CENG 218
Design and Analysis of Algorithms

Izmir Institute of Technology

Lecture 12: Greedy algorithms

Slides were mostly prepared using the material provided by Prof. Charles E. Leiserson and Prof. Erik Demaine from MIT

Greedy algorithms

- A ***greedy algorithm*** always makes the choice that looks best at the moment.
- It makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
- Sometimes, it doesn’t lead to optimal solution. However, when it does, it is more efficient.
- Our first example is an activity-selection problem.

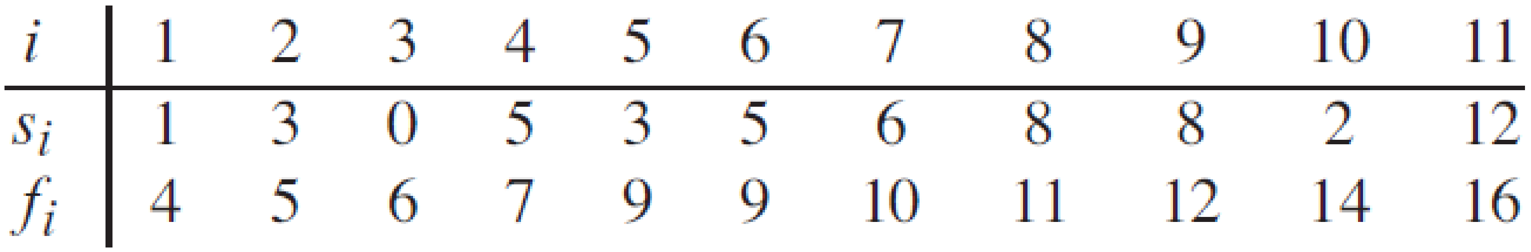
Activity-selection problem

- Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed ***activities*** that wish to use a resource, such as a lecture hall, which can serve only one activity at a time.
- Each activity a_i has a ***start time*** s_i and a ***finish time*** f_i .
- If selected, activity a_i takes place during the time interval $[s_i , f_i)$.
- We say, activities a_i and a_j are ***compatible*** if the intervals $[s_i , f_i)$ and $[s_j , f_j)$ do not overlap.

Activity-selection problem

In the ***activity-selection problem***, we wish to select a maximum-size subset of compatible activities.

For example, consider the following set of activities:



i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

$\{a_3, a_9, a_{11}\}$ is an example of compatible activities.
$\{a_1, a_4, a_8, a_{11}\}$ is larger, also 'a maximum' subset.

Optimal substructure

- We can easily verify that the activity-selection problem exhibits optimal substructure.

  optimal substructure: optimal solutions to a problem incorporate optimal solutions to subproblems
- Let S_{ij} denote the set of activities that start after activity a_i finishes and that finish before a_j starts.
- Suppose further that a maximum set of mutually compatible activities is A_{ij} , which includes some activity a_k .
- Knowing a_k is in an optimal solution, we are left with two subproblems: $A_{ij} = A_{ik} \cup a_k \cup A_{kj}$

Can we solve it by DP?

- Let $c[i,j]$ be the size of an optimal solution for S_{ij}.
- To solve this problem with dynamic programming, we need to examine all activities in S_{ij} to find a_k .

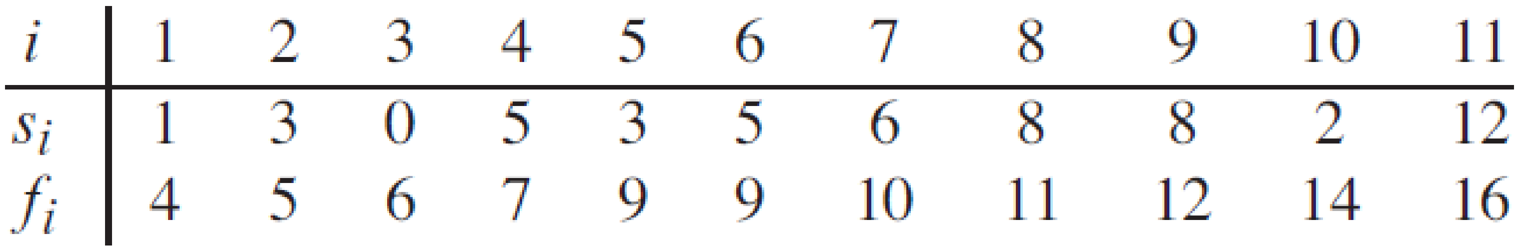
$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \, , \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset \, . \end{cases}$$

- We could then develop a recursive algorithm and memoize it, or work bottom-up and fill in a table.
- But we would miss an important property of the activity-selection problem: *the greedy choice*.

Greedy choice property:

A locally optimal choice is globally optimal.

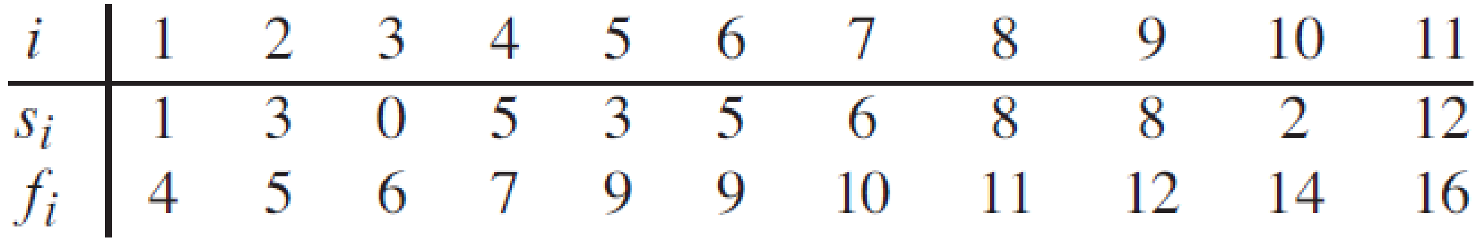
- To solve the activity-selection with greedy choice, we should choose an activity that leaves the maximum resource available: Choose the activity in *S* with the earliest finish time.
- The greedy choice in our example is activity a_1 .



i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Activity-selection with greedy choice

- If we make the greedy choice, we have ***only one*** remaining subproblem (a requirement for a greedy algorithm): Finding activities that start after a_1 finishes.
- In the second step, we see that a_4 is the greedy choice. Then, a_8 follows.



i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Activity-selection with greedy choice

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

- We end up with the set of activities $\{a_1, a_4, a_8, a_{11}\}$ which is a maximum-size subset.
- This greedy choice works for any activity-selection problem of this type.
- But it does not mean that any greedy approach provides us the optimal solution. Check Exercise 16.1-3 it the textbook.

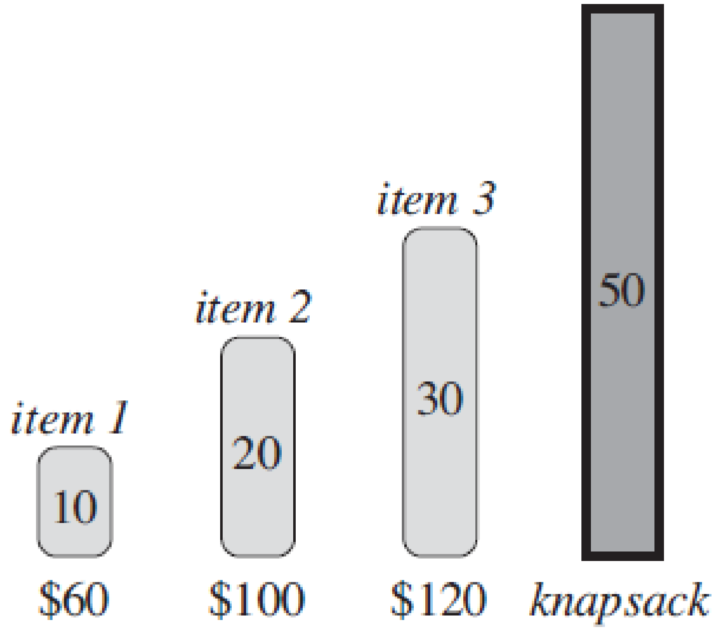
Greedy vs. dynamic programming

0-1 knapsack problem:

A thief robbing a store finds *n* items. The *i*th item worths v_i dollars and weighs w_i pounds. The thief can carry at most *W* pounds in his knapsack. Which items should he take?

A greedy algorithm may say 'taking the items in order of greatest value per pound yields an optimal solution'.

But it is wrong.



