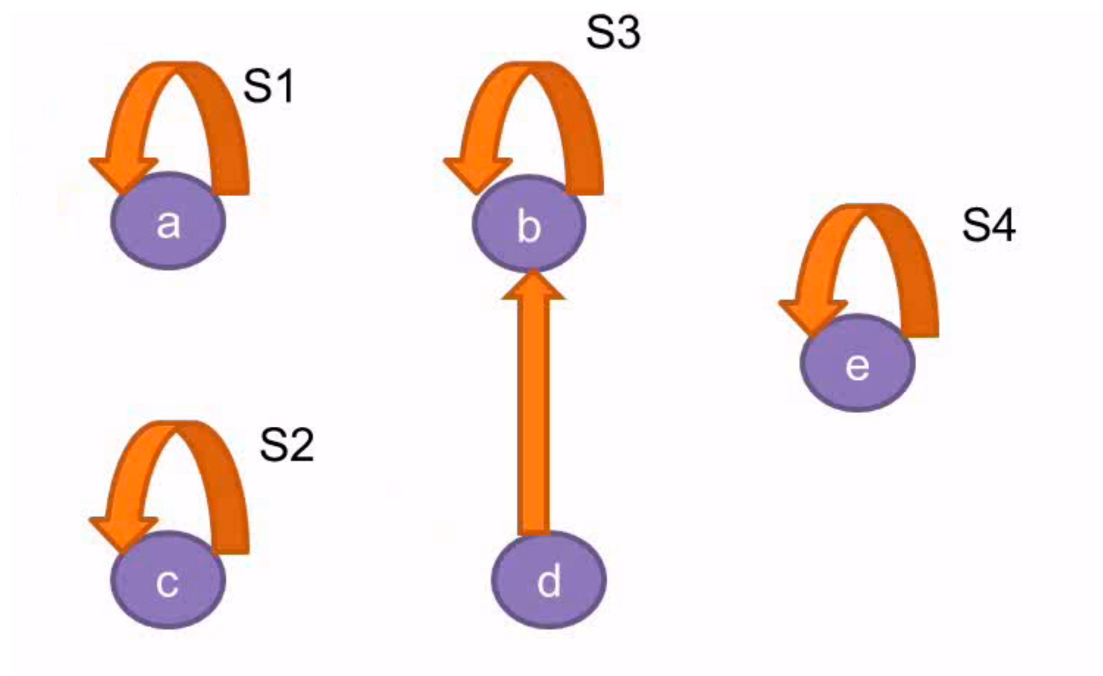
Find(d) => S3
Union(S2, S1)

Tree Based Disjoint Set

- Each set is a tree
- A set is identified by the root of the tree

Supported Operations:
Find()
Union()

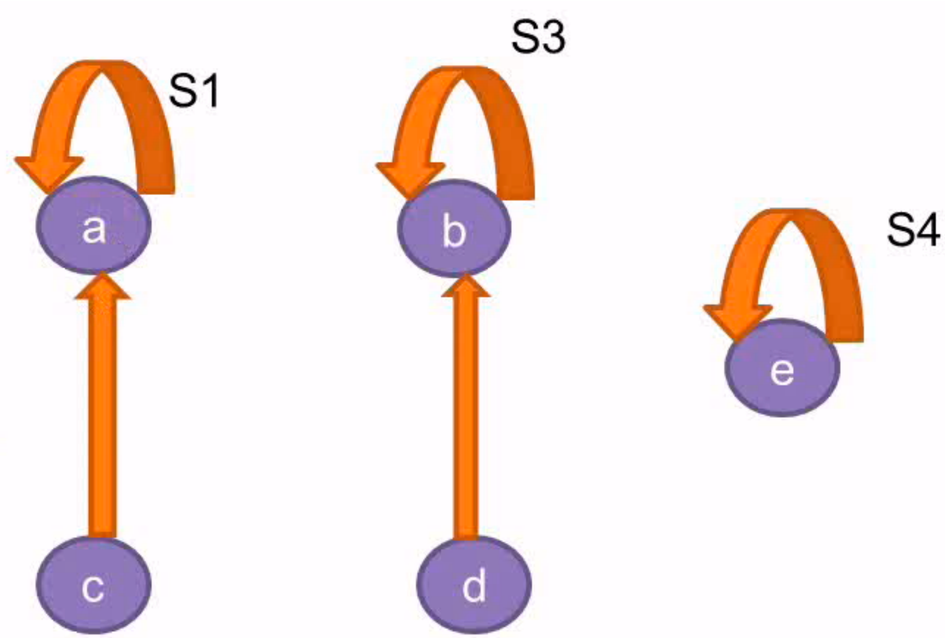
Find(d) => S3 or b
Union(S2, S1)



