APA: Advanced Programming, Algorithms and Data Structures

Emre Güney, PhD

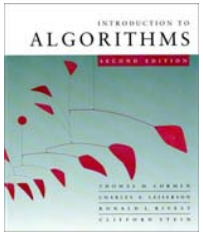
Master in Bioinformatics for Health Sciences
Universitat Pompeu Fabra

Lectures 9-10

October 29-31, 2019

Previously on APA

- Computational complexity
  - Correctness of an algorithm
  - Efficiency of an algorithm
  - Time complexity in terms of input size
  - Space complexity
- Sorting
  - Buble sort, selection sort, insertion sort, merge sort
  - Exhaustive search (brute force) vs divide & conquer
  - Quick sort
  - Heap sort
- Data structures
  - Linked lists, stacks, queues
  - Priority queues and heaps
  - Trees, binary search trees
  - Balanced trees (AVL trees)
- Graphs
  - BFS, DFS
  - Cycles, connected components
  - DAGs, topological sort

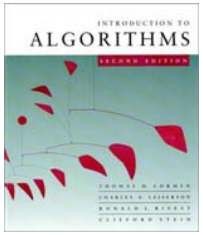


Shortest paths

A ***shortest path*** from u to v is a path of minimum weight from u to v. The ***shortest-path weight*** from u to v is defined as

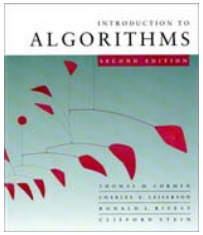
$$\delta(u, v) = \min\{w(p) : p \text{ is a path from } u \text{ to } v\}.$$

Note: $\delta(u, v) = \infty$ if no path from u to v exists.



Dijkstra's algorithm

$d[s] \leftarrow 0$
for each $v \in V - \{s\}$
  do $d[v] \leftarrow \infty$
$S \leftarrow \varnothing$
$Q \leftarrow V$ ▷ Q is a priority queue maintaining $V - S$
while $Q \neq \varnothing$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    $S \leftarrow S \cup \{u\}$
    for each $v \in Adj[u]$
      do if $d[v] > d[u] + w(u, v)$
        then $d[v] \leftarrow d[u] + w(u, v)$



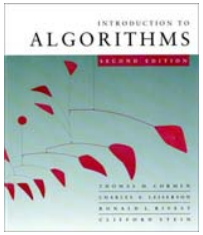
Correctness — Part I

Lemma. Initializing $d[s] \leftarrow 0$ and $d[v] \leftarrow \infty$ for all $v \in V - \{s\}$ establishes $d[v] \geq \delta(s, v)$ for all $v \in V$, and this invariant is maintained over any sequence of relaxation steps.

Proof. Suppose not. Let v be the first vertex for which $d[v] < \delta(s, v)$, and let u be the vertex that caused $d[v]$ to change: $d[v] = d[u] + w(u, v)$. Then,

$$
\begin{aligned}
d[v] &< \delta(s, v) && \text{supposition} \\
&\leq \delta(s, u) + \delta(u, v) && \text{triangle inequality} \\
&\leq \delta(s,u) + w(u, v) && \text{sh. path} \leq \text{specific path} \\
&\leq d[u] + w(u, v) && v \text{ is first violation}
\end{aligned}
$$

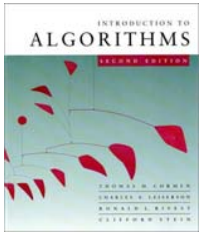
Contradiction. □



Correctness — Part II

Lemma. Let u be v's predecessor on a shortest path from s to v. Then, if $d[u] = \delta(s, u)$ and edge (u, v) is relaxed, we have $d[v] = \delta(s, v)$ after the relaxation.

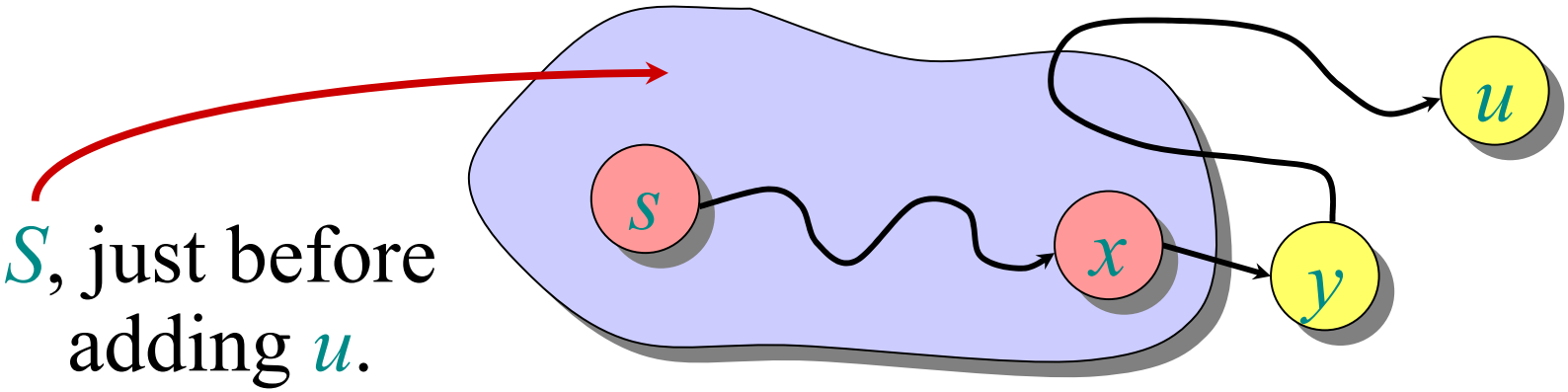
Proof. Observe that $\delta(s, v) = \delta(s, u) + w(u, v)$. Suppose that $d[v] > \delta(s, v)$ before the relaxation. (Otherwise, we're done.) Then, the test $d[v] > d[u] + w(u, v)$ succeeds, because $d[v] > \delta(s, v) = \delta(s, u) + w(u, v) = d[u] + w(u, v)$, and the algorithm sets $d[v] = d[u] + w(u, v) = \delta(s, v)$. □

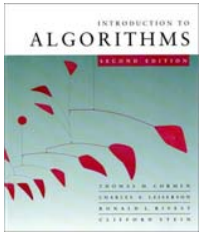


Correctness — Part III

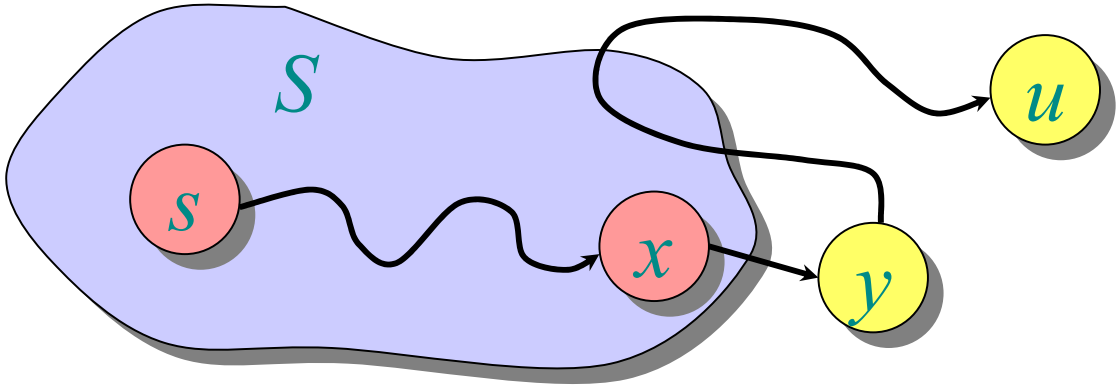
Theorem. Dijkstra's algorithm terminates with $d[v] = \delta(s, v)$ for all $v \in V$.

Proof. It suffices to show that $d[v] = \delta(s, v)$ for every $v \in V$ when v is added to S. Suppose u is the first vertex added to S for which $d[u] > \delta(s, u)$. Let y be the first vertex in $V - S$ along a shortest path from s to u, and let x be its predecessor:





Correctness — Part III (continued)

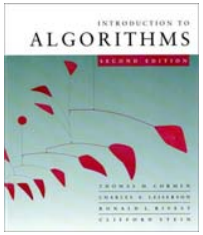


Since u is the first vertex violating the claimed invariant, we have $d[x] = \delta(s, x)$. When x was added to S, the edge (x, y) was relaxed, which implies that $d[y] = \delta(s, y) \leq \delta(s, u) < d[u]$. But, $d[u] \leq d[y]$ by our choice of u. Contradiction. □

Widely used algorithm design techniques

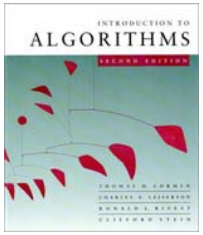
- Exhaustive Search
  - Enumerate all possible solutions
- Divide-and-Conquer Algorithms
  - Break problem into pieces which are easier to solve, then combine.
- Greedy Algorithms
  - Make the locally optimal choice to reach a globally optimum solution
- Dynamic Programming
  - Build up solutions for larger problems from smaller ones
- Branch-and-Bound Algorithms
  - Eliminate search paths that will not improve
- Machine Learning
  - Collect statistics over time and use these to solve current problem
- Randomized Algorithms
  - Make random decisions (to overcome certain worst-case scenarios)

Adapted from: Daniel Lopresti lecture notes for Lehigh Uni. CSE 308-408



Hallmark for "greedy" algorithms

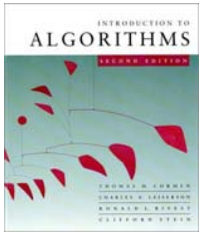
Greedy-choice property
A locally optimal choice is globally optimal.



Dynamic-programming hallmark #1

Optimal substructure

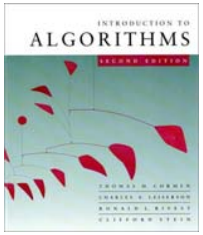
An optimal solution to a problem (instance) contains optimal solutions to subproblems.



Dynamic-programming hallmark #2

Overlapping subproblems

A recursive solution contains a "small" number of distinct subproblems repeated many times.



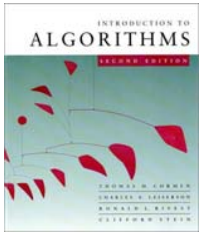
Memoization algorithm

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

More insights on dynamic programming

Dynamic Programming slides (8 to 48) from colt_steele
(click to access)

Source: cs.slides.com, accessed on Oct 2019



Dynamic programming

Consider the $n \times n$ adjacency matrix $A = (a_{ij})$ of the digraph, and define

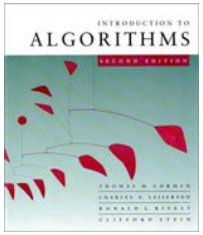
$d_{ij}^{(m)}$ = weight of a shortest path from i to j that uses at most m edges.

Claim: We have

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j, \\ \infty & \text{if } i \neq j; \end{cases}$$

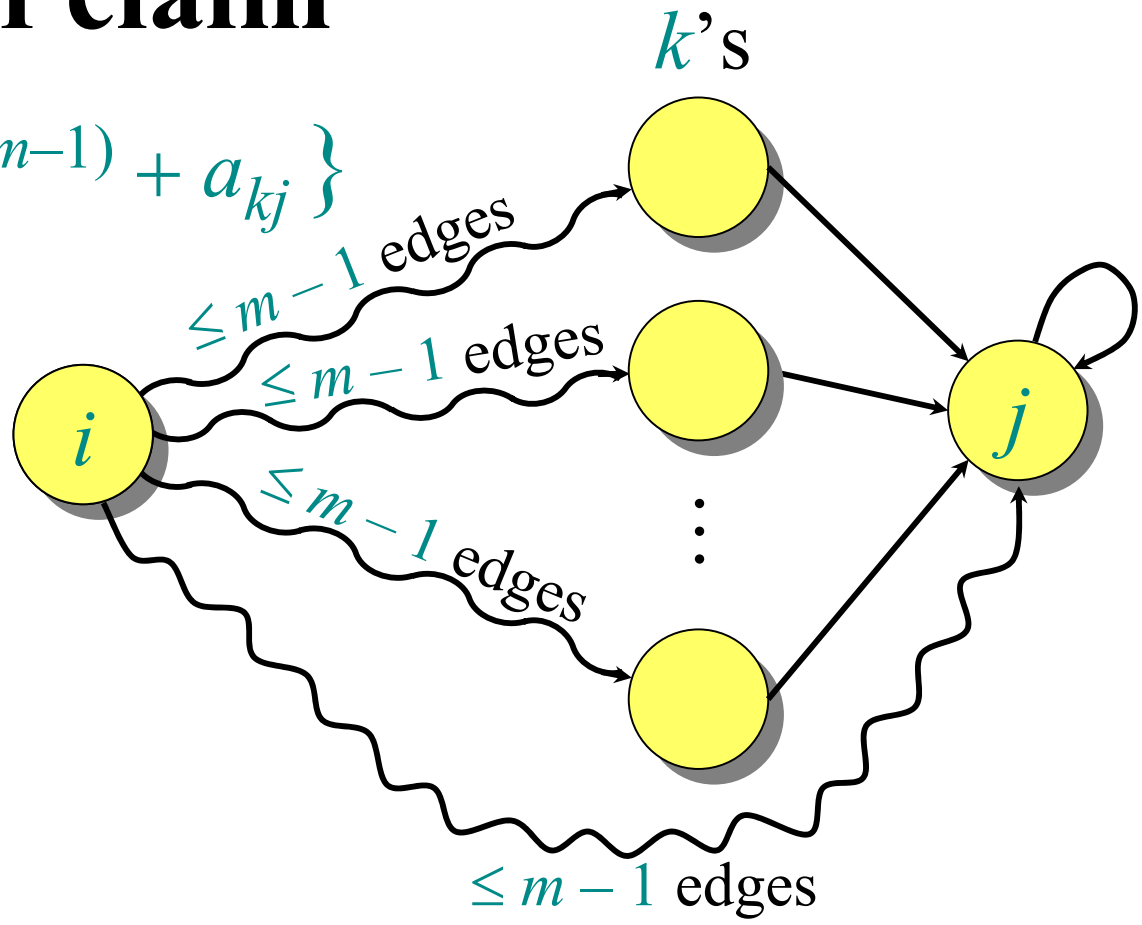
and for $m = 1, 2, \ldots, n-1$,

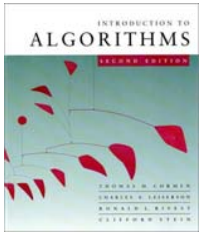
$$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}.$$