Minimum spanning trees

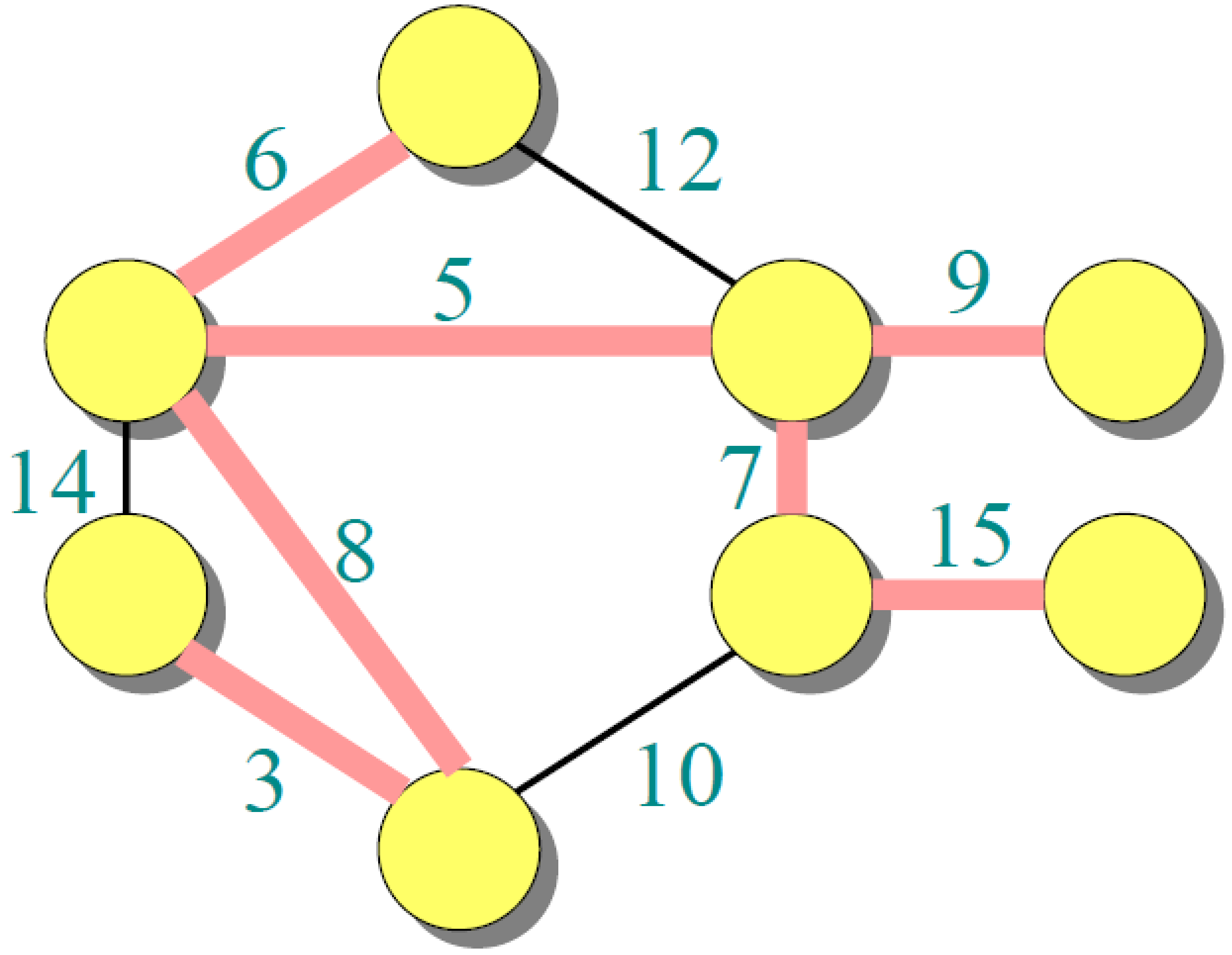
Input: An undirected graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$.
Output: A ***spanning*** (connecting all vertices) tree T of minimum weight:

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

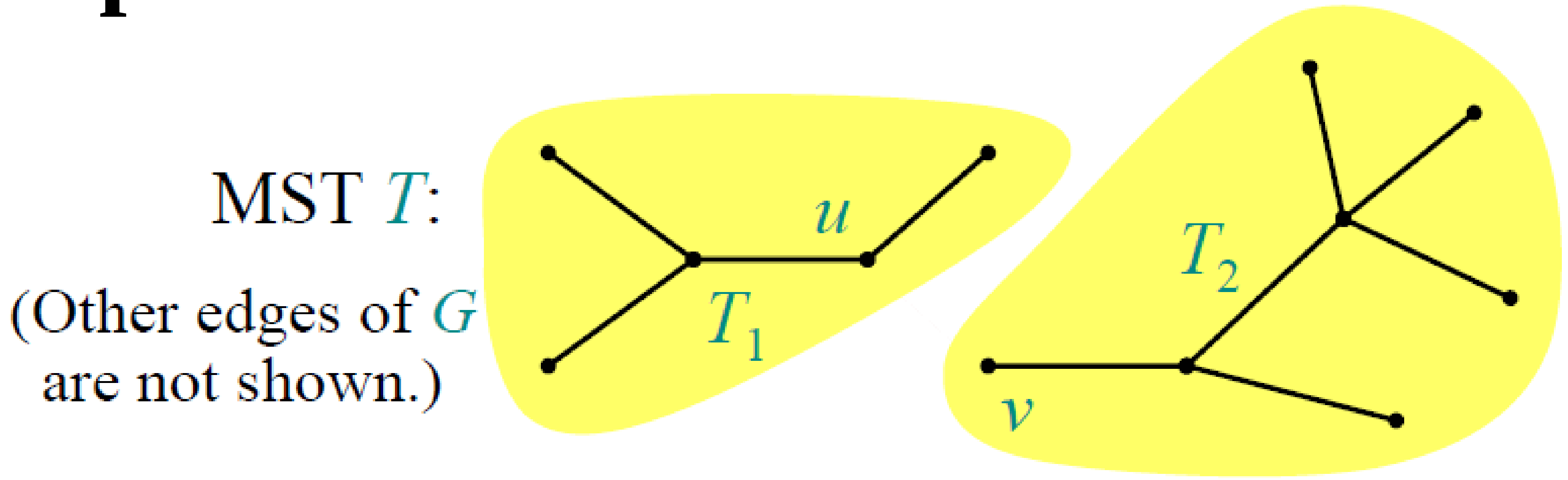
Background:
A graph $G = (V, E)$ is an ordered pair consisting of
- a set V of ***vertices*** (singular: ***vertex***),
- a set $E \subseteq V \times V$ of ***edges***.

In an ***undirected graph*** $G = (V, E)$, the edge set E consists of *unordered* pairs of vertices.

Example of MST



Optimal substructure



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:

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

Similarly for T_2.

Proof of optimal substructure

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

Proof. If T_1' was a lower-weight spanning tree than T_1, then $T' = \{(u, v)\} \cup T_1' \cup T_2$ would be a lower-weight spanning tree than T.

This is not possible since we know that T is a MST.

Proof by contradiction.

Do subproblems overlap in MST problem?

Q. Do we also have ***overlapping subproblems*** (hallmark #2 of DP)?

A. Yes, since taking out edges in different order leads to same subproblems.

Q. Then, dynamic programming may work?

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

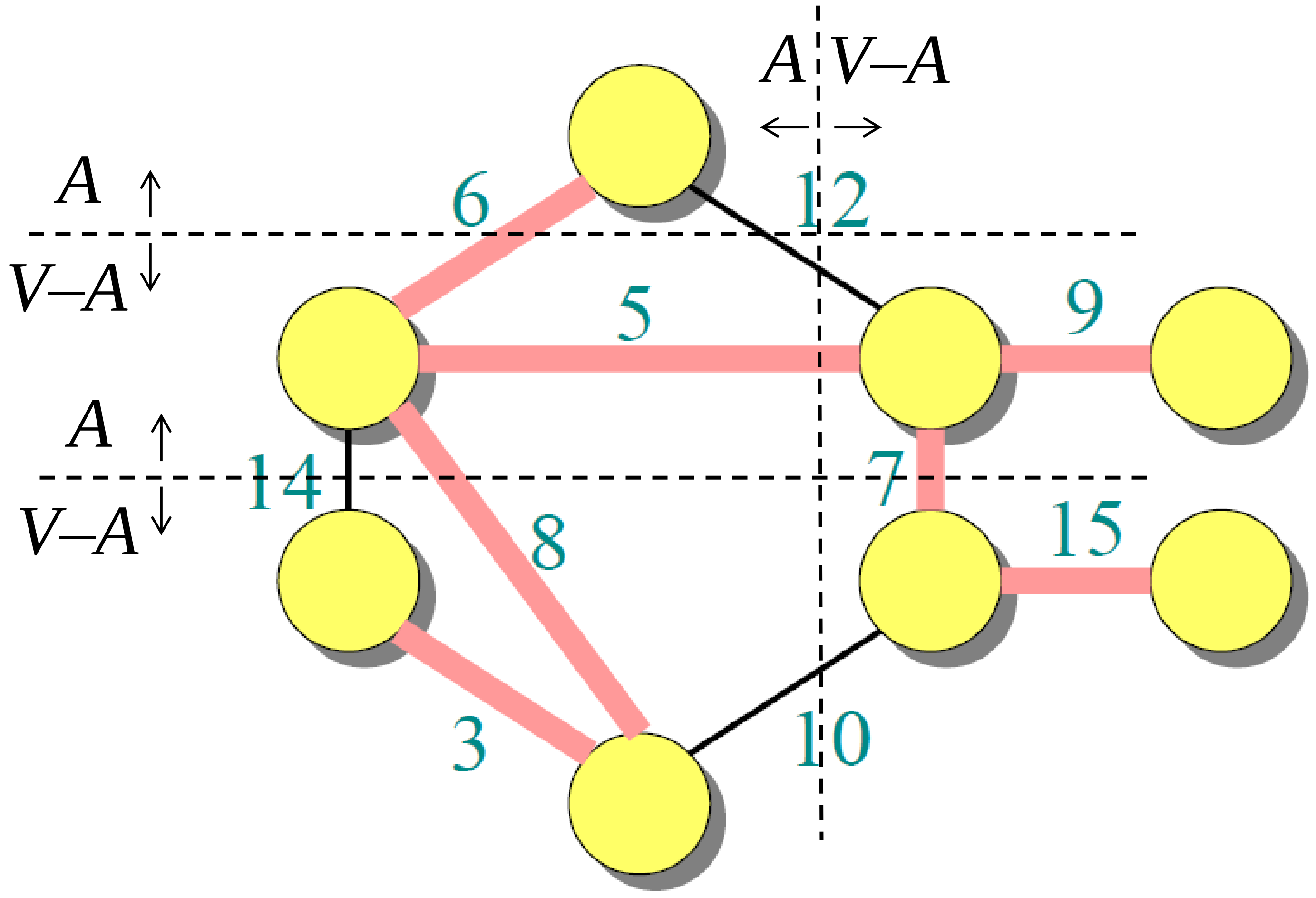
Greedy-choice holds in MST problem?