Tree Based Disjoint Set

- Each set is a tree
- A set is identified by the root of the tree

Supported Operations:
Find()
Union()

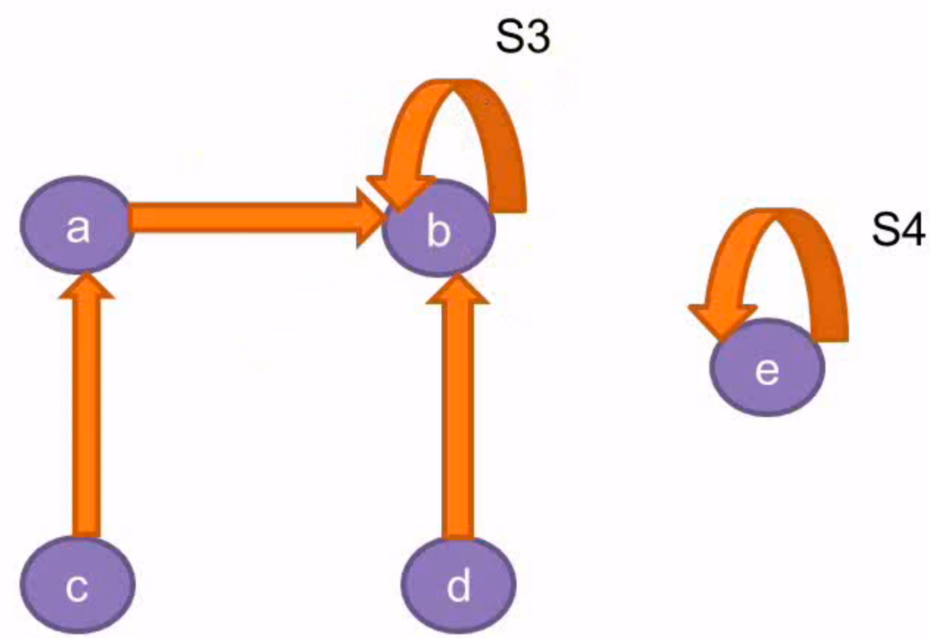


Union(S1, S3)

Tree Based Disjoint Set

- Each set is a tree
- A set is identified by the root of the tree

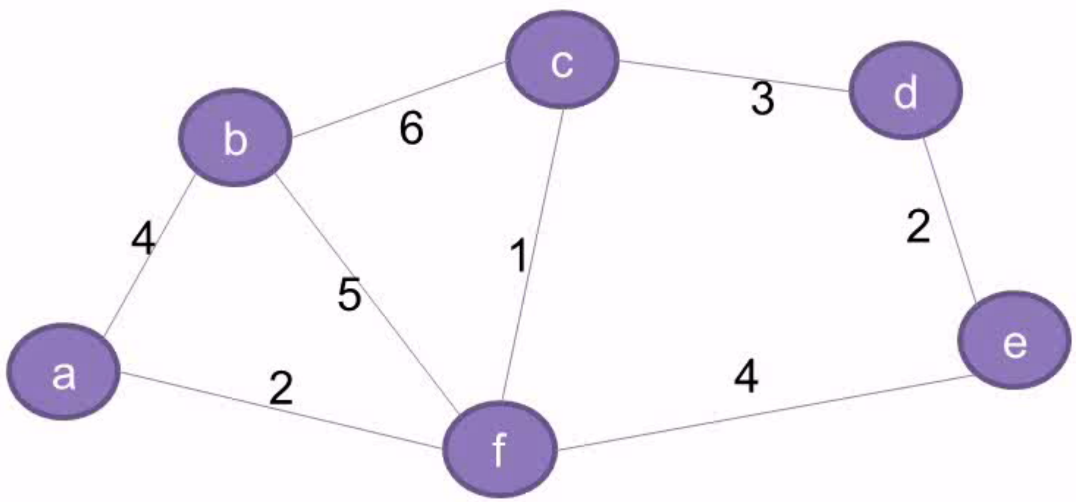
Supported Operations:
Find()
Union()