Proof of claim

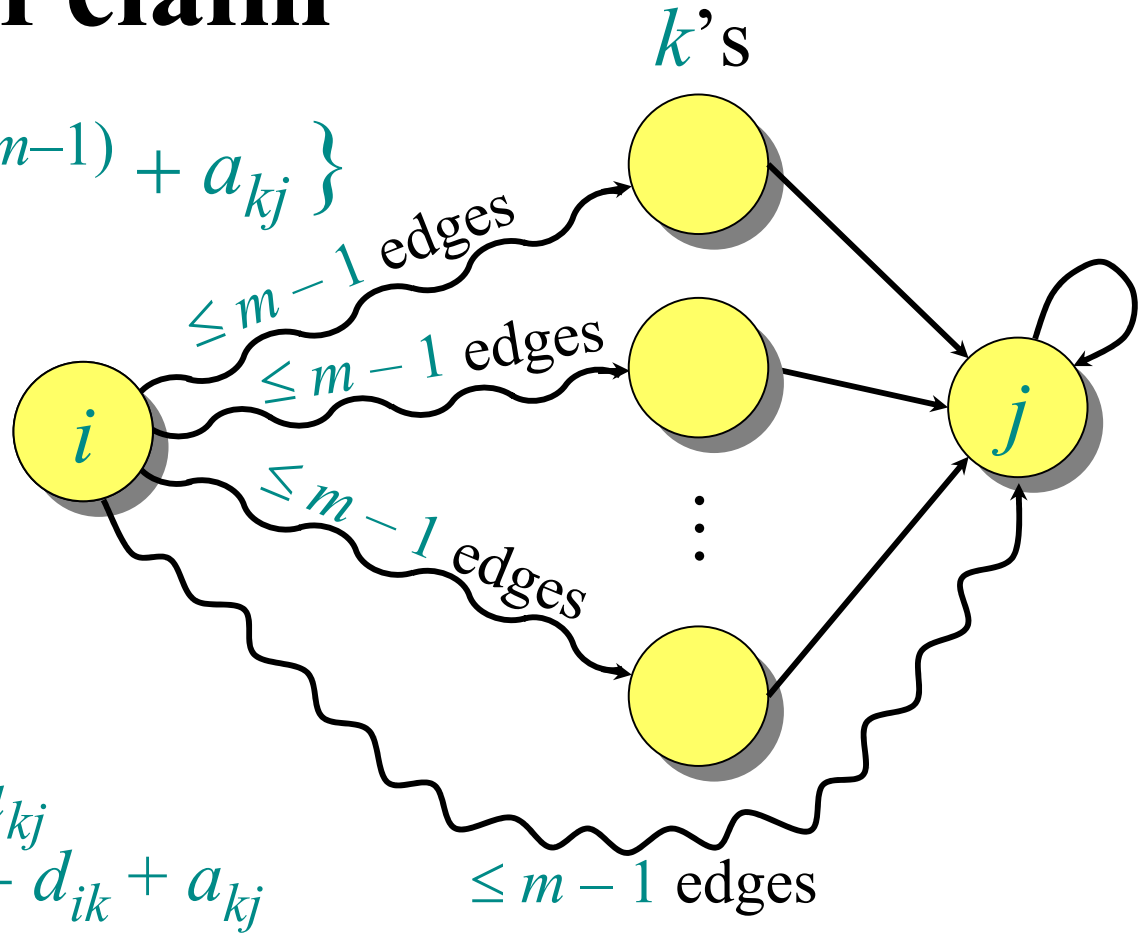
$$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}$$





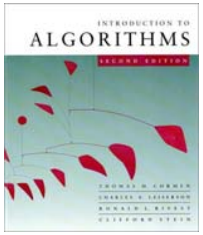
Proof of claim

$$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}$$



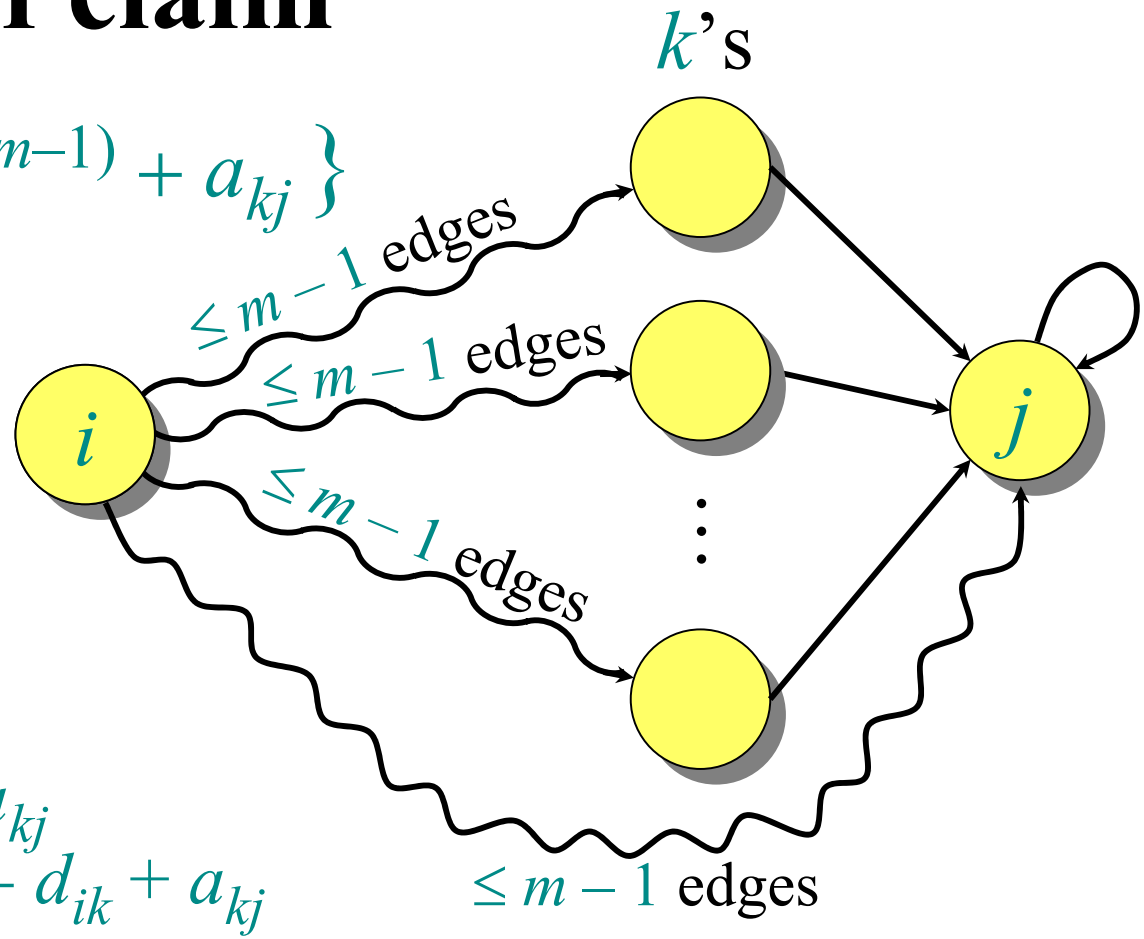
Relaxation!

for $k \leftarrow 1$ **to** n
  do if $d_{ij} > d_{ik} + a_{kj}$
    then $d_{ij} \leftarrow d_{ik} + a_{kj}$



Proof of claim

$$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}$$

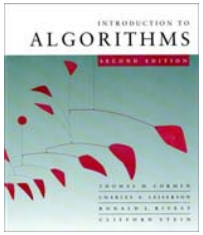


Relaxation!

for $k \leftarrow 1$ **to** n
  do if $d_{ij} > d_{ik} + a_{kj}$
    then $d_{ij} \leftarrow d_{ik} + a_{kj}$

Note: No negative-weight cycles implies

$$\delta(i,j) = d_{ij}^{(n-1)} = d_{ij}^{(n)} = d_{ij}^{(n+1)} = \ldots$$



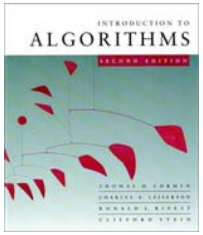
Minimum spanning trees

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

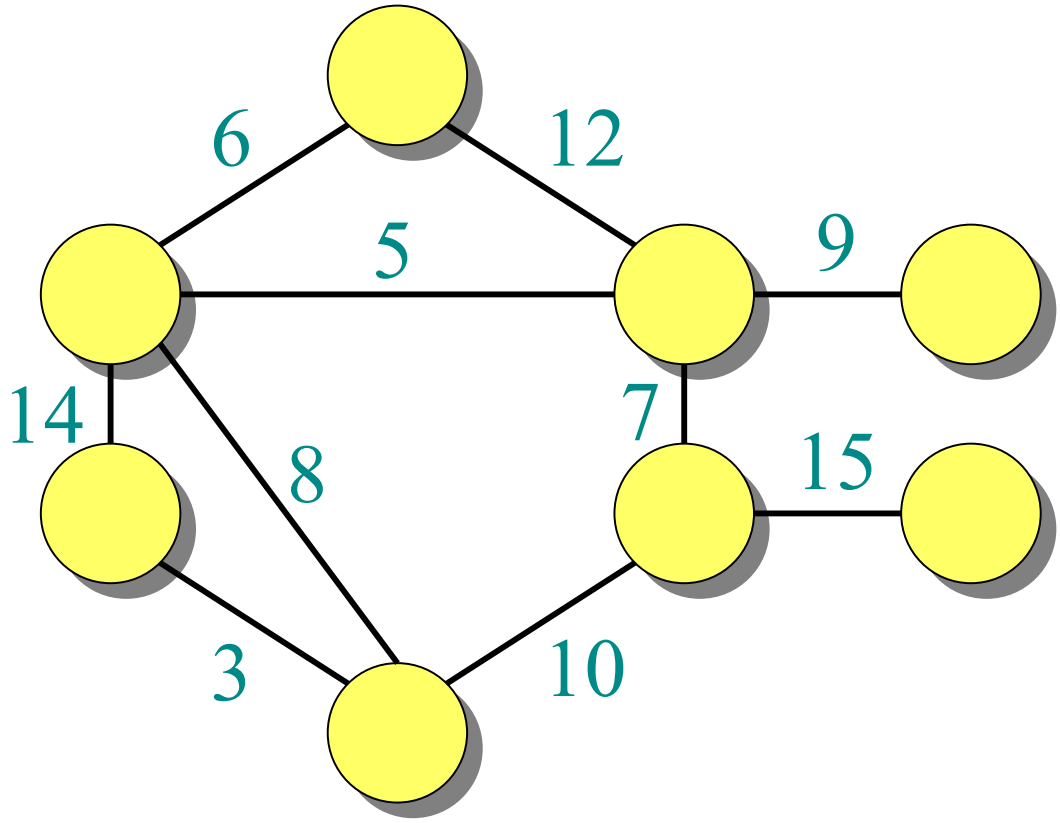
- For simplicity, assume that all edge weights are distinct. (CLRS covers the general case.)