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, $(u, v) \in T$.

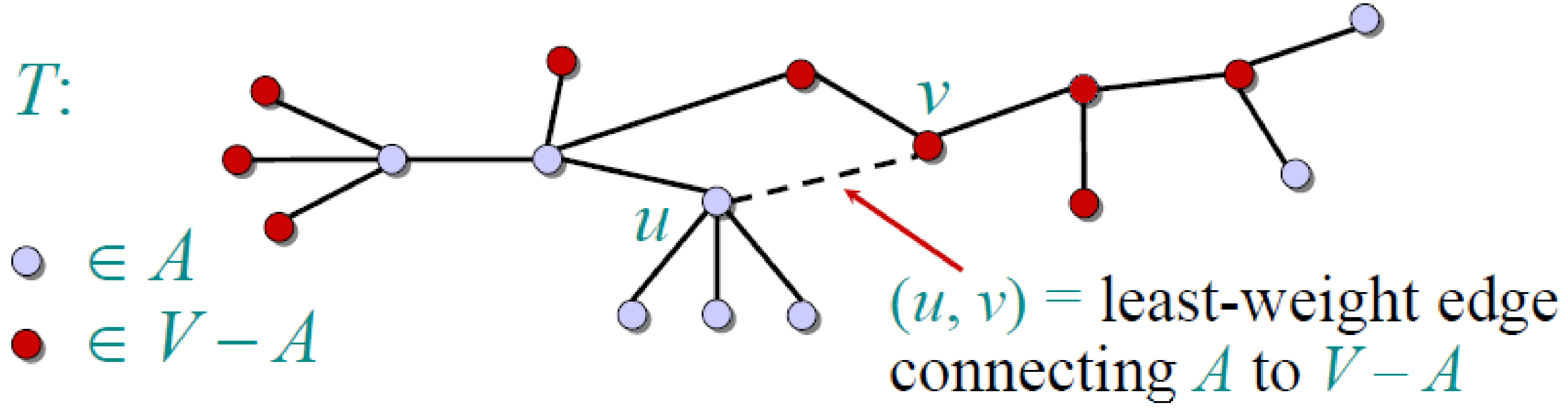
Understanding the theorem

T is the MST. The least-weight edge connecting *A* to *V* – *A* is in *T*.



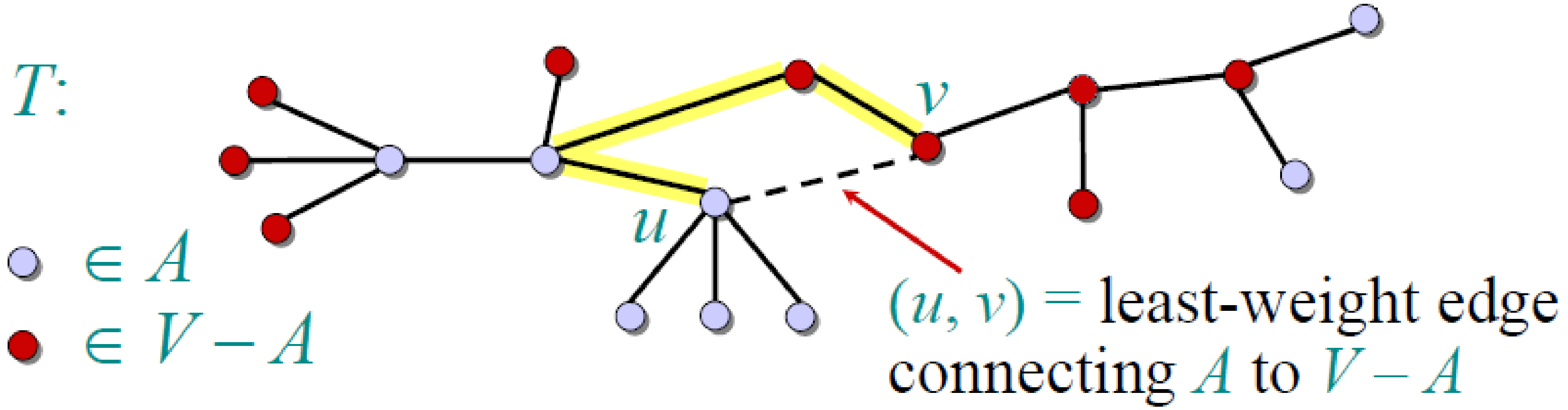
Proof of theorem

Proof. Suppose T is the MST and $(u, v) \notin T$.



Proof of theorem

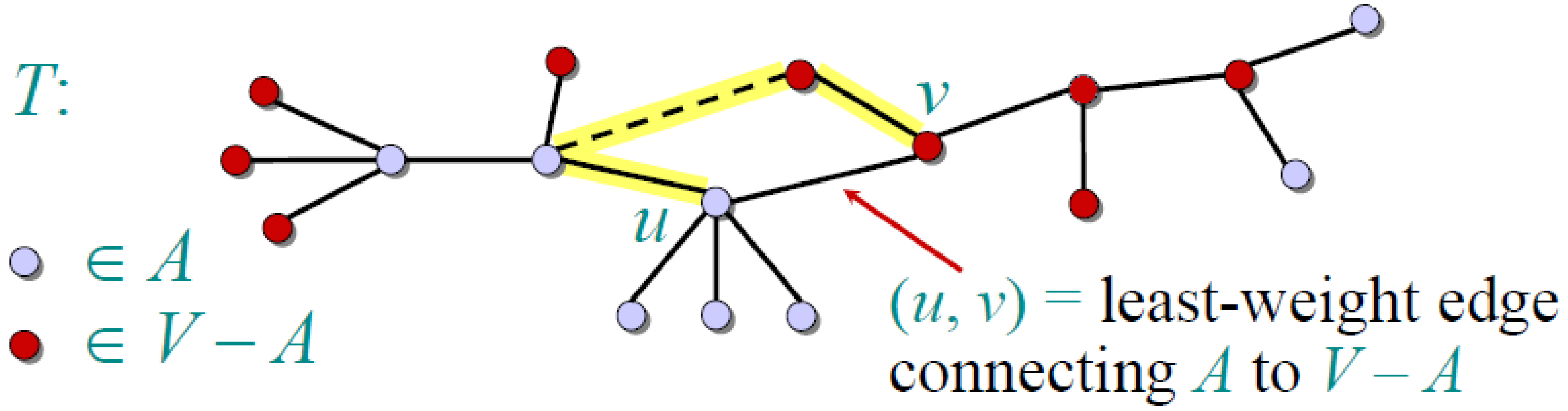
Proof. Suppose $(u, v) \notin T$.



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

Proof of theorem

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

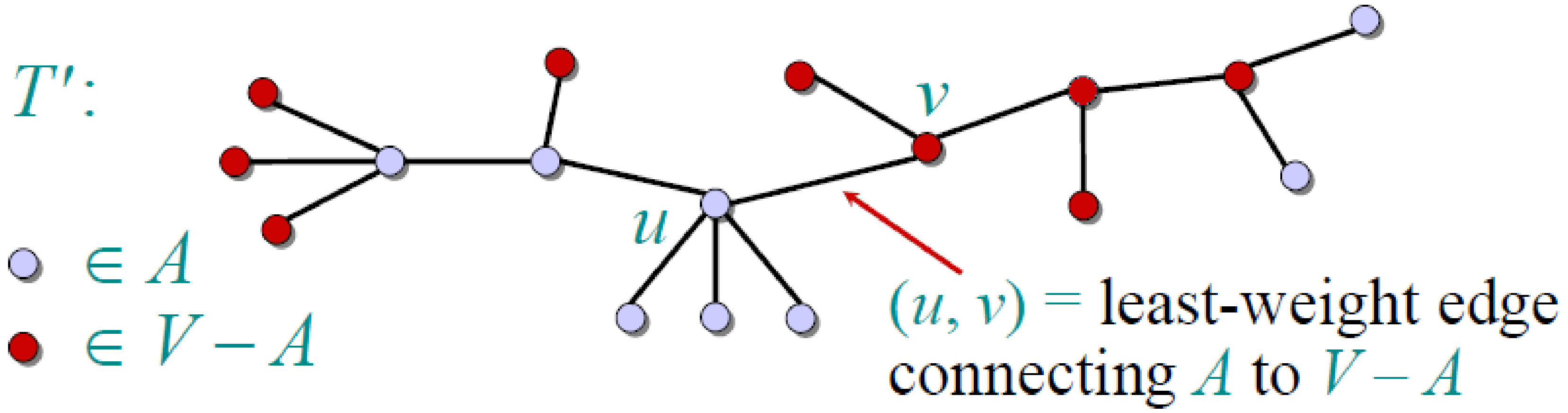


Consider the unique simple path from u to v in T.