Complexity:
Find() - O (depth)
Union() - O(1)

Kruskal’s algorithm

Uses the disjoint-set data structure



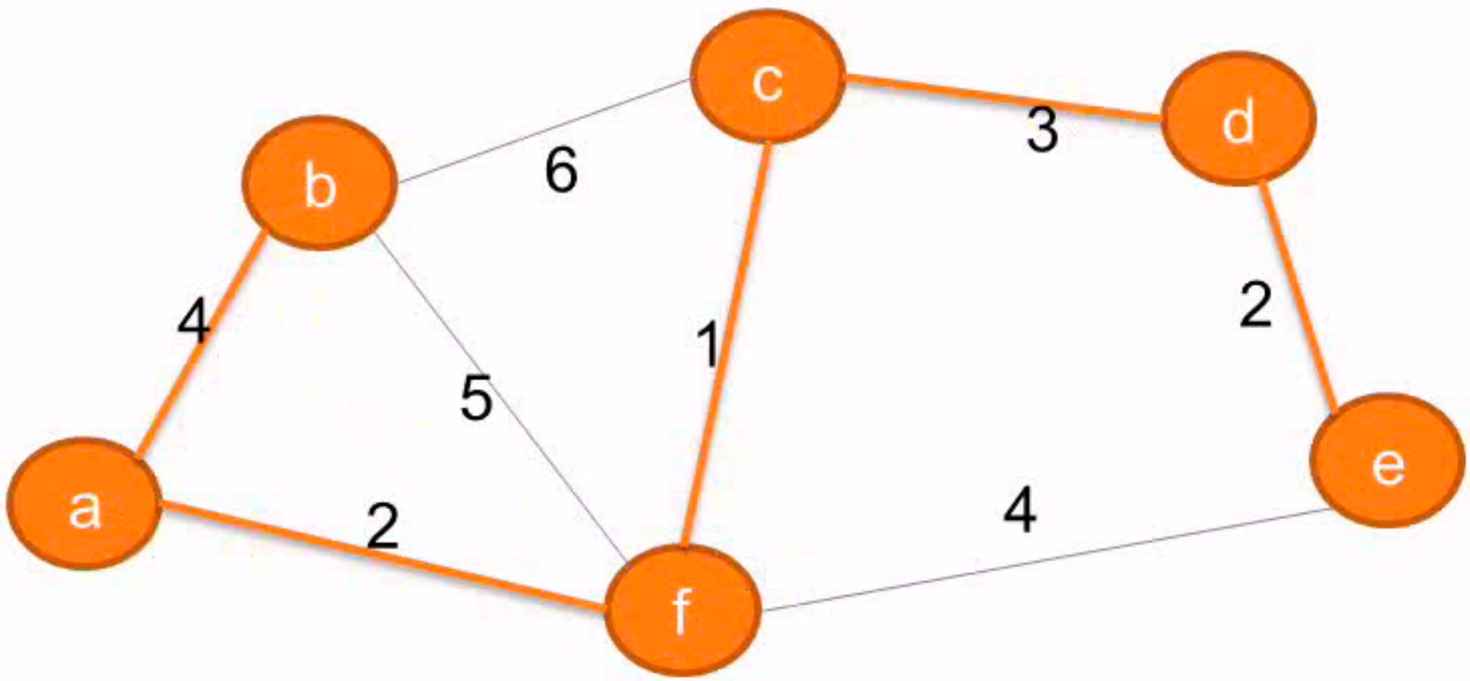
A = { }

Kruskal(V, E)

A = ∅
foreach v ∈ V:
    Make-disjoint-set(v)
Sort E by weight increasingly
foreach (v1, v2) ∈ E:
    if Find(v1) ≠ Find(v2):
        A = A ∪ {(v1, v2)}
        Union(v1, v2)
return A

Running time = O(ElgV)

Kruskal's algorithm



A = { (c, f), (a, f), (d, e), (c, d), (a, b) }

Next Lecture

- Shortest path
  - Single-source shortest paths
  - All-pairs shortest paths