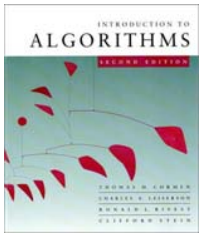
Output: A ***spanning tree*** T — a tree that connects all vertices — of minimum weight:

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

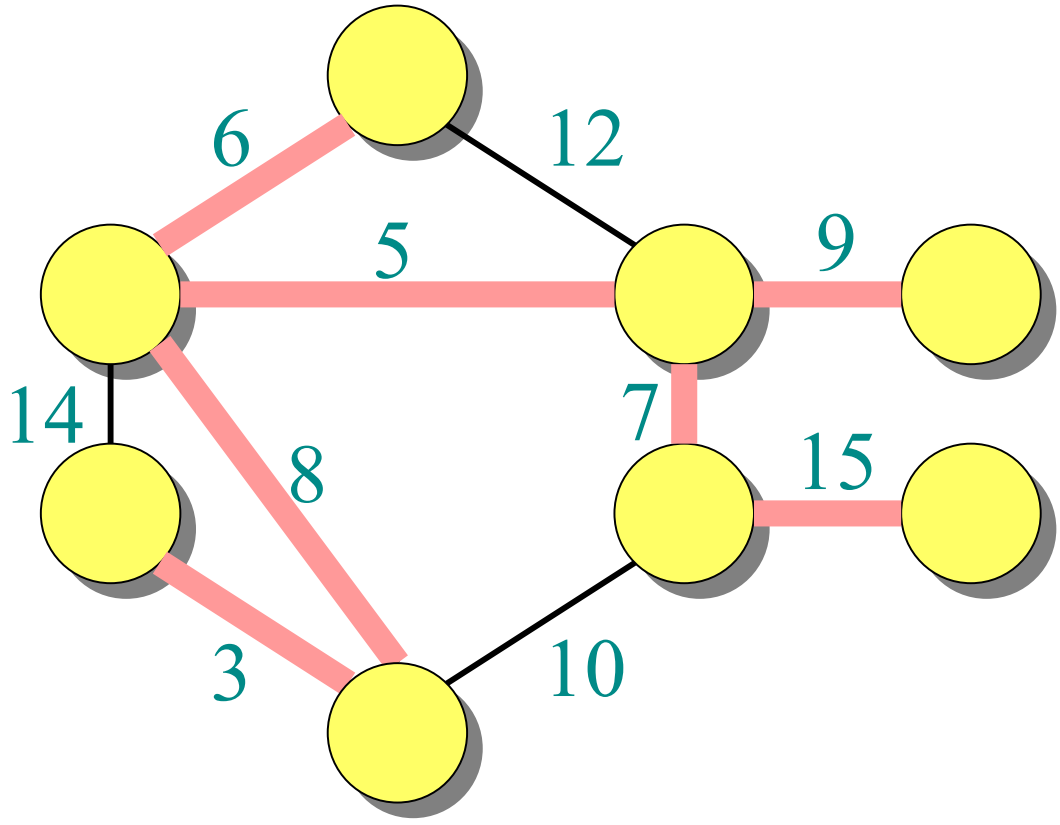


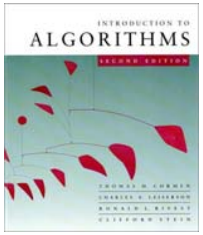
Example of MST





Example of MST

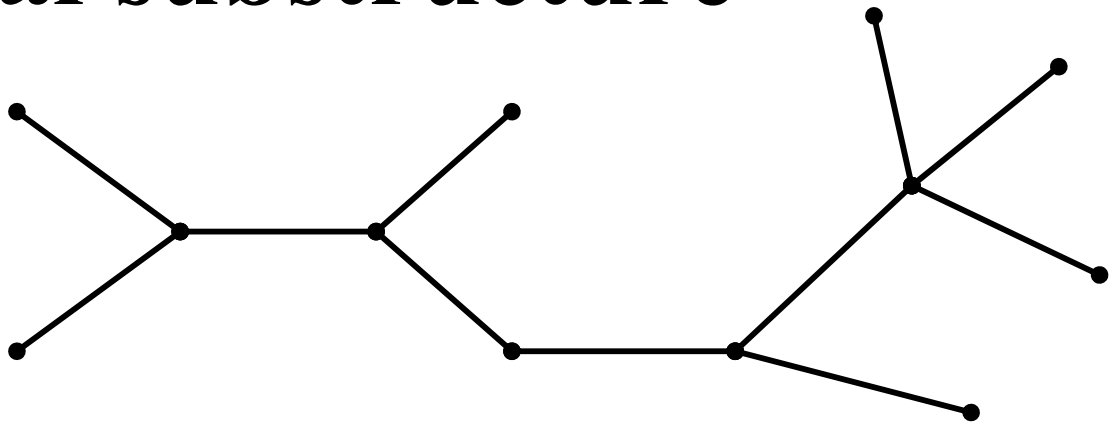


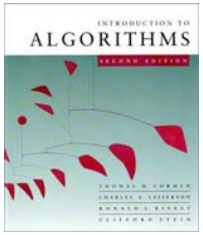


Optimal substructure

MST T:

(Other edges of G are not shown.)

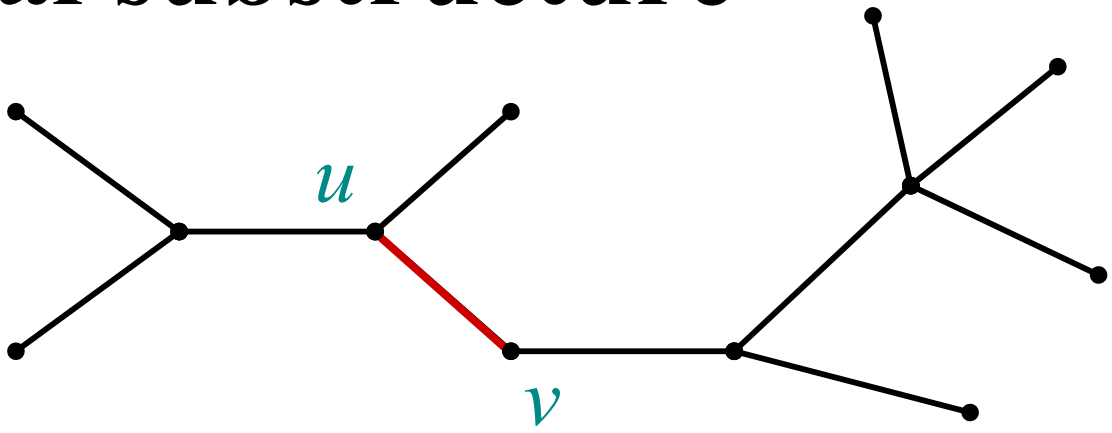




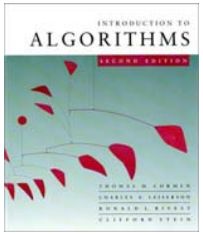
Optimal substructure

MST T:

(Other edges of G are not shown.)



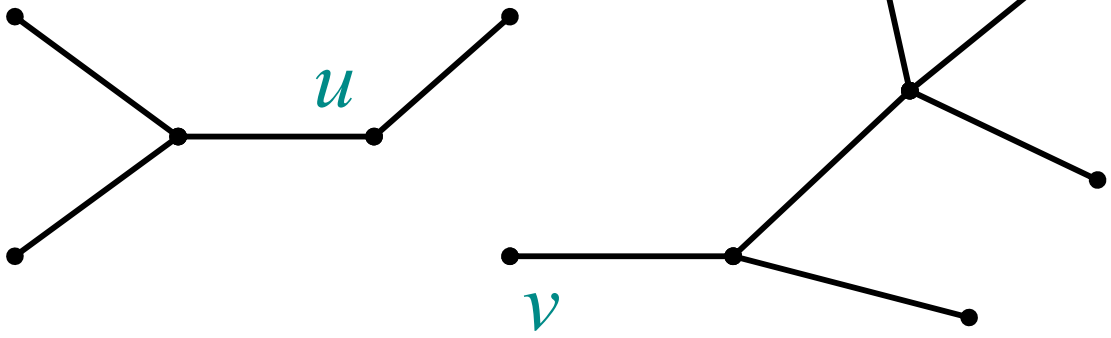
Remove any edge $(u, v) \in T$.



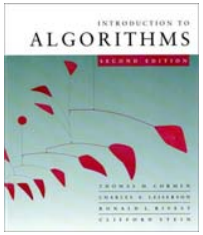
Optimal substructure

MST T:

(Other edges of G are not shown.)



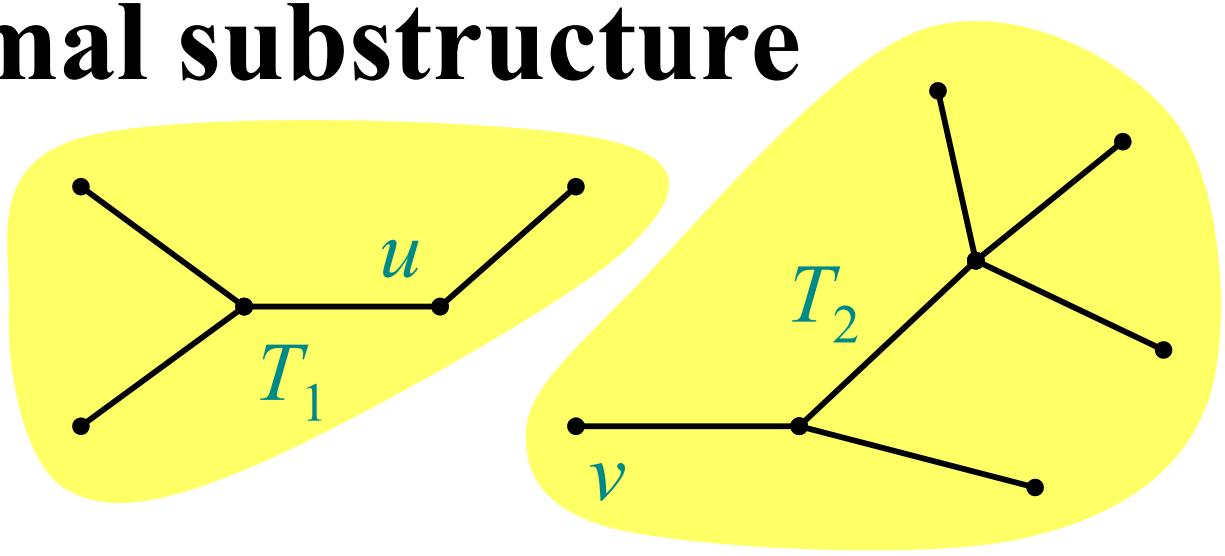
Remove any edge $(u, v) \in T$.



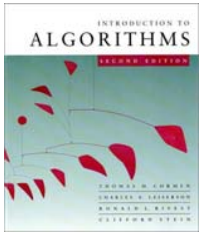
Optimal substructure

MST T:

(Other edges of G are not shown.)



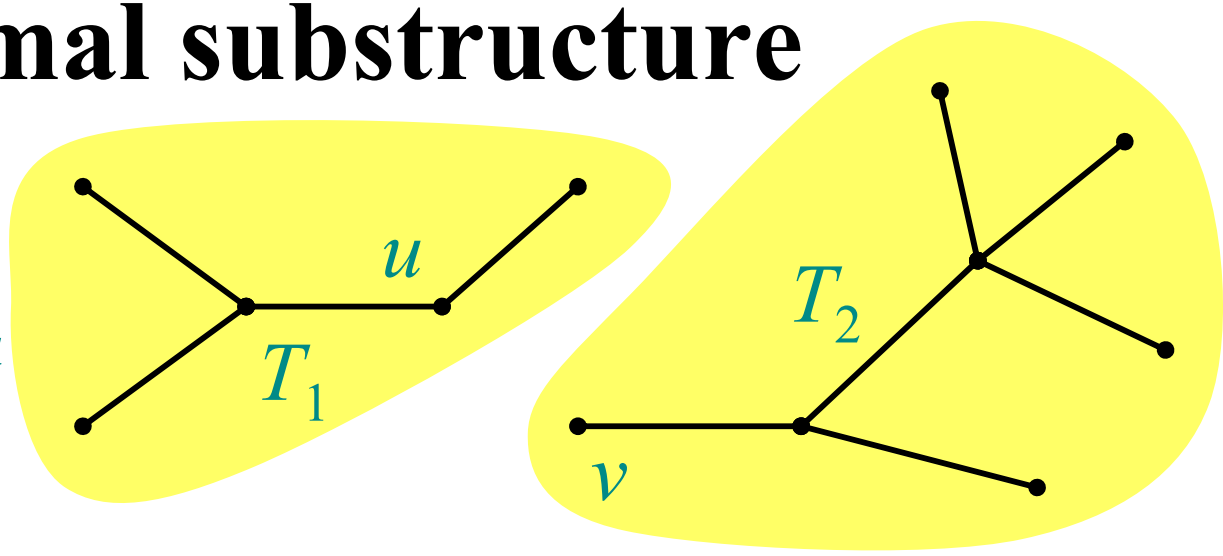
Remove any edge $(u, v) \in T$. Then, T is partitioned into two subtrees T_1 and T_2.



Optimal substructure

MST T:

(Other edges of G are not shown.)

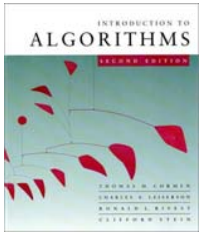


Remove any edge $(u, v) \in T$. Then, T is partitioned into two subtrees T_1 and T_2.

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.

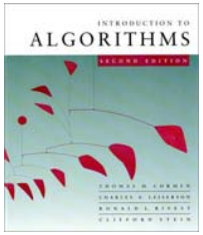


Proof of optimal substructure

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. □



Proof of optimal substructure

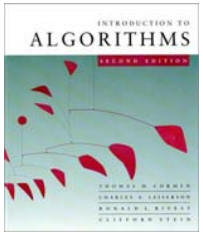
Proof. Cut and paste:

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

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. □

Do we also have overlapping subproblems?

- Yes.



Proof of optimal substructure

Proof. Cut and paste:

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

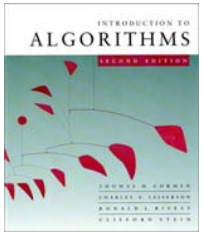
If T_1' were a lower-weight spanning tree than T_1 for G_1, then $T' = \{(u, v)\} \cup T_1' \cup T_2$ would be a lower-weight spanning tree than T for G. □

Do we also have overlapping subproblems?

- Yes.

Great, then dynamic programming may work!

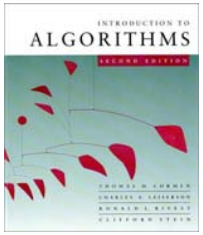
- Yes, but MST exhibits another powerful property which leads to an even more efficient algorithm.