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 $(u, v) \notin T$.



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 $V - A$.

T' becomes a lighter-weight spanning tree than T, contradicts with the fact that T is the MST.

How to solve MST problem?

Since we have ***optimal substructure*** and ***greedy choice*** properties, let us discover greedy algorithms to solve MST.

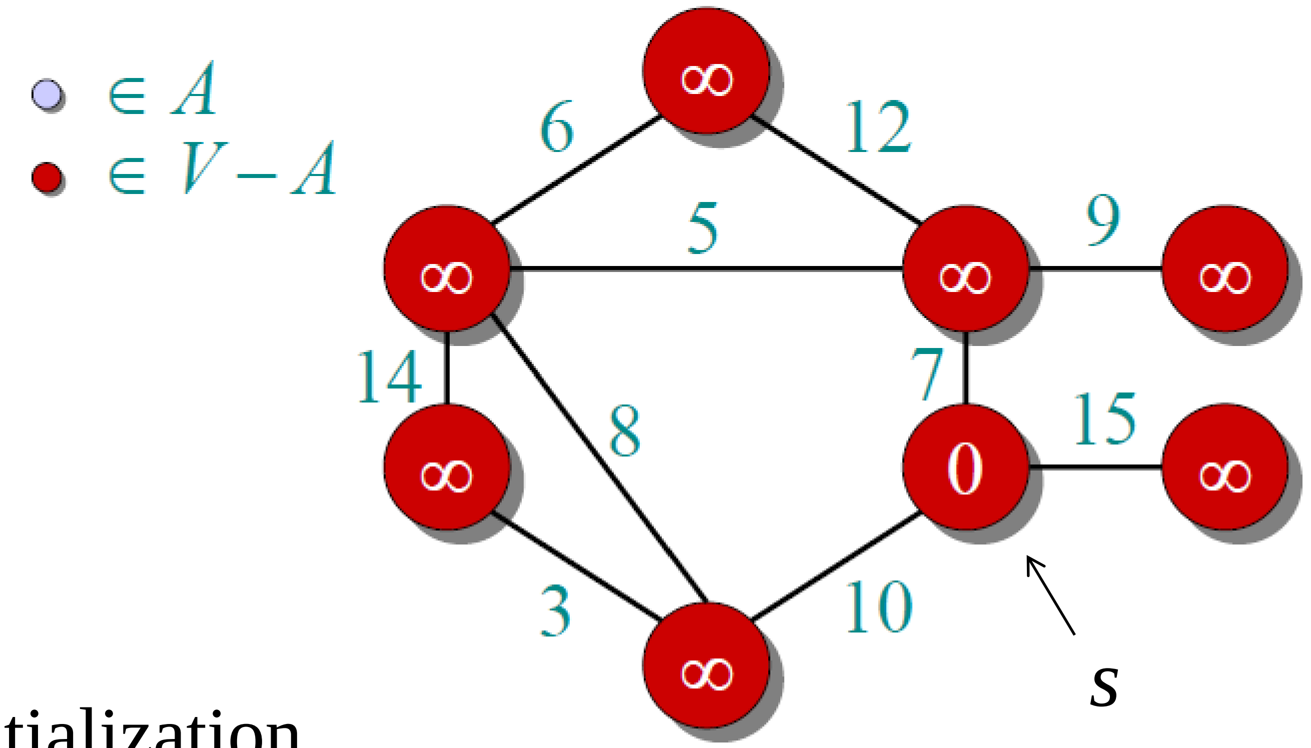
Prim’s algorithm to compute MST

Idea: Maintain $V - A$ as a priority queue Q. Assign a key to 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 $key[v] \leftarrow w(u, v)$        ▷ DECREASE-KEY
                         $\pi[v] \leftarrow u$
```

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

Example of Prim's algorithm



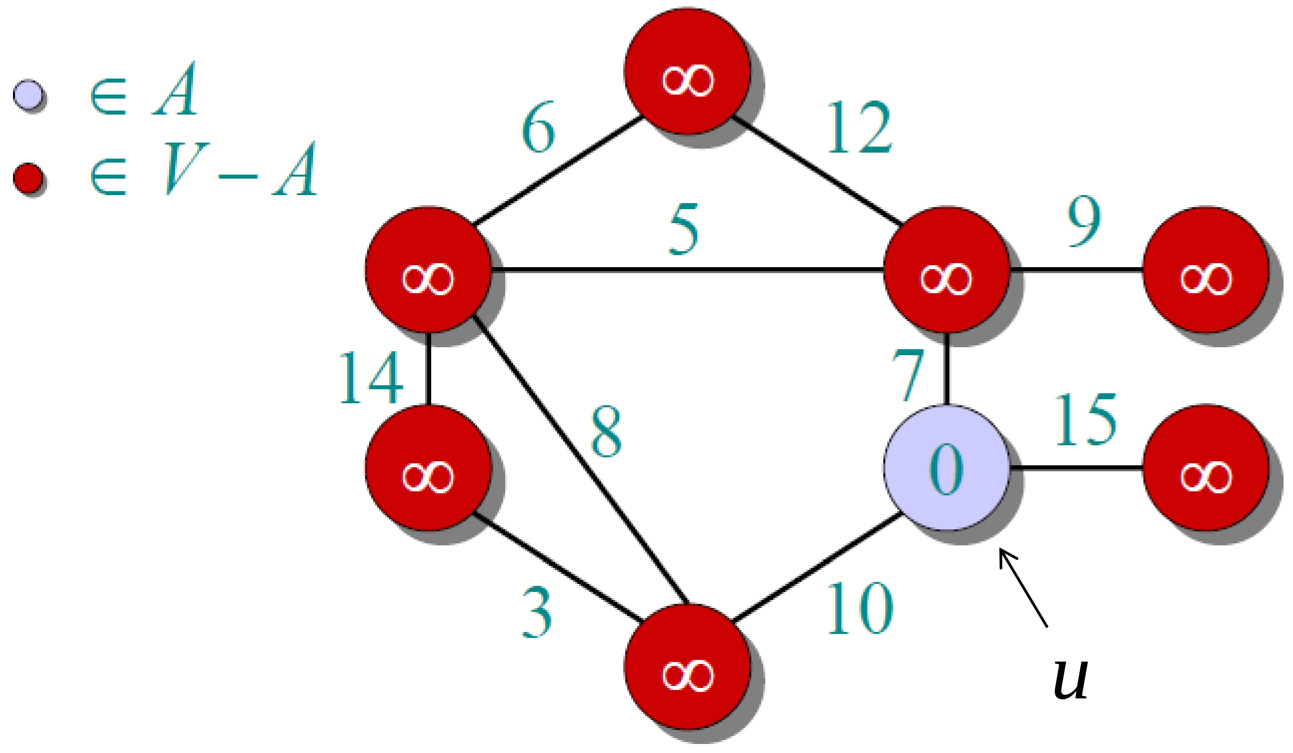
Initialization

$Q \leftarrow V$

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

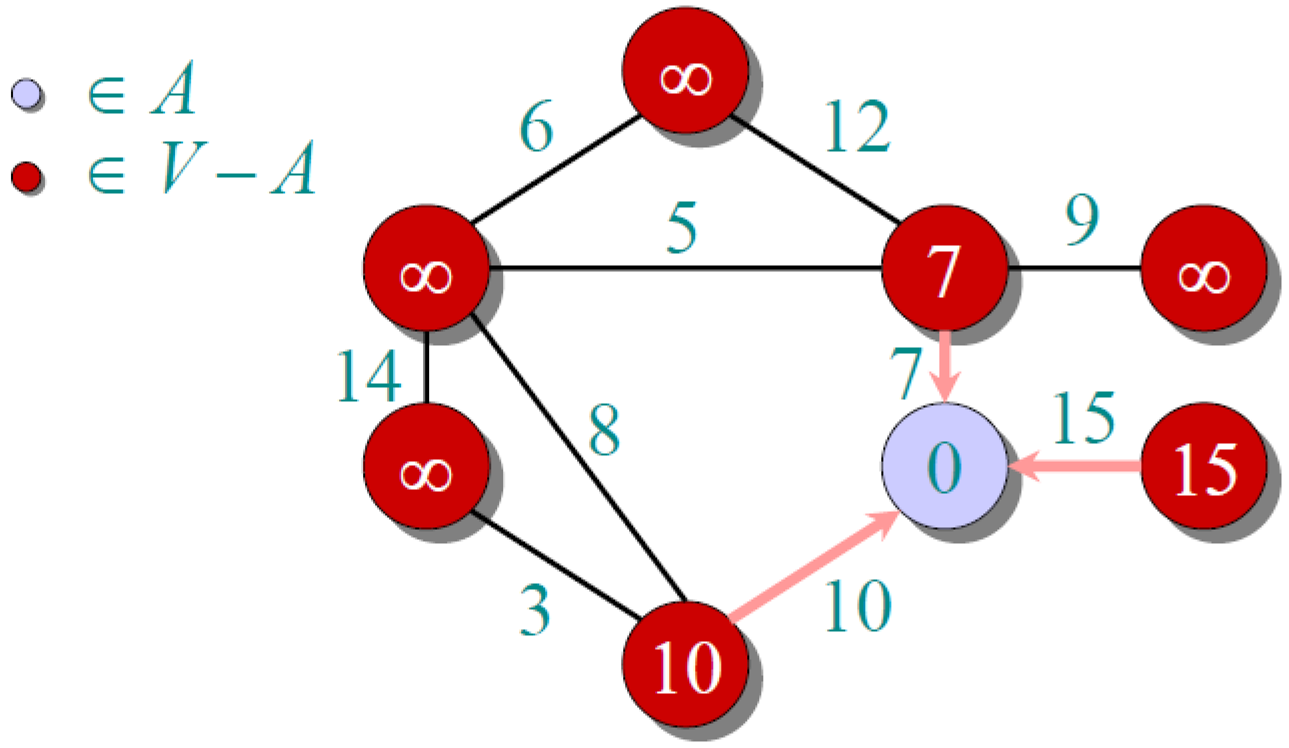
$key[s] \leftarrow 0$ for some arbitrary $s \in V$

Example of Prim's algorithm



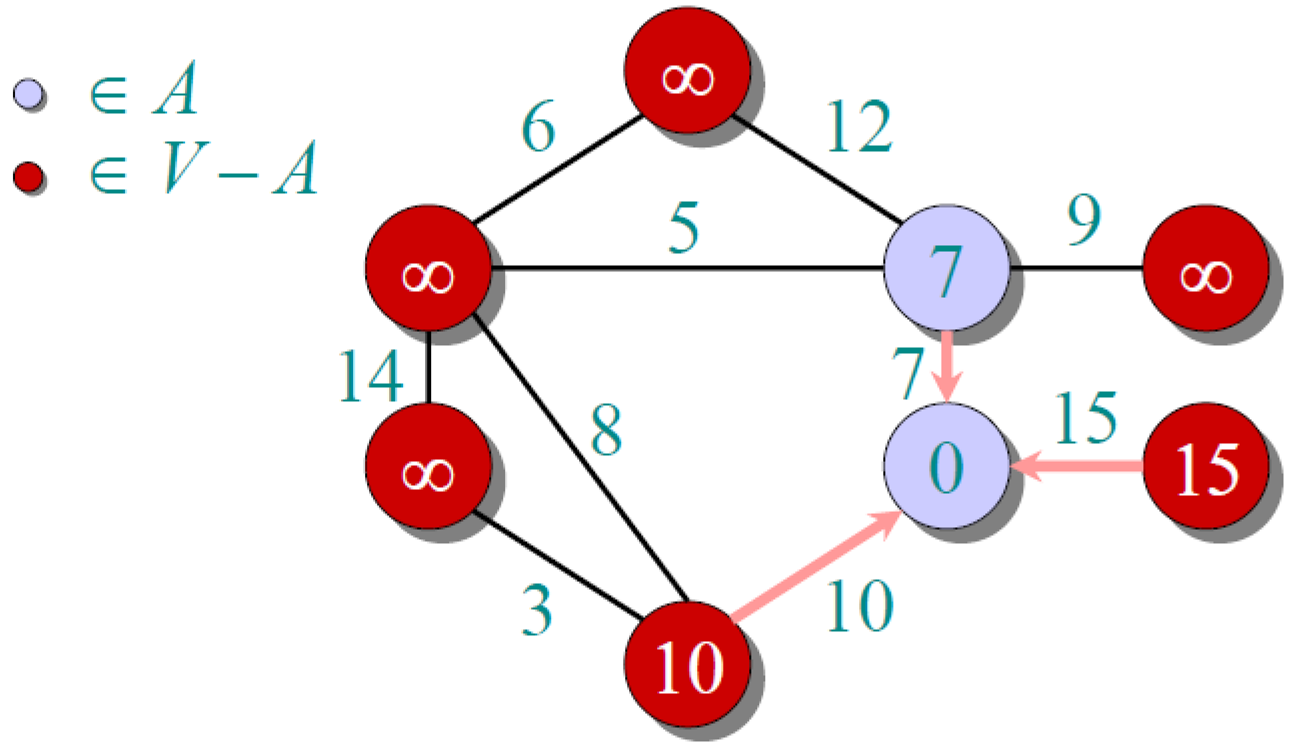
$u \leftarrow$ EXTRACT-MIN(Q)

Example of Prim's algorithm



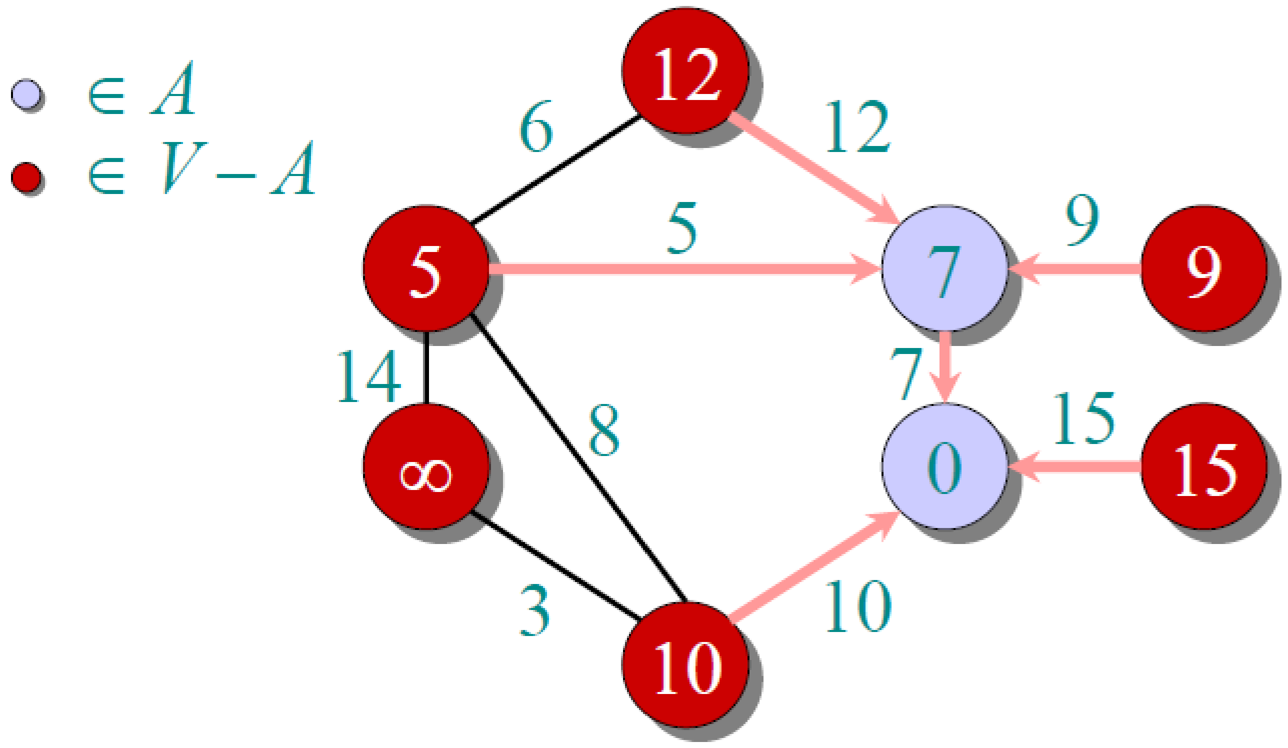
```
for each $v \in Adj[u]$
    do if $v \in Q$ and $w(u, v) < key[v]$
        then $key[v] \leftarrow w(u, v)$      ▷ DECREASE-KEY
             $\pi[v] \leftarrow u$
```

Example of Prim's algorithm



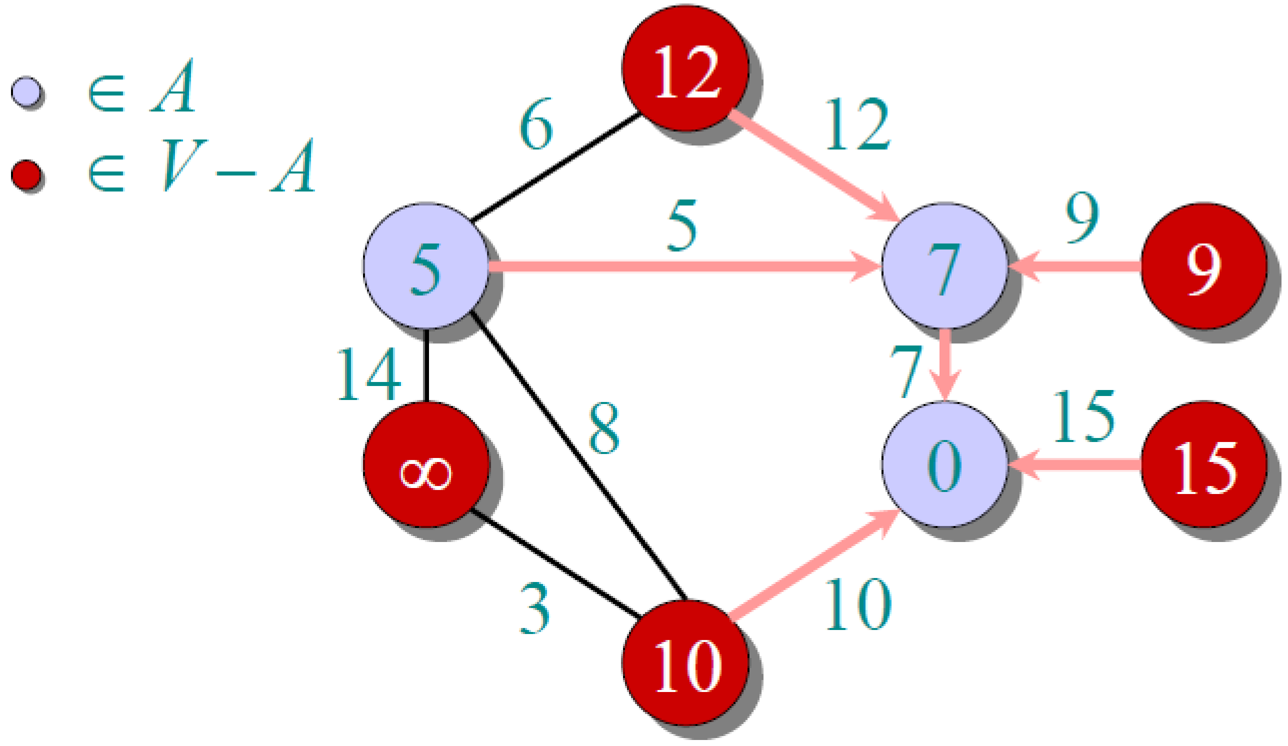
$u \leftarrow$ EXTRACT-MIN(Q)

Example of Prim's algorithm



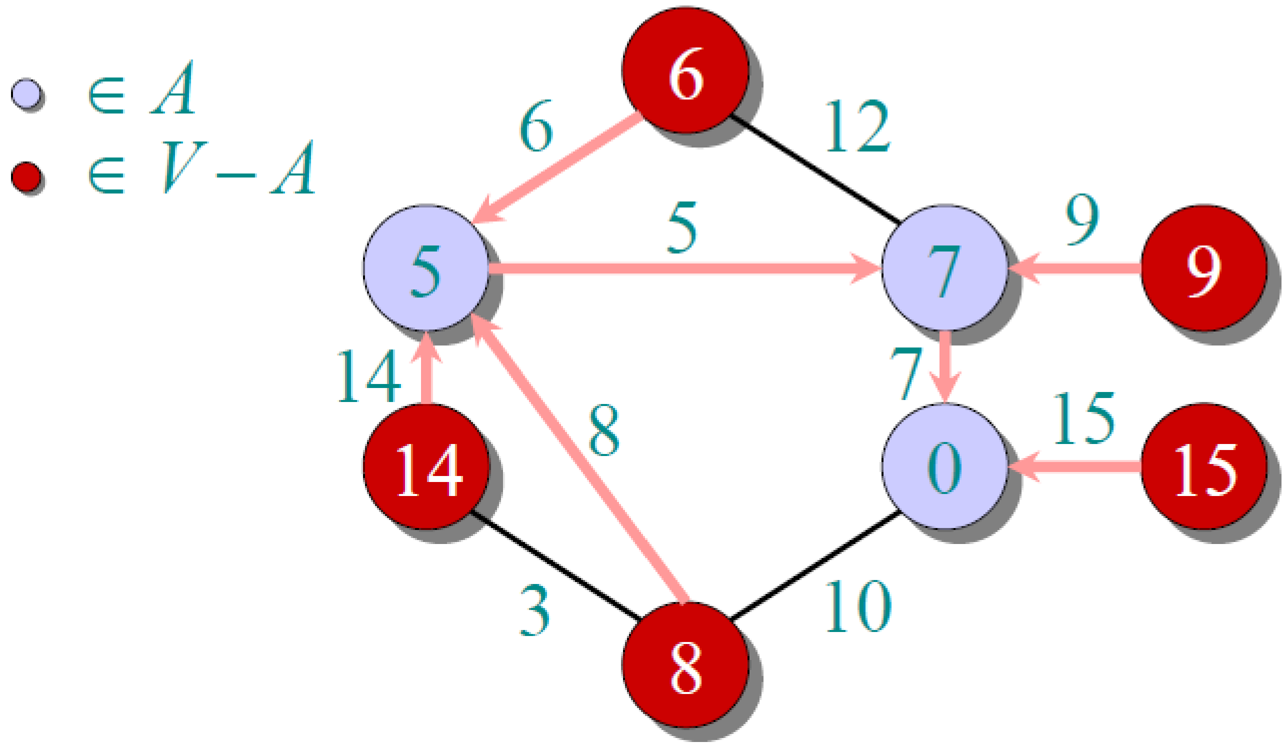
```
for each $v \in Adj[u]$
    do if $v \in Q$ and $w(u, v) < key[v]$
        then $key[v] \leftarrow w(u, v)$    ▷ DECREASE-KEY
             $\pi[v] \leftarrow u$
```

Example of Prim's algorithm



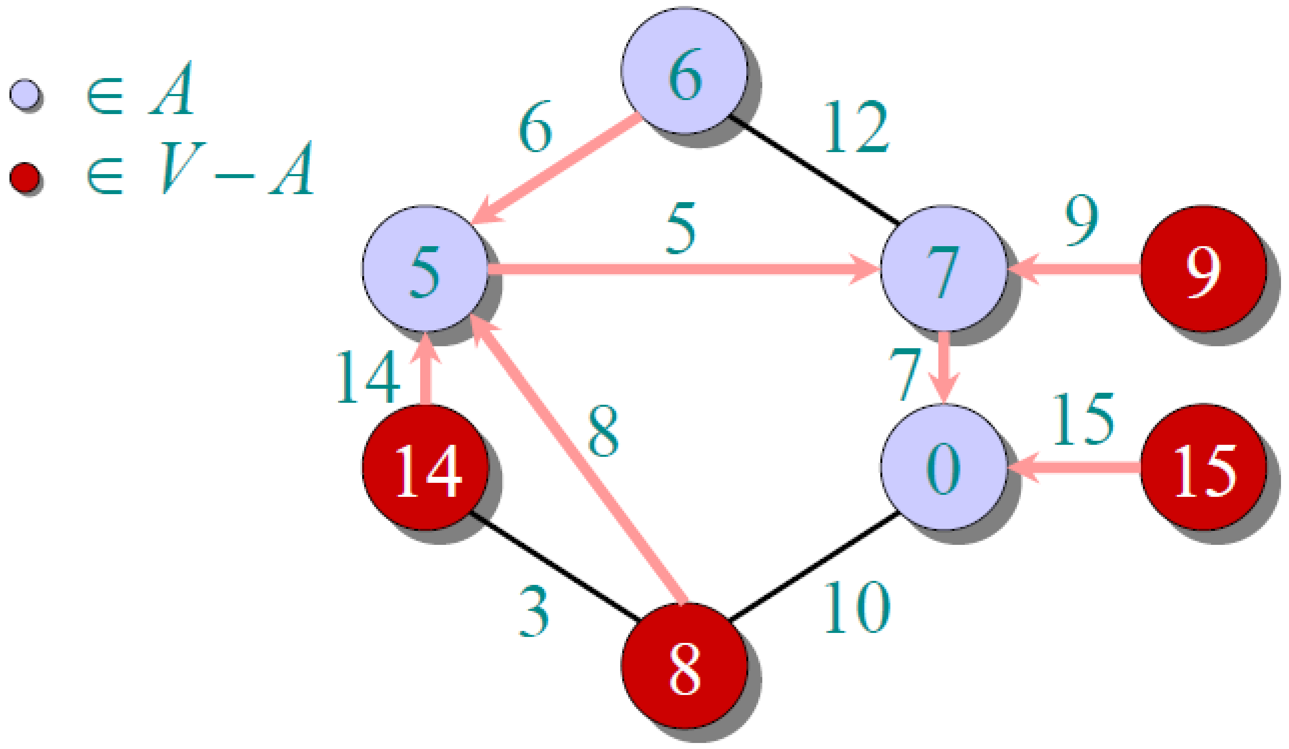
$u \leftarrow$ EXTRACT-MIN(Q)

Example of Prim's algorithm



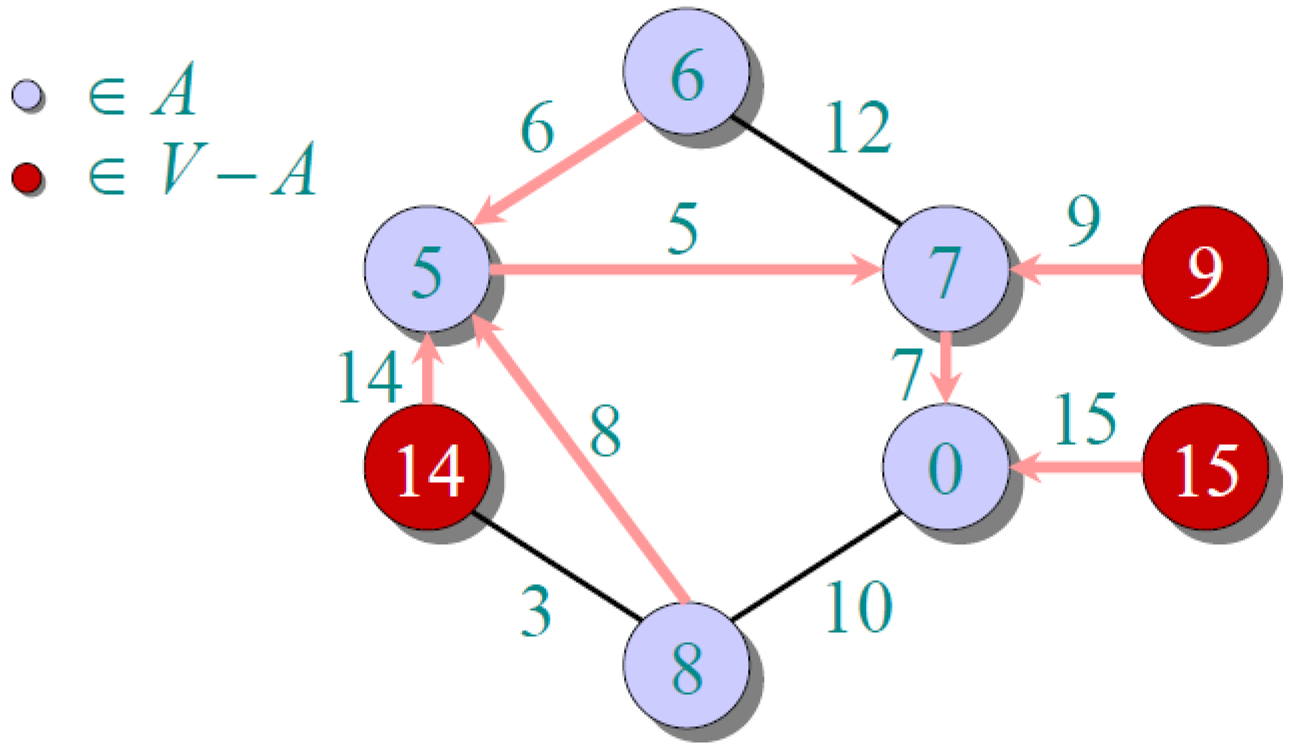
```
for each $v \in Adj[u]$
    do if $v \in Q$ and $w(u, v) < key[v]$