Hallmark for "greedy" algorithms

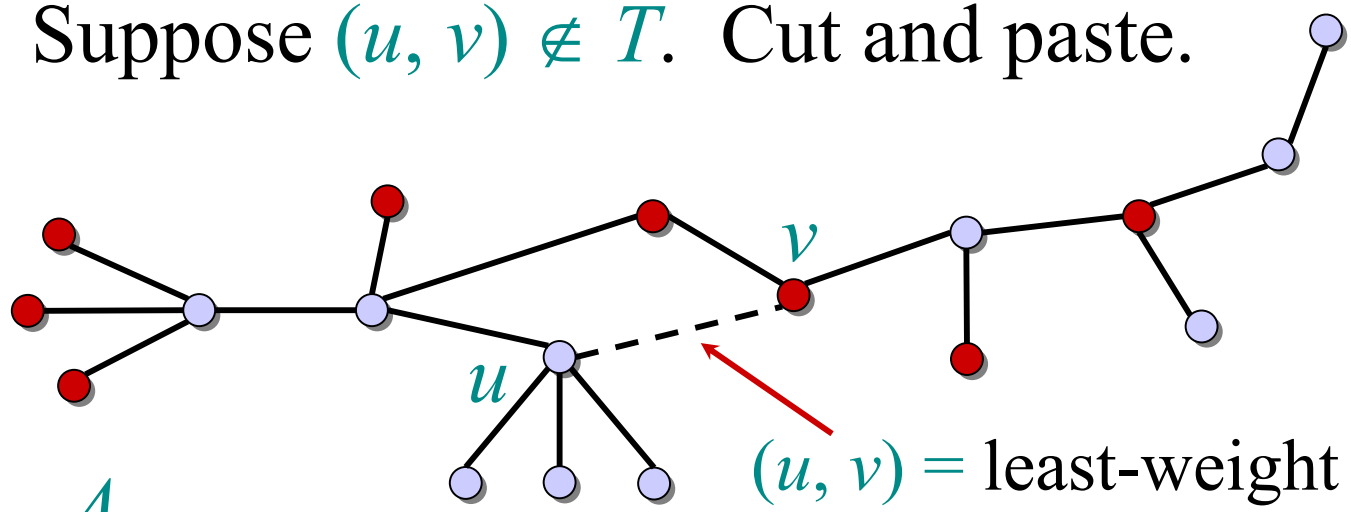
Greedy-choice property
A locally optimal choice is globally optimal.

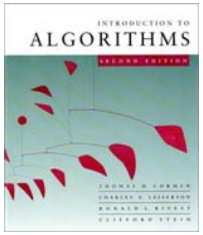
Theorem. Let T be the MST of $G = (V, E)$, and let $A \subseteq V$. Suppose that $(u, v) \in E$ is the least-weight edge connecting A to $V - A$. Then, $(u, v) \in T$.



Proof of theorem

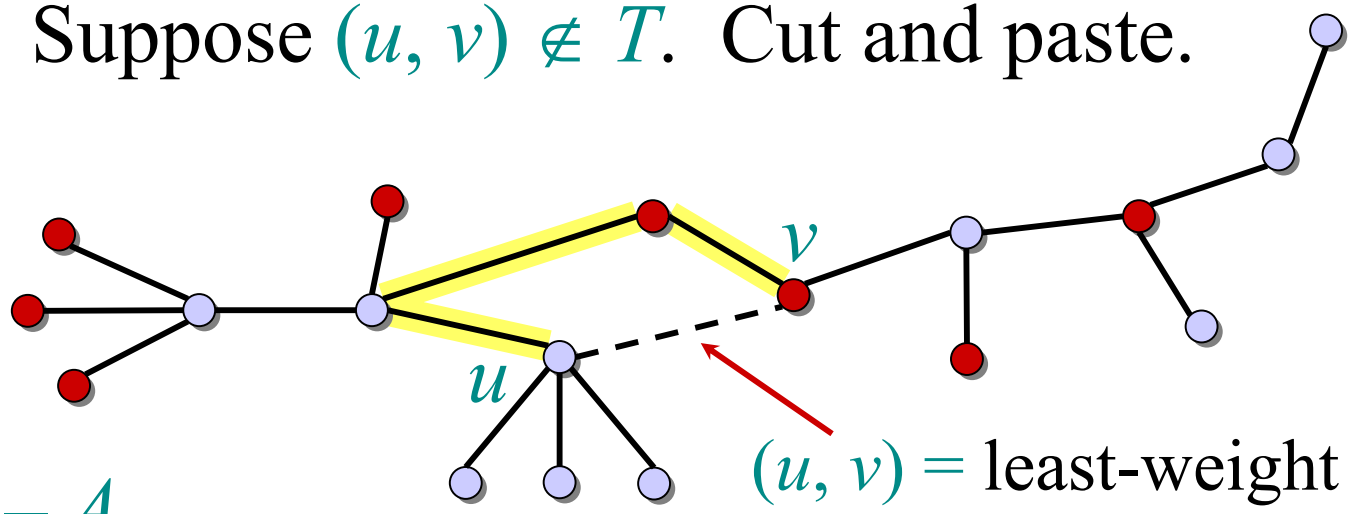
Proof. Suppose $(u, v) \notin T$. Cut and paste.



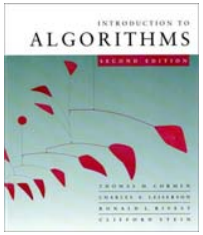


Proof of theorem

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

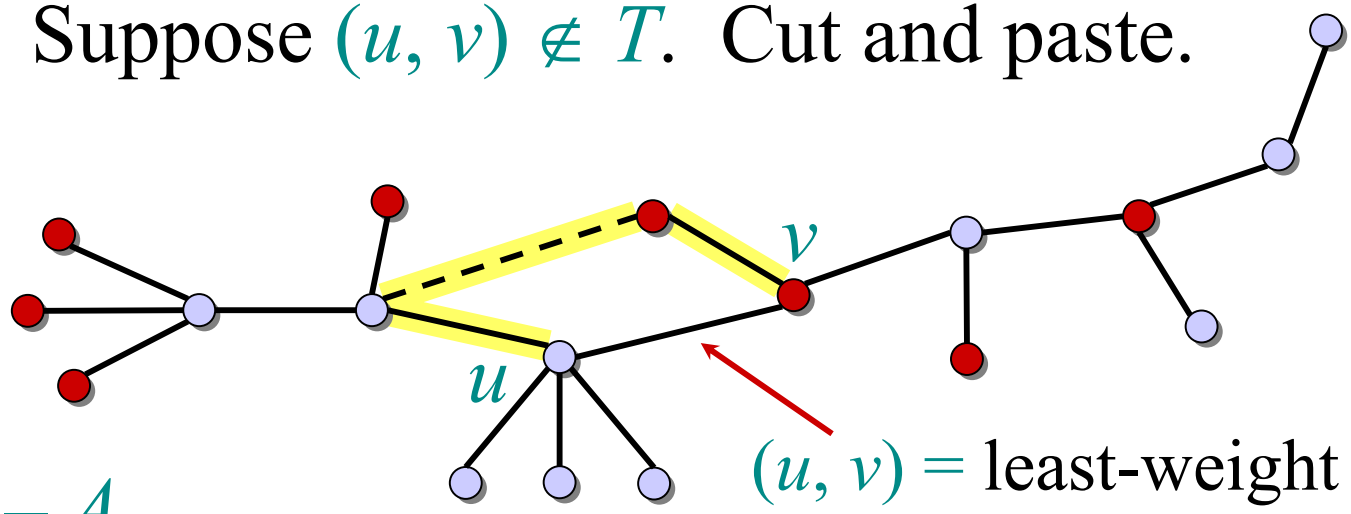


Consider the unique simple path from u to v in T.



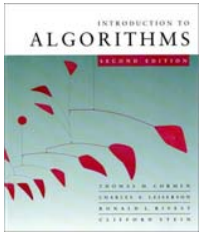
Proof of theorem

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



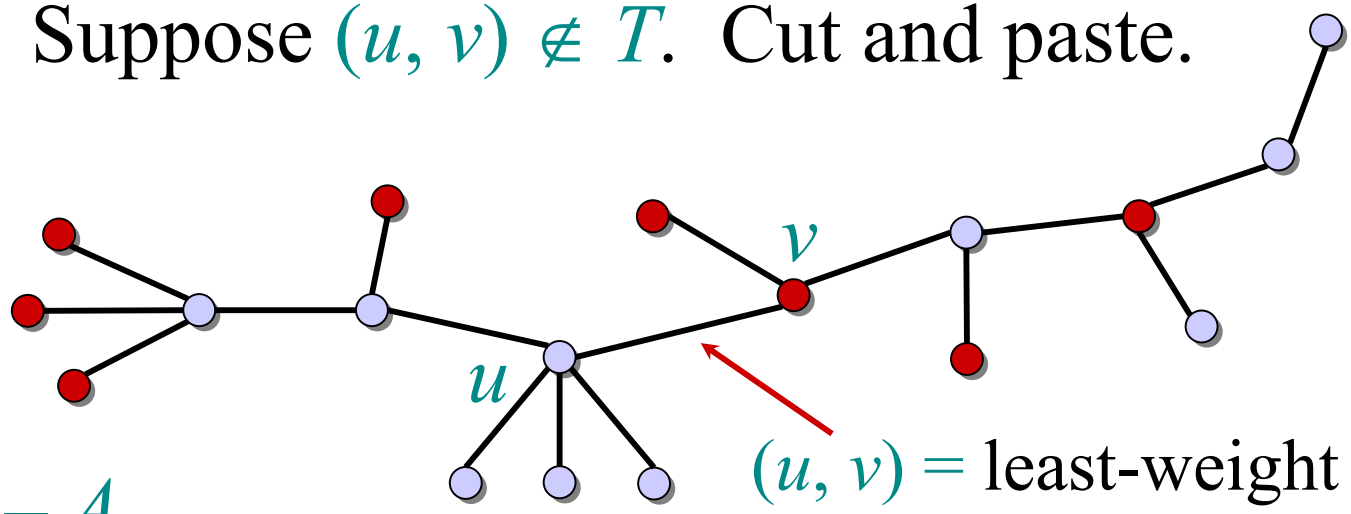
Consider the unique simple path from u to v in T.

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 $V - A$.

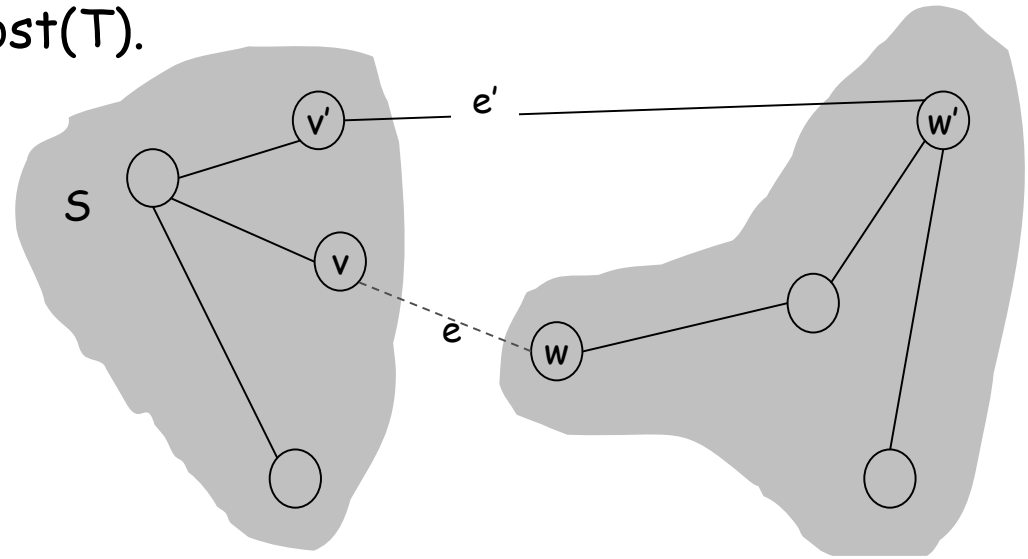
A lighter-weight spanning tree than T results. □

The cut property

Cut property. Let S be a subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST T contains e.

Proof. (exchange argument)

- If e =(v,w) is the only edge connecting S and V-S it must be in T, else e is on a cycle in the graph (not the MST). Now suppose e does not belong to T.
- Let e′= (v',w') be the first edge between S and V-S on the path from v'. T' = T ∪ { e } - { e' } is also a spanning tree.
- Since $c_e < c_{e'}$, cost(T') < cost(T).
- This is a contradiction. ▪

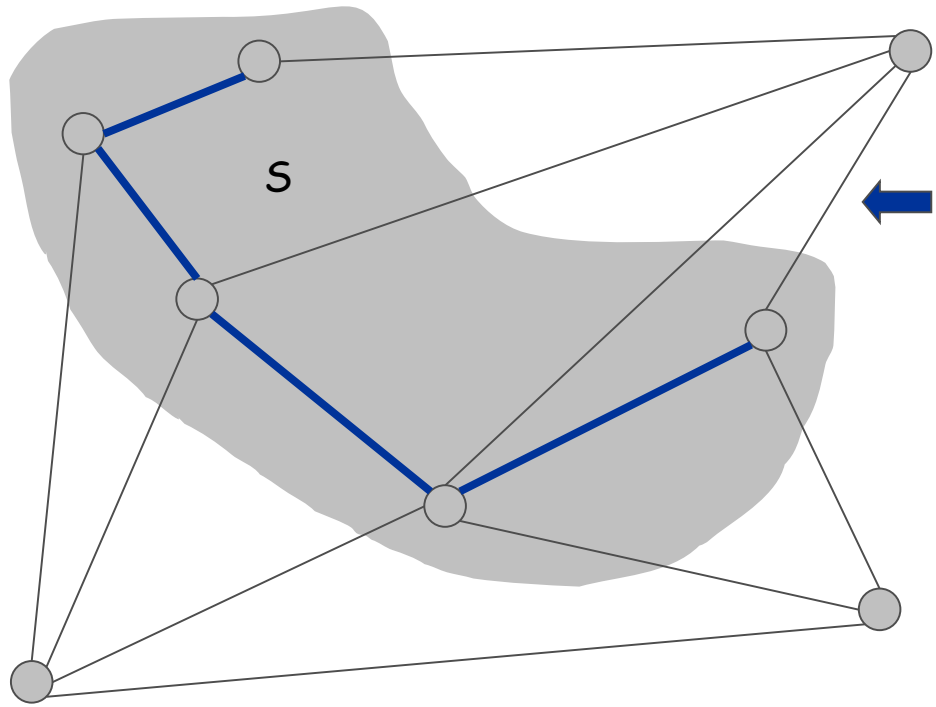


Source: ColoState - Wim Bohm

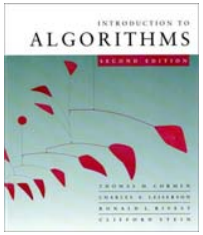
Prim's Algorithm

Prim's algorithm. [Jarník 1930, Prim 1957, Dijkstra 1959]

- Initialize S = any node.
- Apply cut property to S: add min cost edge (v, w) where v is in S and w is in V-S, and add w to S.
- Repeat until S = V.



Source: ColoState - Wim Bohm

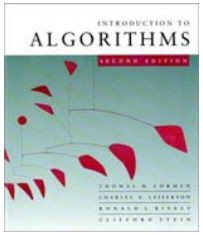


Prim's algorithm

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.

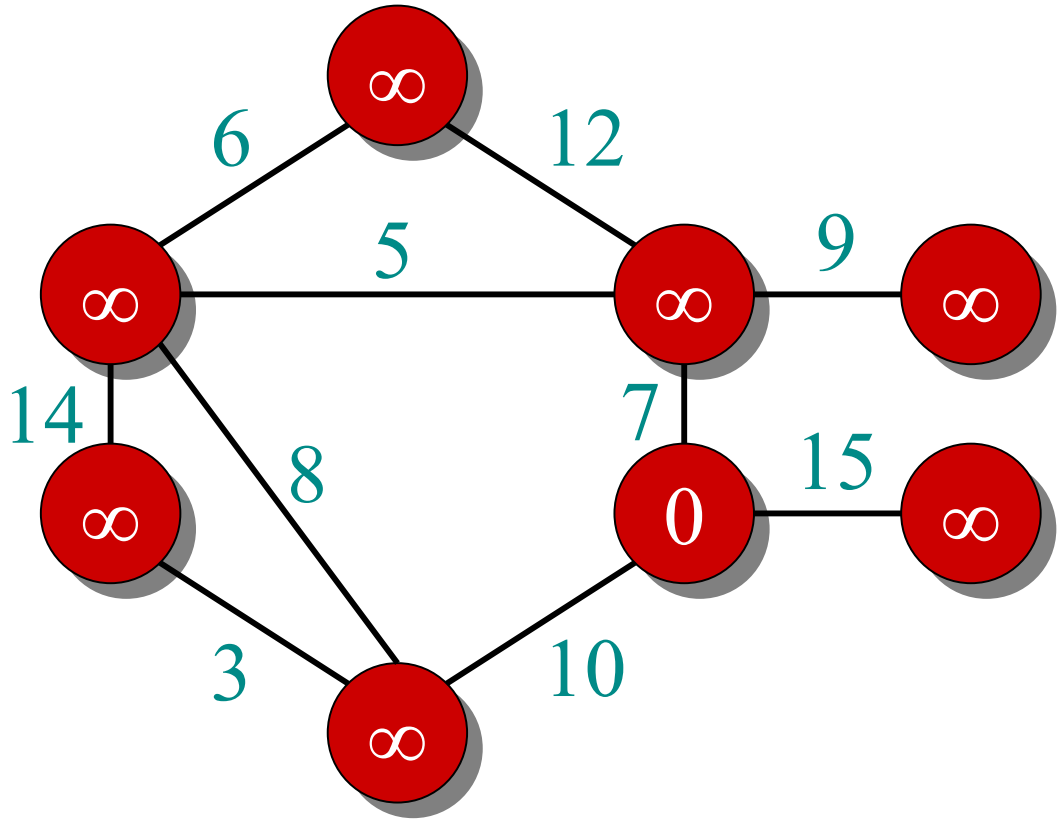
$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$
while $Q \neq \varnothing$
    do $u \leftarrow$ EXTRACT-MIN(Q)
        for each $v \in Adj[u]$
            do if $v \in Q$ and $w(u, v) < key[v]$
                then $key[v] \leftarrow w(u,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

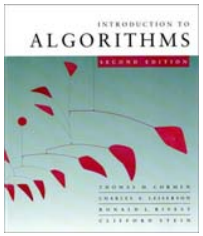
At the end, $\{(v, \pi[v])\}$ forms the MST.