        then $key[v] \leftarrow w(u, v)$     ▷ DECREASE-KEY
             $\pi[v] \leftarrow u$
```

Example of Prim's algorithm



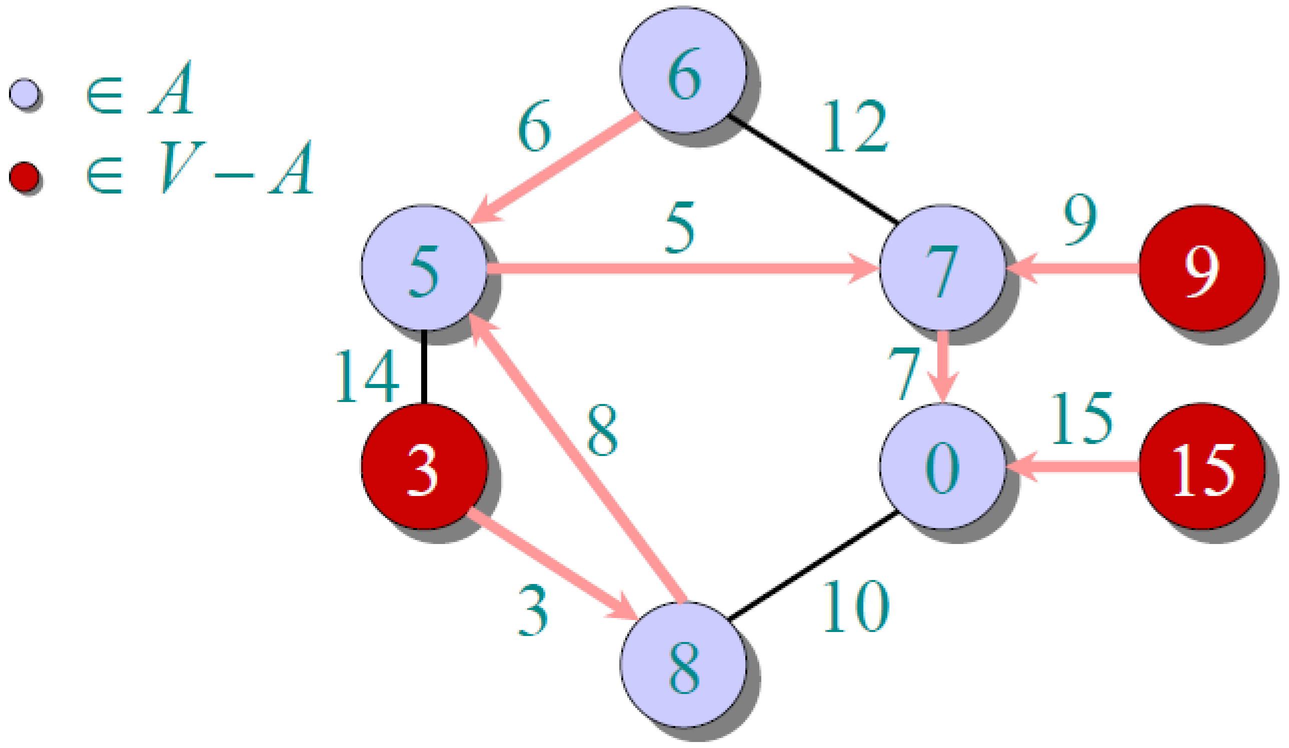
$u \leftarrow \text{EXTRACT-MIN}(Q)$

Example of Prim's algorithm



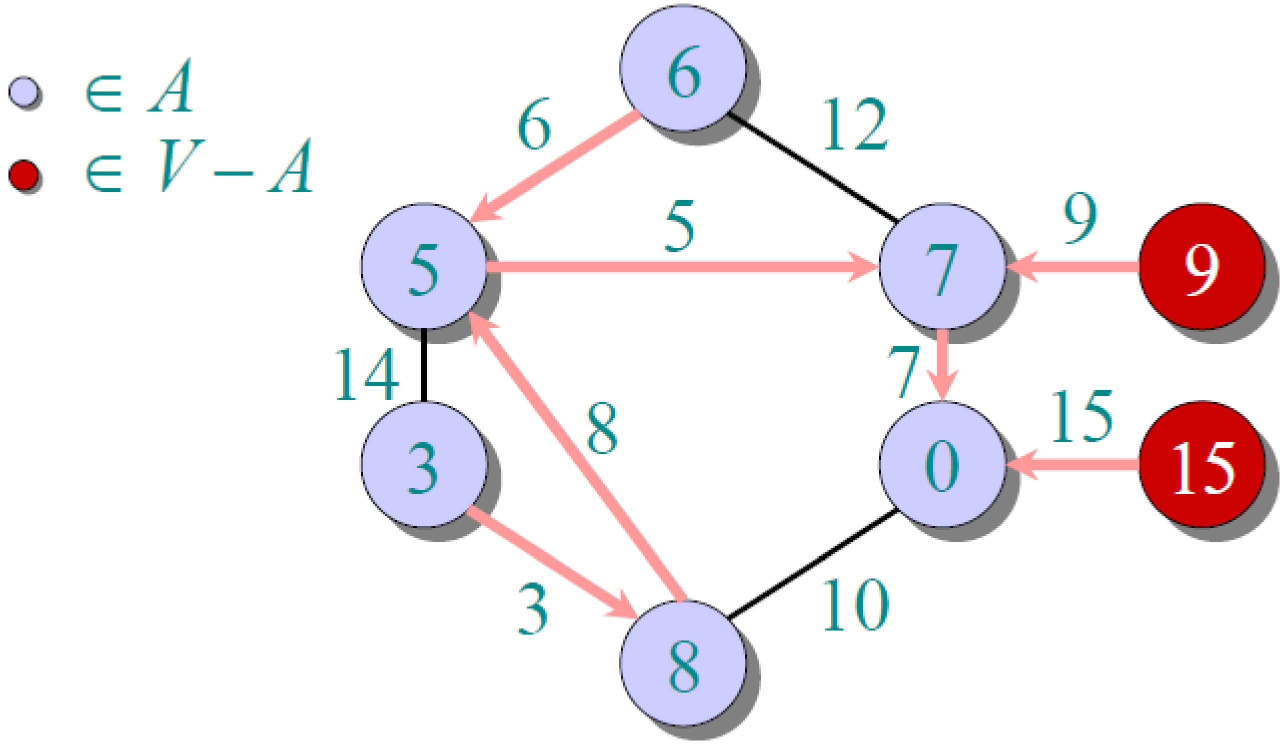
$u \leftarrow \text{EXTRACT-MIN}(Q)$

Example of Prim's algorithm



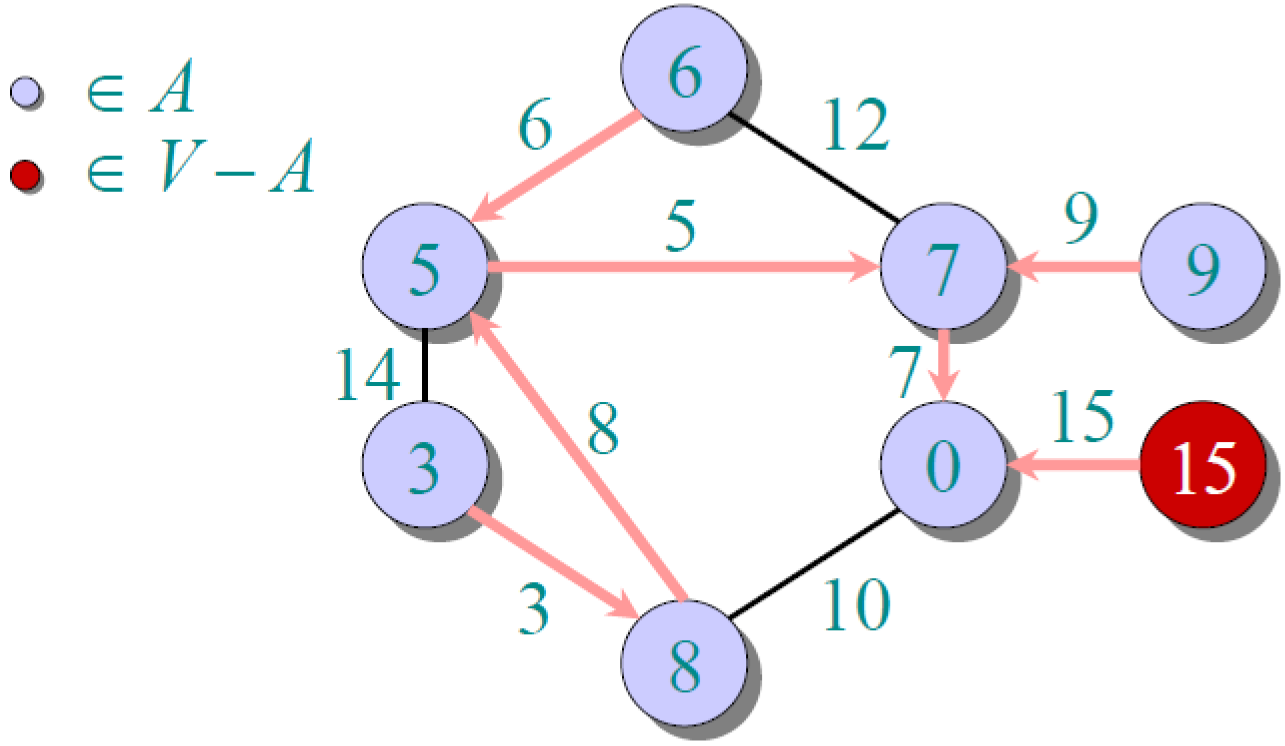
```
for each $v \in Adj[u]$
    do if $v \in Q$ and $w(u, v) < key[v]$
        then $key[v] \leftarrow w(u, v)$    ▷ DECREASE-KEY
             $\pi[v] \leftarrow u$
```

Example of Prim's algorithm



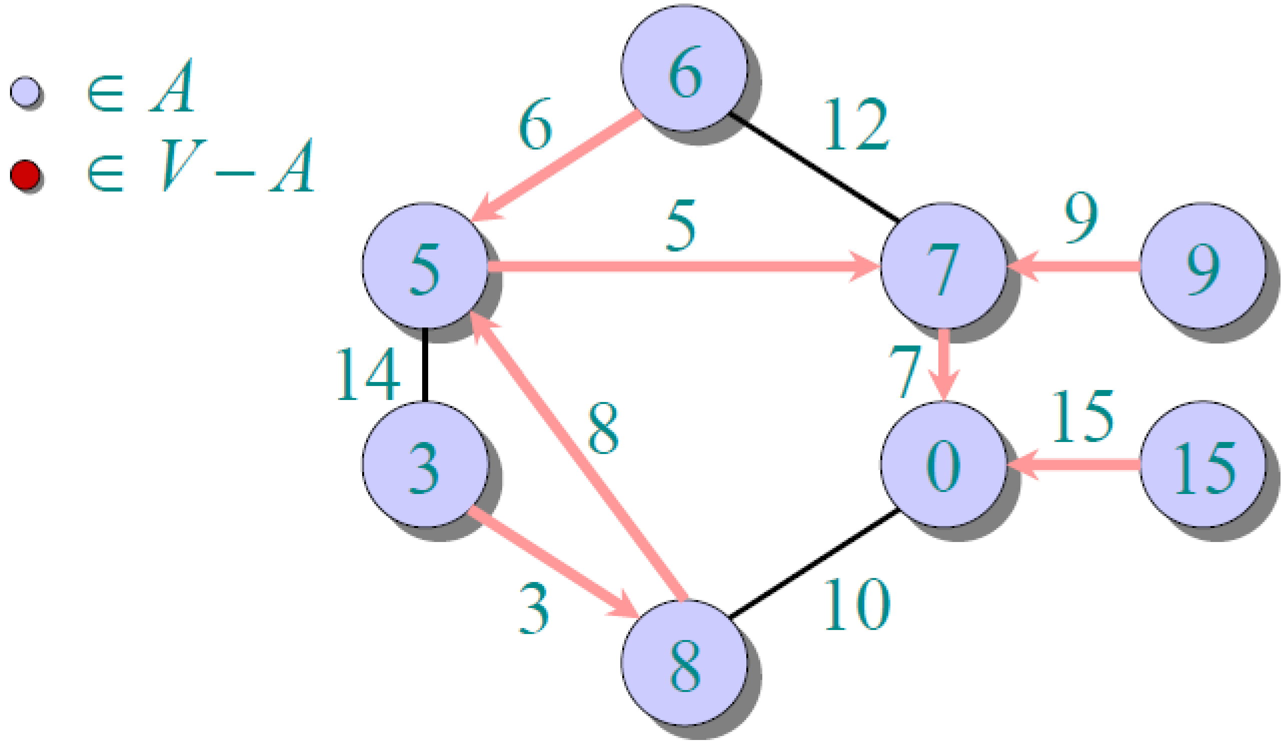
$u \leftarrow$ EXTRACT-MIN(Q)

Example of Prim's algorithm



$u \leftarrow \text{EXTRACT-MIN}(Q)$

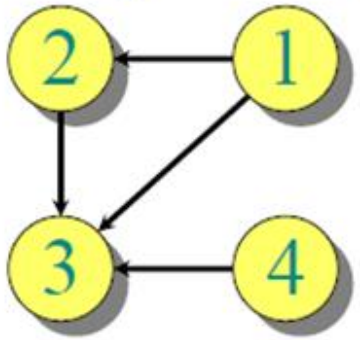
Example of Prim's algorithm



$u \leftarrow \text{Extract-Min}(Q)$

Background: adjacency list

An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.



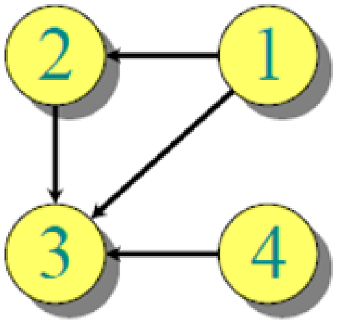