Example of Prim's algorithm

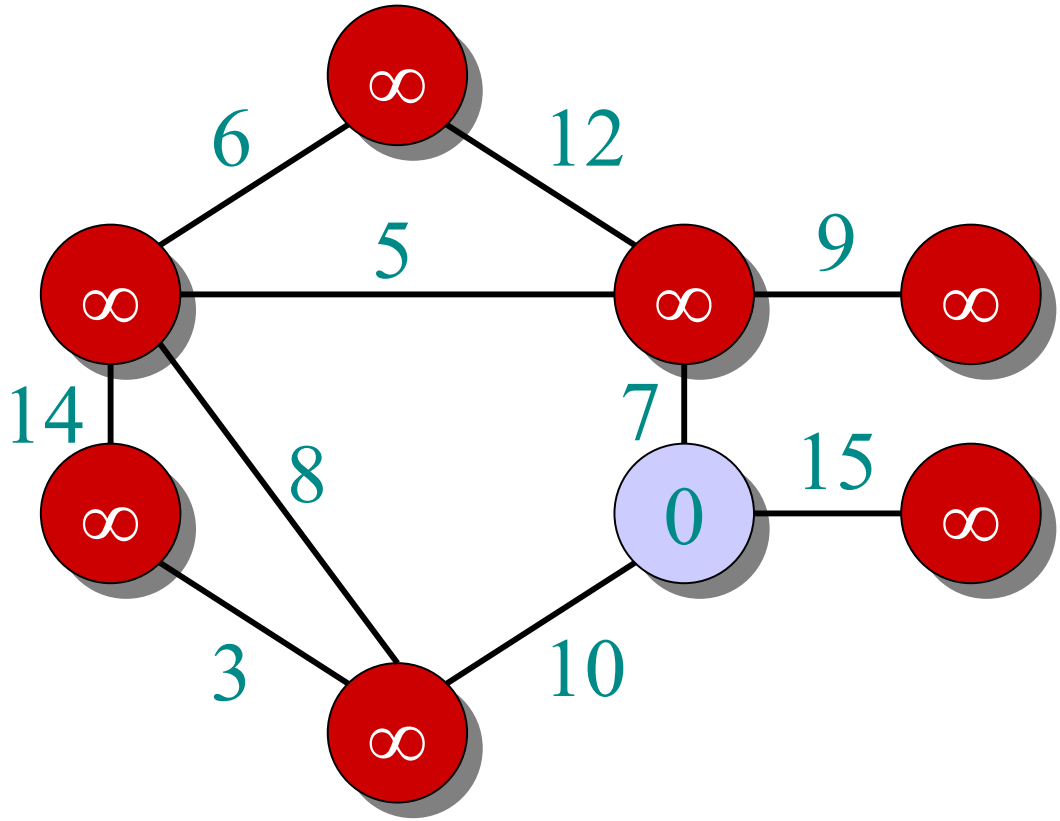
- $\in A$
- $\in V - A$

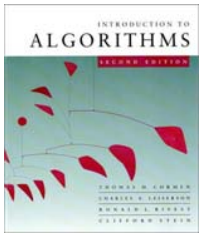




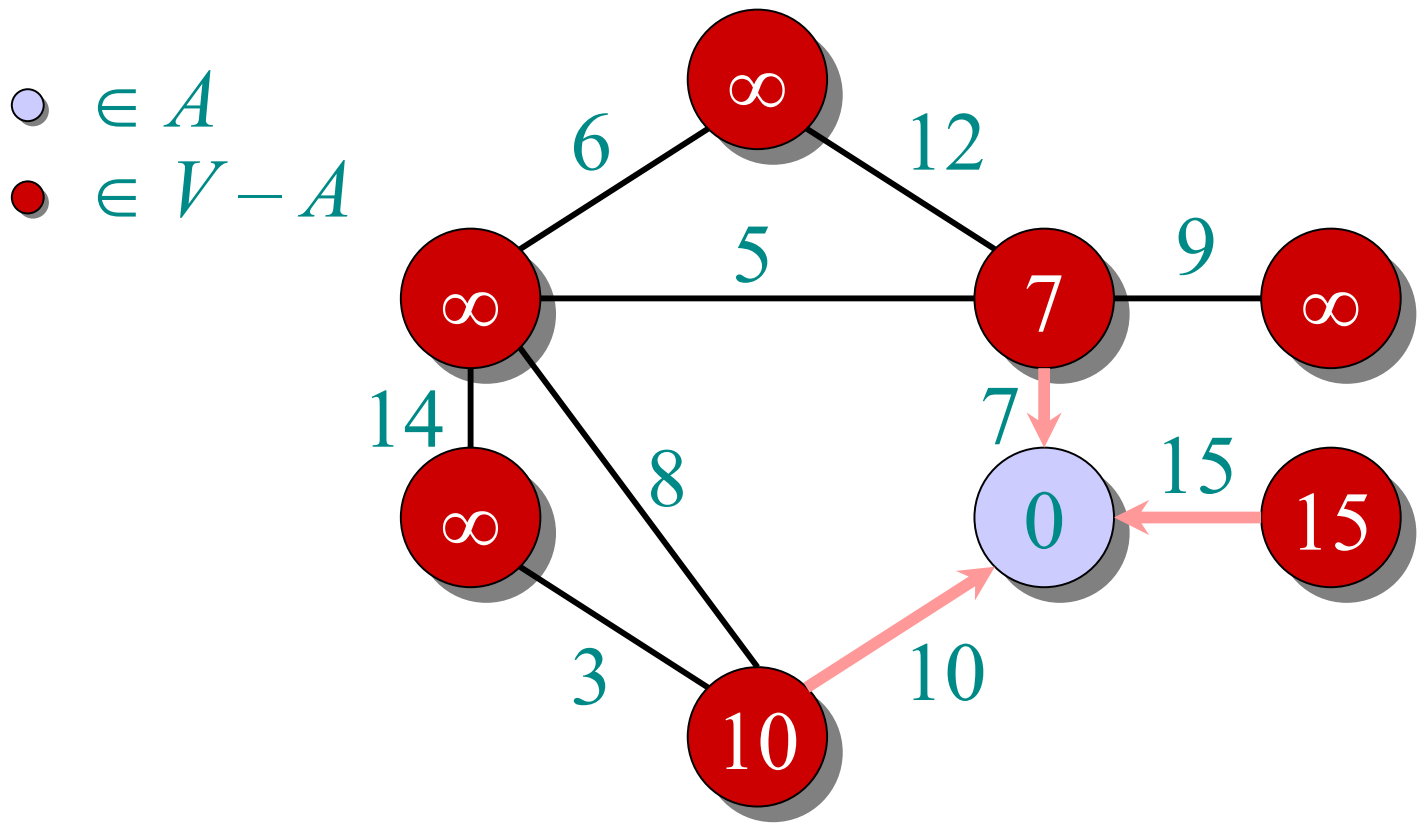
Example of Prim's algorithm

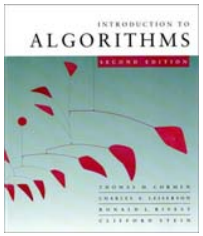
- $\in A$
- $\in V - A$



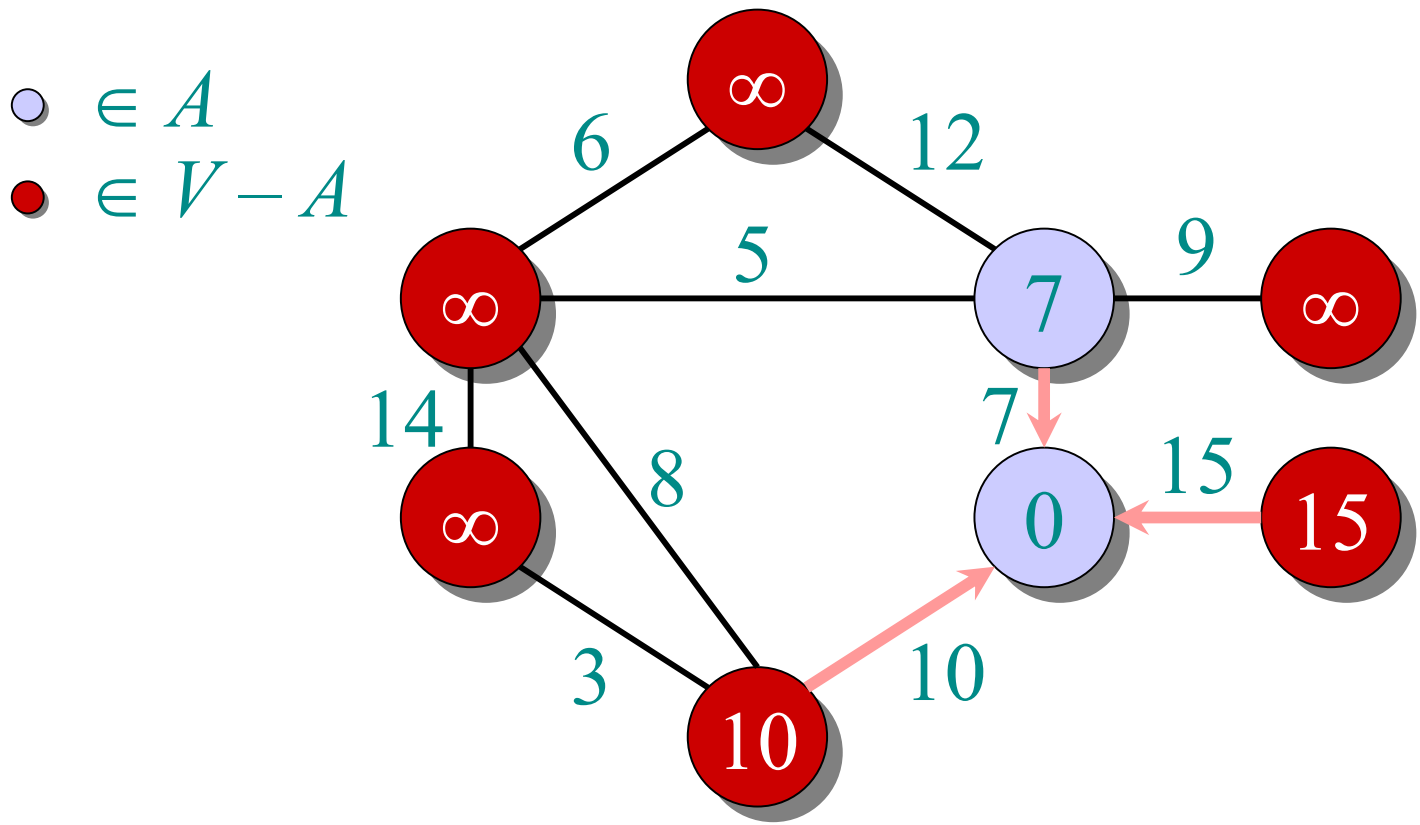


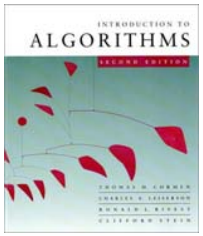
Example of Prim's algorithm





Example of Prim's algorithm

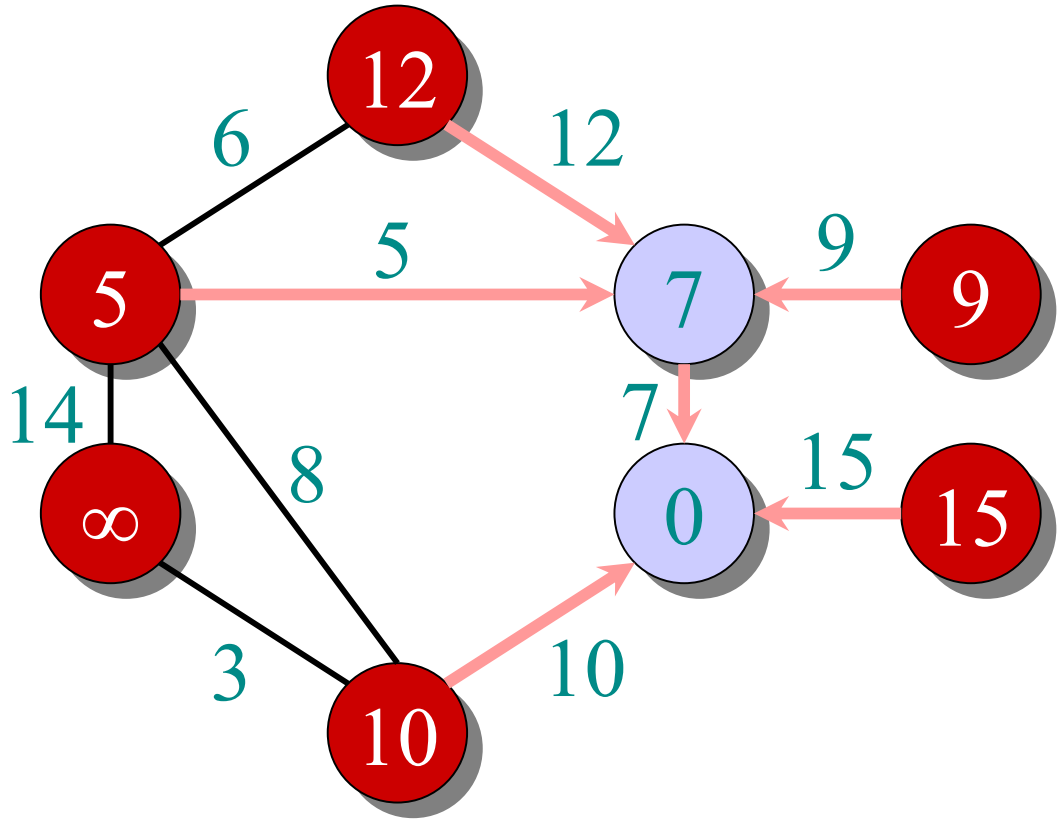


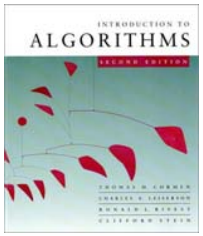


Example of Prim's algorithm

$\in A$

$\in V - A$

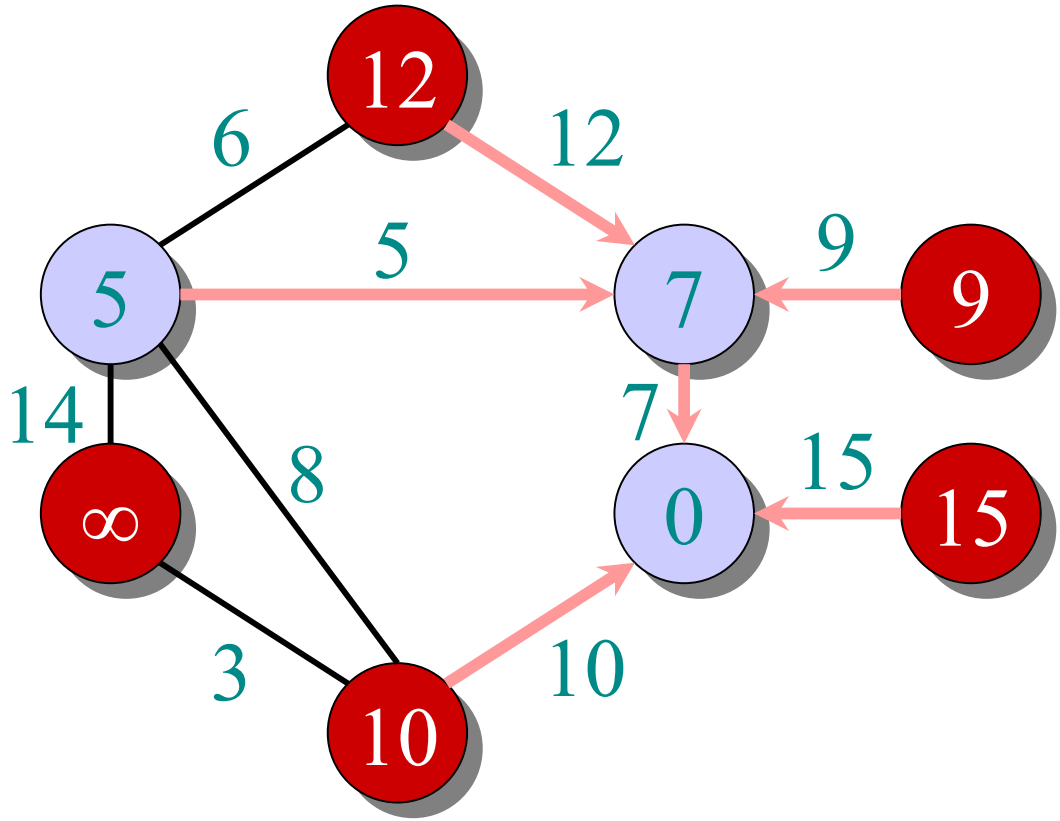


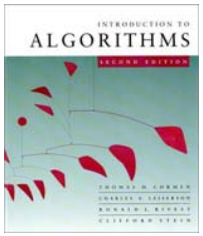


Example of Prim's algorithm

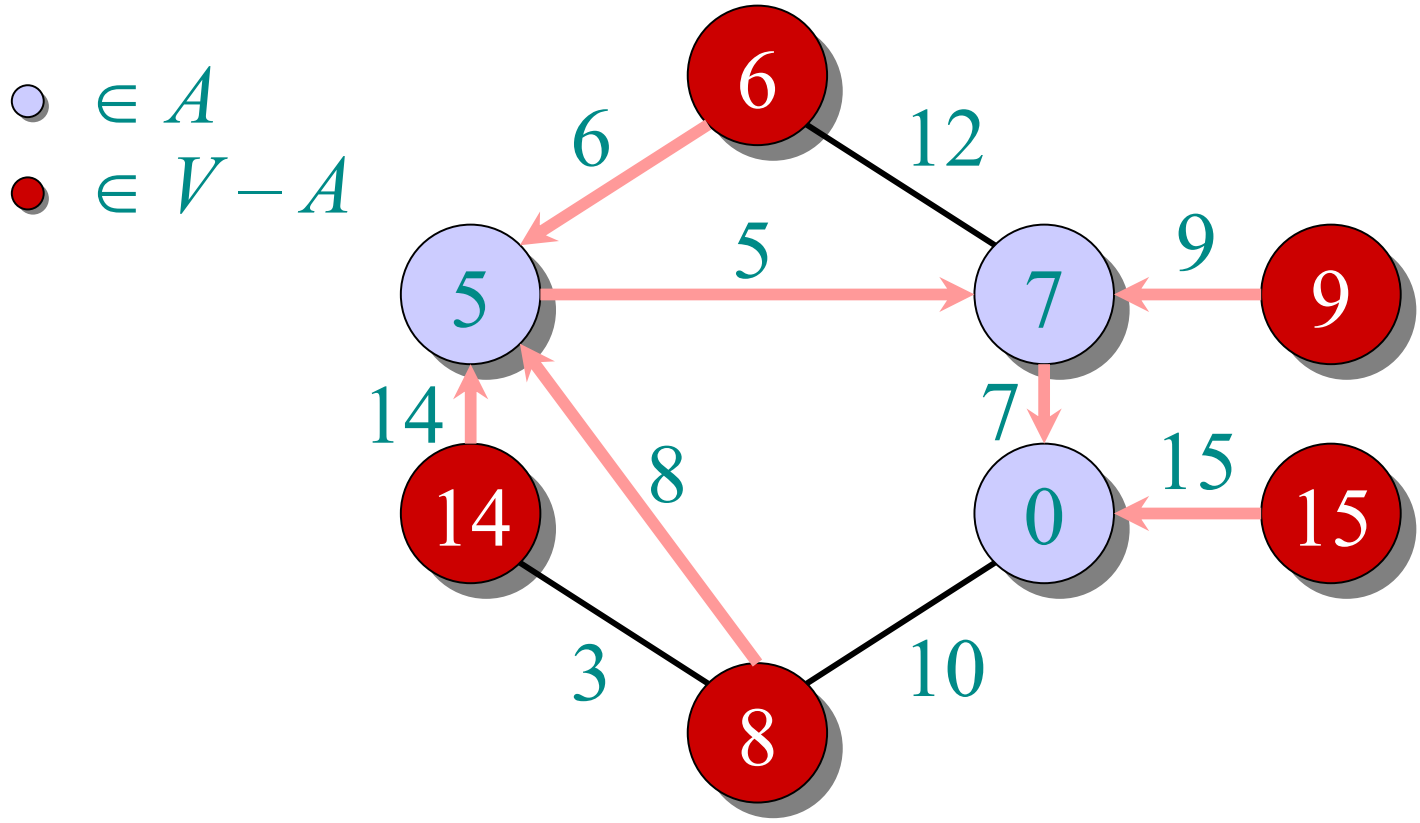
$\in A$

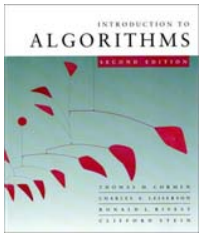
$\in V - A$



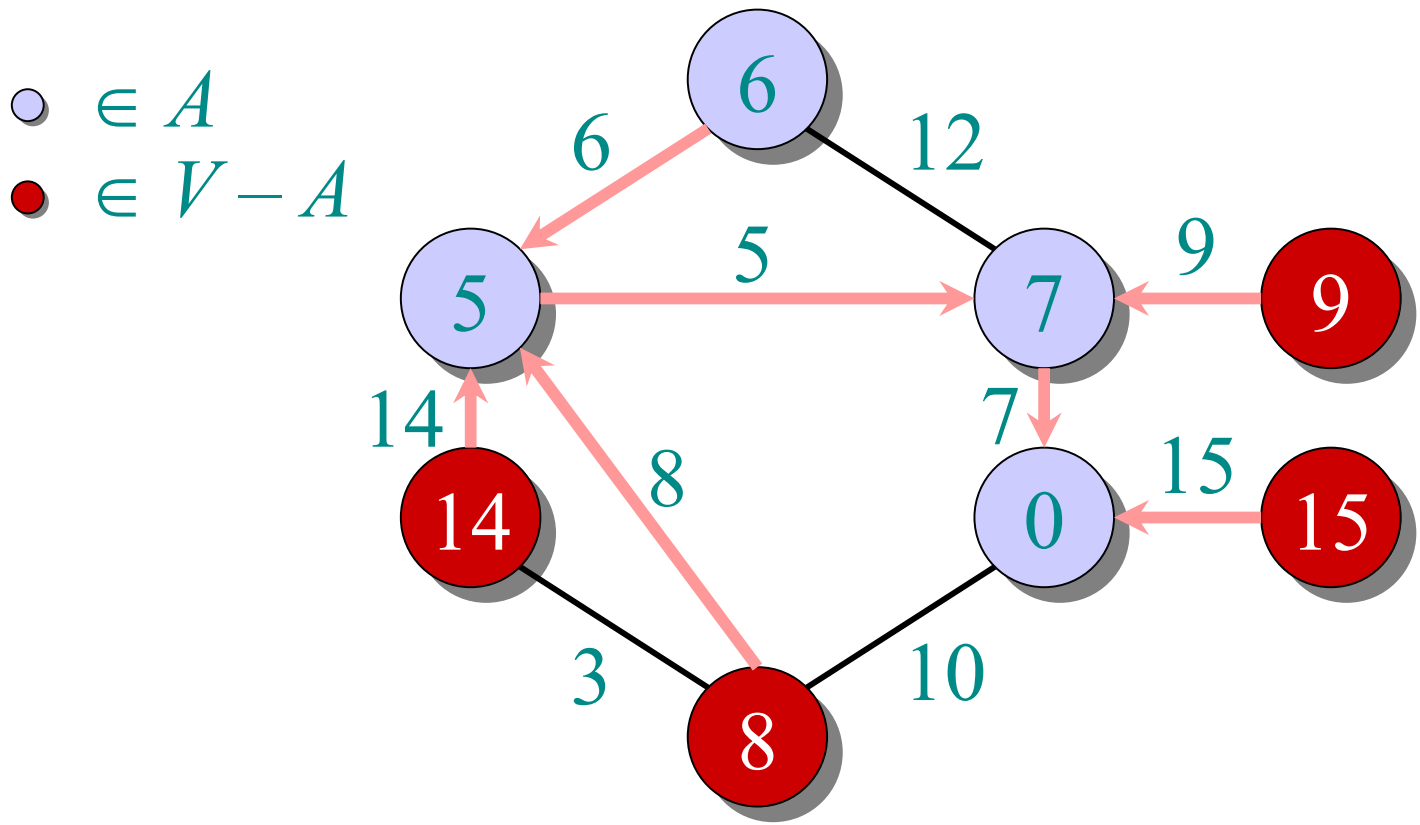


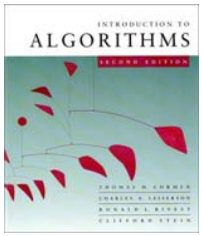
Example of Prim's algorithm



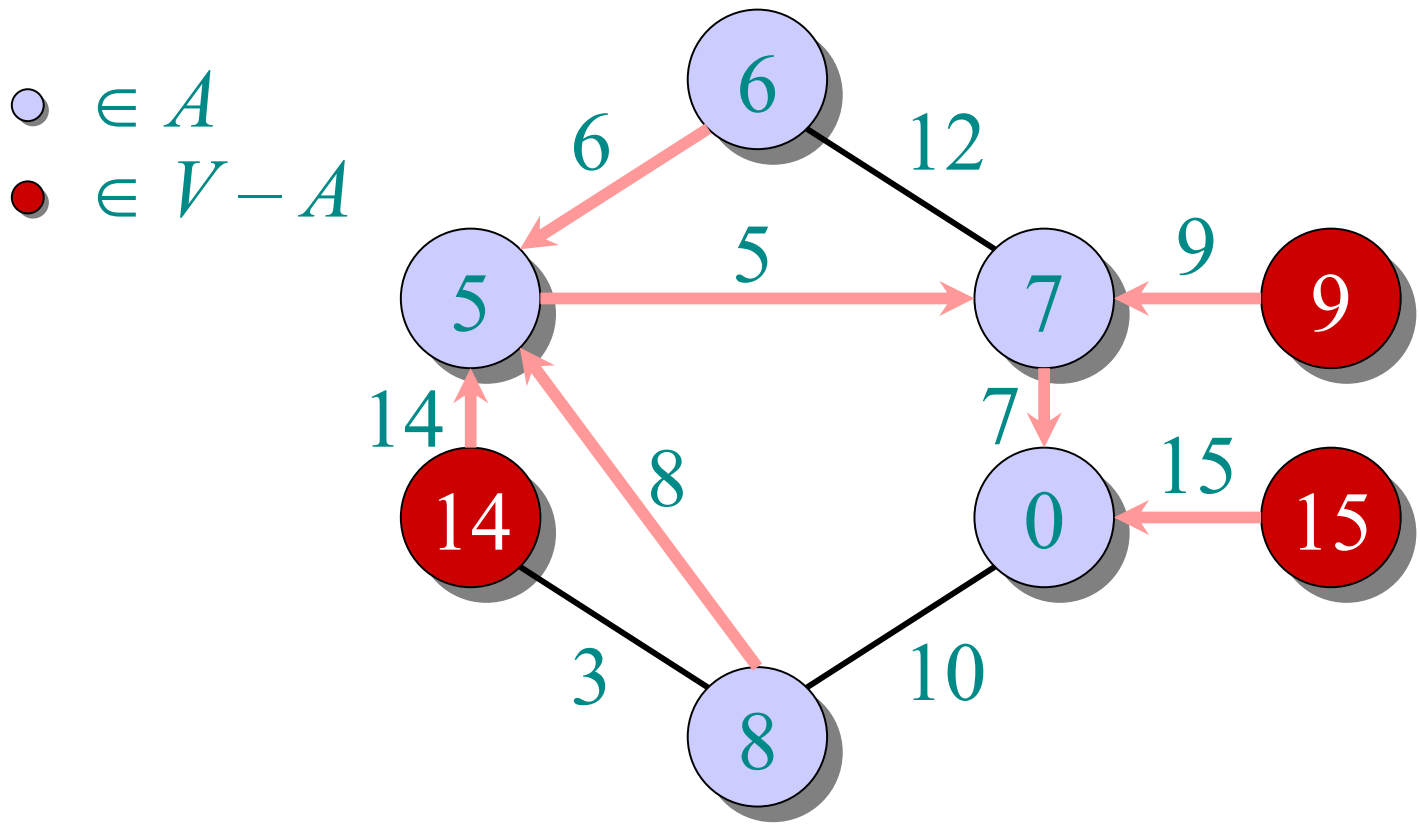


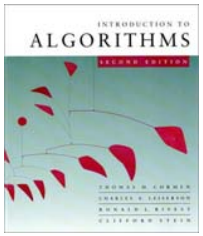
Example of Prim's algorithm





Example of Prim's algorithm

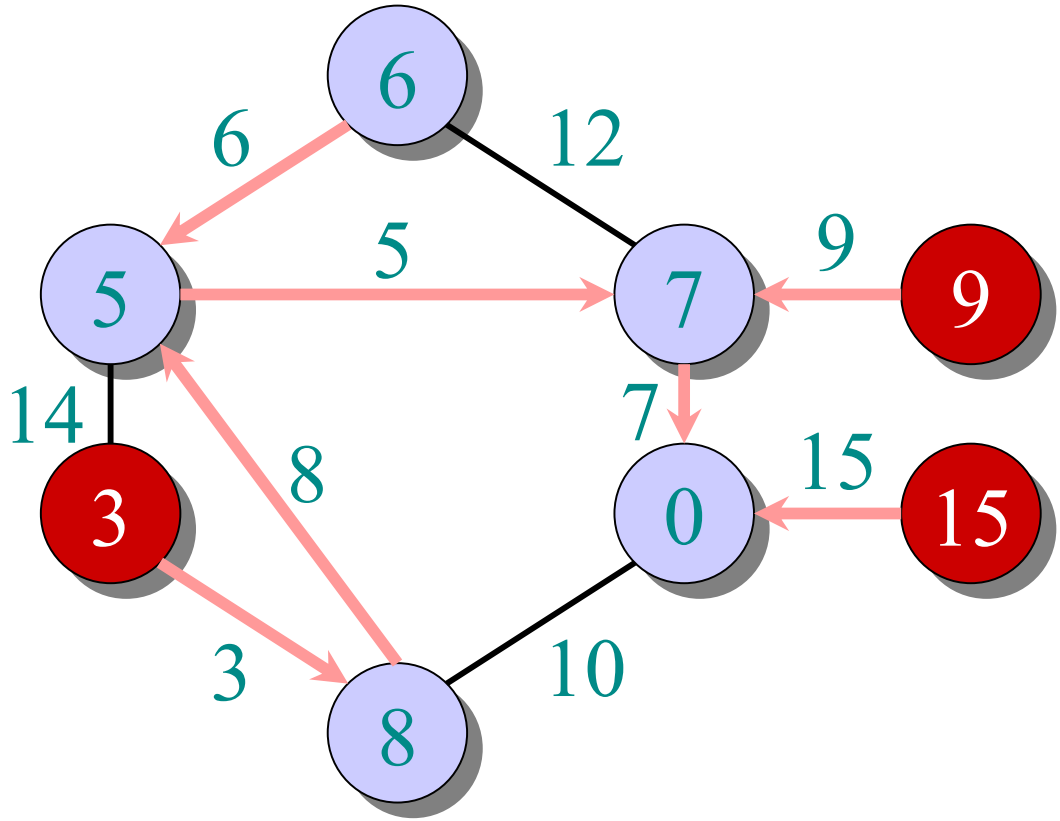


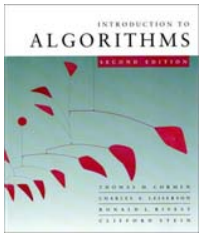


Example of Prim's algorithm

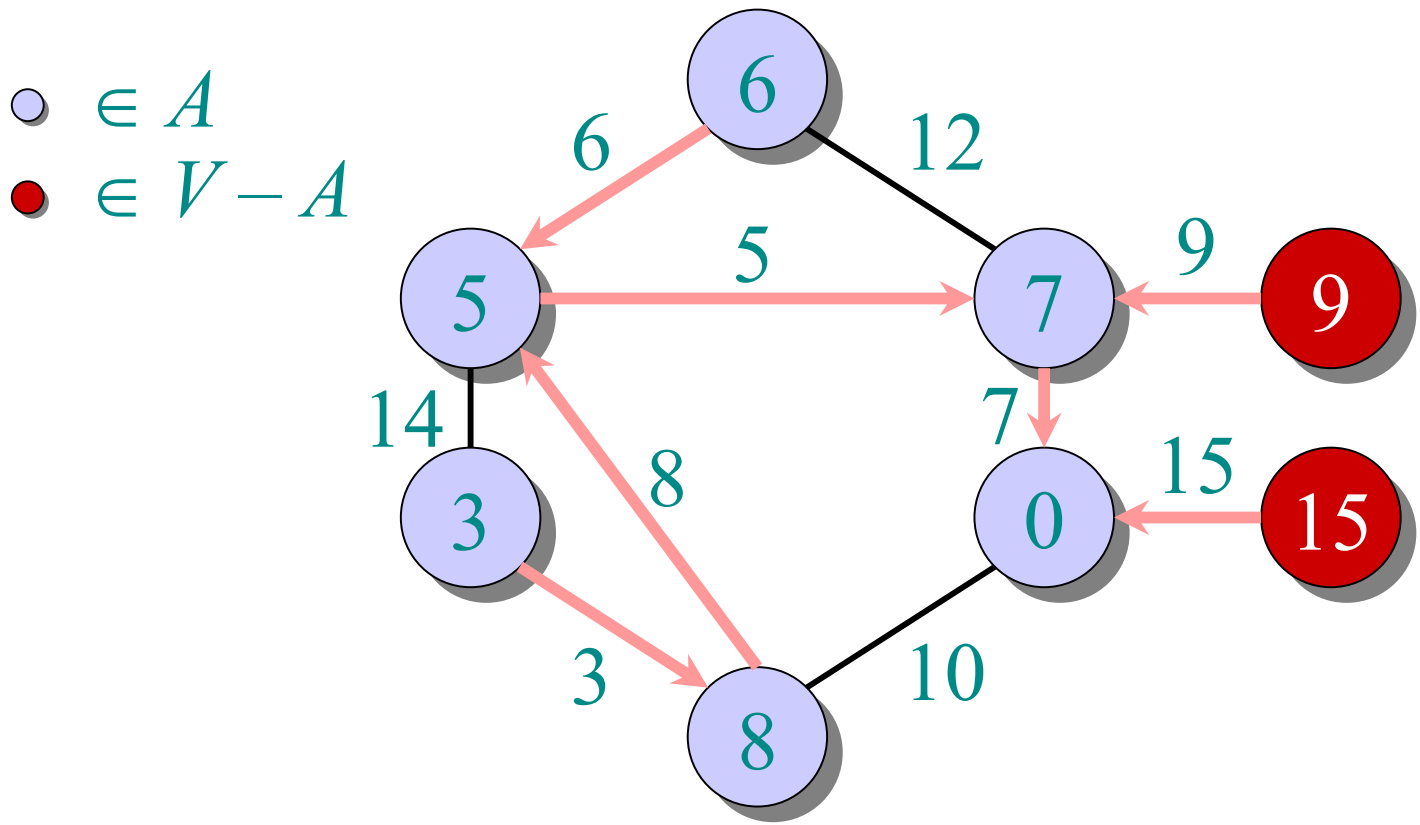
$\in A$

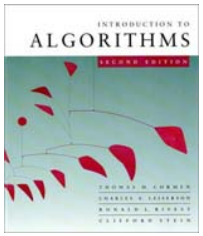
$\in V - A$



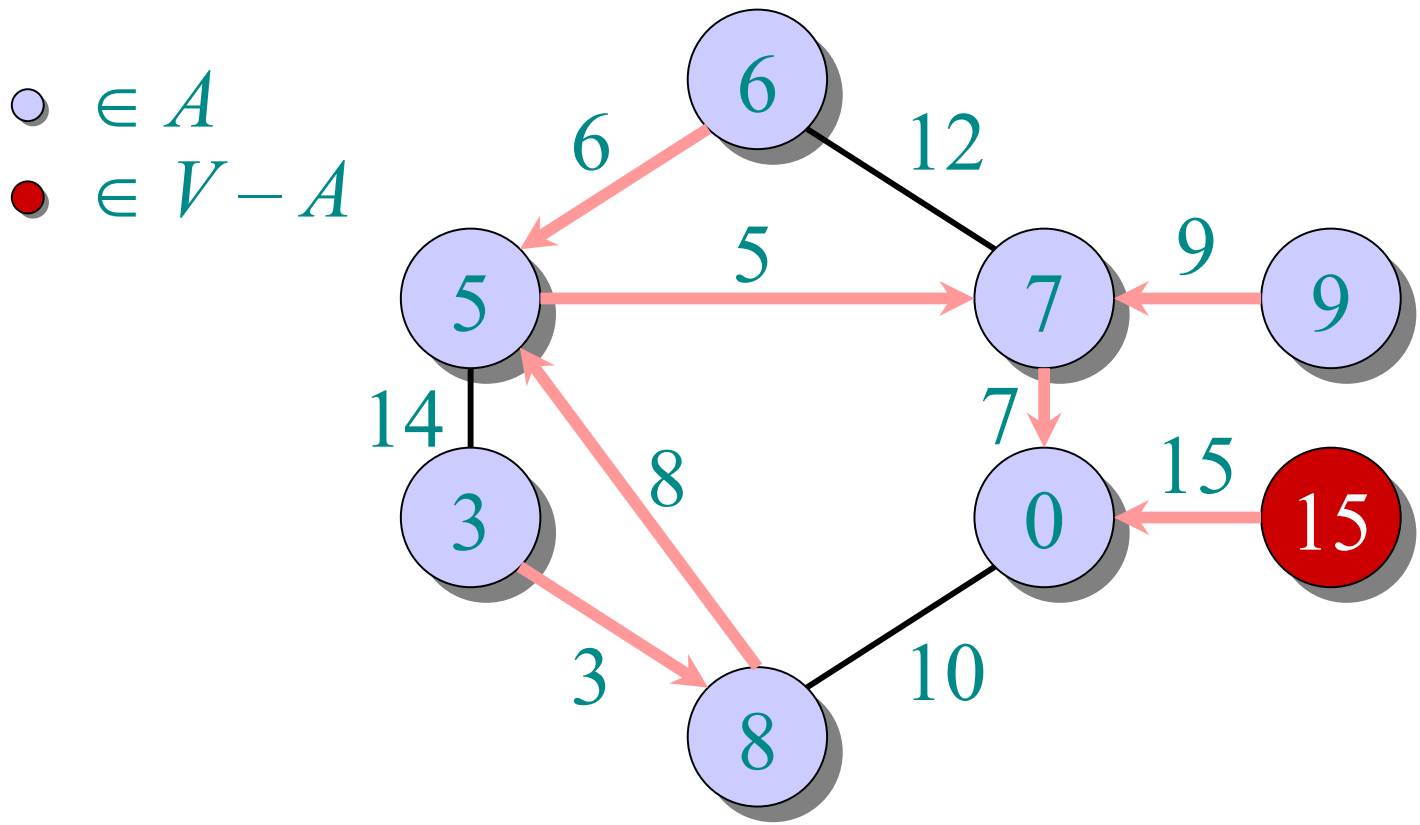


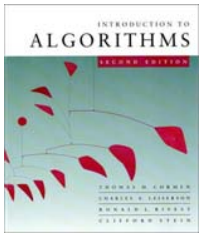
Example of Prim's algorithm



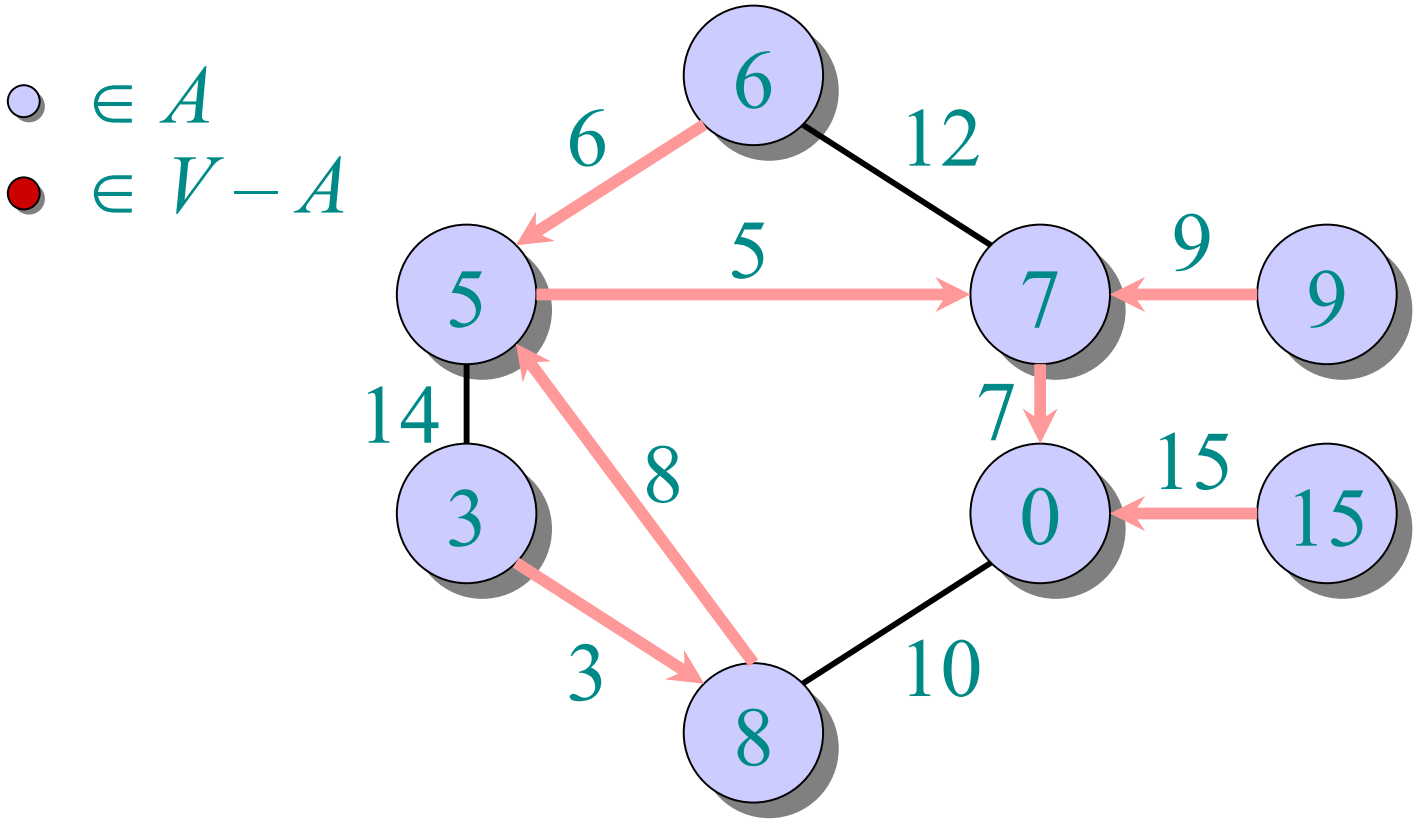


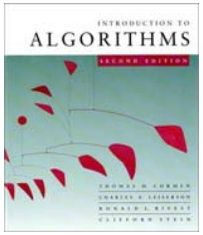
Example of Prim's algorithm





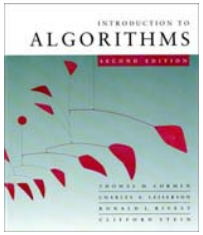
Example of Prim's algorithm





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 $Q \neq \varnothing$
    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 $Q \neq \varnothing$

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

$|V|$ times:

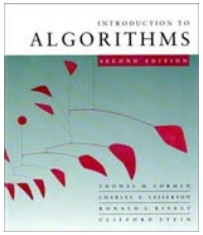
$degree(u)$ times:

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 (continued)

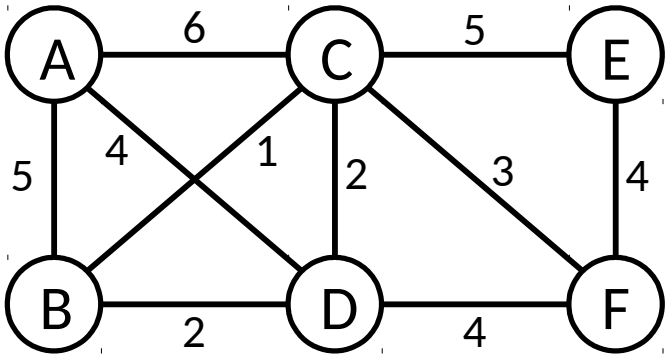
$$\text{Time} = \Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$$

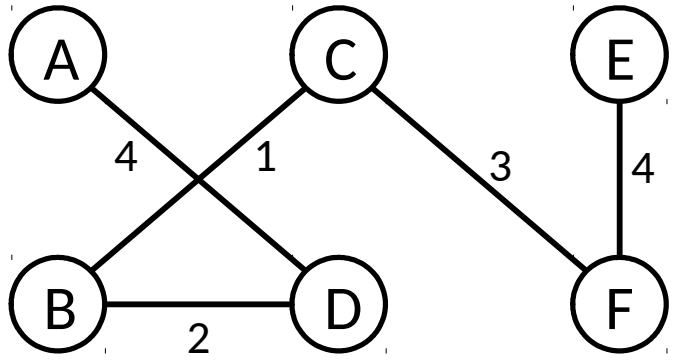
Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$ amortized	$O(1)$ amortized	$O(E + V \lg V)$ worst case

Prim's algorithm

```
def Prim(G, w):          # Pseudo-Python!
    # Input: A connected undirected graph G(V,E),
    #        with edge weights w_e
    # Output: An MST defined by the vector prev
    for all u in V:
        cost[u] =
        prev[u] = nil
    pick any initial node u_0
    cost[u_0] = 0
    # H priority queue using cost as key (set V-S)
    H = makequeue(V)
    while not H.empty():
        v = H.deletemin()
        for all(v,z) in E, z in H:
            if cost[z] > w[v,z]:
                cost[z] = w[v,z]
                prev[z] = v
                H.decreasekey(z)
```

Prim's algorithm





Set S	A	B	C	D	E	F
$\{\}$	**0/nil**	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
A		**5/A**	**6/A**	**4/A**	∞/nil	∞/nil
A, D		**2/D**	**2/D**		∞/nil	**4/D**
A, D, B			**1/B**		∞/nil	**4/D**
A, D, B, C					**5/C**	**3/C**
A, D, B, C, F					**4/F**	

Complexity: the same as Dijkstra's algorithm, $O((|V| + |E|) \log |V|)$

A* search

A search algorithm that finds the shortest path between some nodes u and v in a graph G(V, E).

Compare A* with Dijkstra algorithm
(click to access)

Source: youtube.com, accessed on Oct 2019

Heuristic Functions

- Suppose we want to get to node T, and we are currently at node v. Informally, a *heuristic function* $h(v)$ is a function that 'estimates' how v is away from T.
- Example: Suppose I am driving from Durham to Raleigh. A heuristic function would tell me approximately how much longer I have to drive.

Source: Duke Uni. - Siyang Chen

Description of A*

We are now ready to define the A* algorithm. Suppose we are given the following inputs:

- A graph $G = (V, E)$, with nonnegative edge distances $e(u, v)$
- A start node S and an end node T
- An admissible heuristic h ("never overestimates")

Let $d(v)$ store the best path distance from S to v that we have seen so far. Then we can think of $d(v) + h(v)$ as the estimate of the distance from S to v, then from v to T. Let Q be a queue of nodes, sorted by $d(v) + h(v)$.

Source: Duke Uni. - Siyang Chen

Pseudocode for A*

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v) + h(v)$

while Q not empty **do**

  $v \leftarrow Q.pop()$

  for all neighbours u of v **do**

    if $d(v) + e(v, u) \leq d(u)$ **then**

      $d(u) \leftarrow d(v) + e(v, u)$

    end if

  end for

end while

Source: Duke Uni. - Siyang Chen

Comparison to Dijkstra's Algorithm

Observation: A* is very similar to Dijkstra's algorithm:

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$
while Q not empty **do**
  $v \leftarrow Q.pop()$
  for all neighbours u of v **do**
    if $d(v) + e(v, u) \leq d(u)$ **then**
      $d(u) \leftarrow d(v) + e(v, u)$
    end if
  end for
end while

In fact, Dijkstra's algorithm is a special case of A*, when we set $h(v) = 0$ for all v.

Source: Duke Uni. - Siyang Chen

Performance

How good is A*?

- If we use an admissible heuristic, then A* returns the optimal path distance. Furthermore, any other algorithm using the same heuristic will expand at least as many nodes as A*.
- In practice, if we have a consistent heuristic, then A* can be *much* faster than Dijkstra's algorithm.
- Example: Consider cities (points on the plane), with roads (edges) connecting them. Then the straight-line distance is a consistent heuristic.

Source: Duke Uni. - Siyang Chen

More algorithms for shortest paths

- ML-based shortest paths (click to access)
- Shortest paths via a genetic algorithm (click to access)

Sources: medium.com, researchgate.net, accessed on Oct 2019

A recent application of graph traversal algorithms

- Hypergraph-based connectivity measures for signaling pathway topologies (click to access)

Franzese et al., PLoS Comp Bio, 2019

Algorithms, programming and data structures in Bioinformatics

Subject	4 Exhaustive Search	5 Greedy Algorithms	6 Dynamic Programming Algorithms	7 Divide-and-Conquer Algorithms	8 Graph Algorithms	9 Combinatorial Pattern Matching	10 Clustering and Trees	11 Hidden Markov Models	12 Randomized Algorithms
Mapping DNA	o								
Sequencing DNA					o				
Comparing Sequences			o	o		o			
Predicting Genes			o						
Finding Signals	o	o						o	o
Identifying Proteins					o				
Repeat Analysis						o			
DNA Arrays					o				
Genome Rearrangements		o							
Molecular Evolution							o		

- Read chapters 2 & 4 for the next class

Source: An Introduction to Bioinformatics Algorithms, N.C. Jones and P.A. Pevzner, The MIT Press Cambridge, 2004

(click here for the online version)