$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

For undirected graphs, $|Adj[v]| = degree(v)$.
For digraphs, $|Adj[v]| = out\text{-}degree(v)$.

Background: adjacency matrix

The ***adjacency matrix*** of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, is the matrix $A[1 \,.\,.\, n, 1 \,.\,.\, n]$ given by

$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{if } (i, j) \notin E. \end{cases}$$



$$\begin{array}{c|cccc} A & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ 2 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 1 & 0 \end{array}$$

$|E| = O(V^2)$

$|E| = \Theta(V^2)$
for dense representation

Analysis of Prim's algorithm

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

$degree(u)$ times

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

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

Analysis of Prim's algorithm

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

assume we keep Q as an array

$$= \Theta(V)\cdot \underbrace{O(V)}_{} + \underbrace{O(V^2)}_{}\cdot \underbrace{O(1)}_{}$$

$$= O(V^2)$$

Analysis of Prim’s algorithm

When it is dense graph, implementing priority queue as an array is OK since the best we can get is $O(V^2)$.

But when the graph is sparse, $E << V^2$, then use <u>binary heap</u> for priority queue of vertices:

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

$$= \Theta(V)\cdot \overbrace{\Theta(\log V)}^{\substack{Q \text{ is a binary heap.}\\ \text{Reorganization}\\ \text{after deleting min}}} + \overbrace{\Theta(E)}\cdot \overbrace{\Theta(\log V)}^{\substack{\text{Reorganization}\\ \text{of binary heap}}}$$

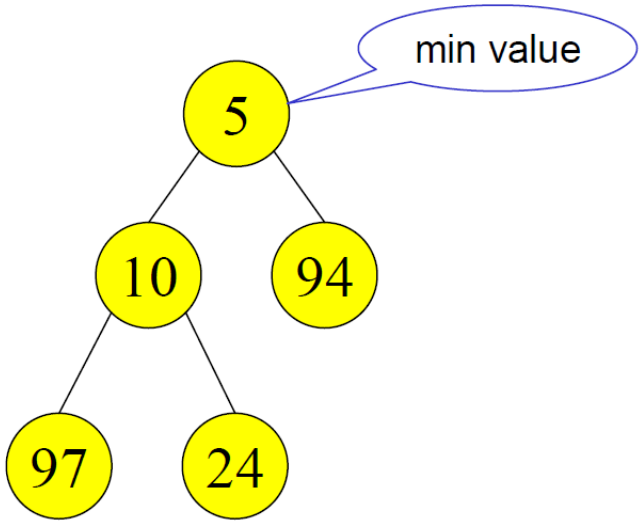
$$= \Theta(E\log V)$$

Analysis of Prim’s algorithm

Background: Binary heap is a data structure from which min (or max) element is taken in a single step.

It is actually a binary tree in which min element is always at the top.

Each parent is less than both children.

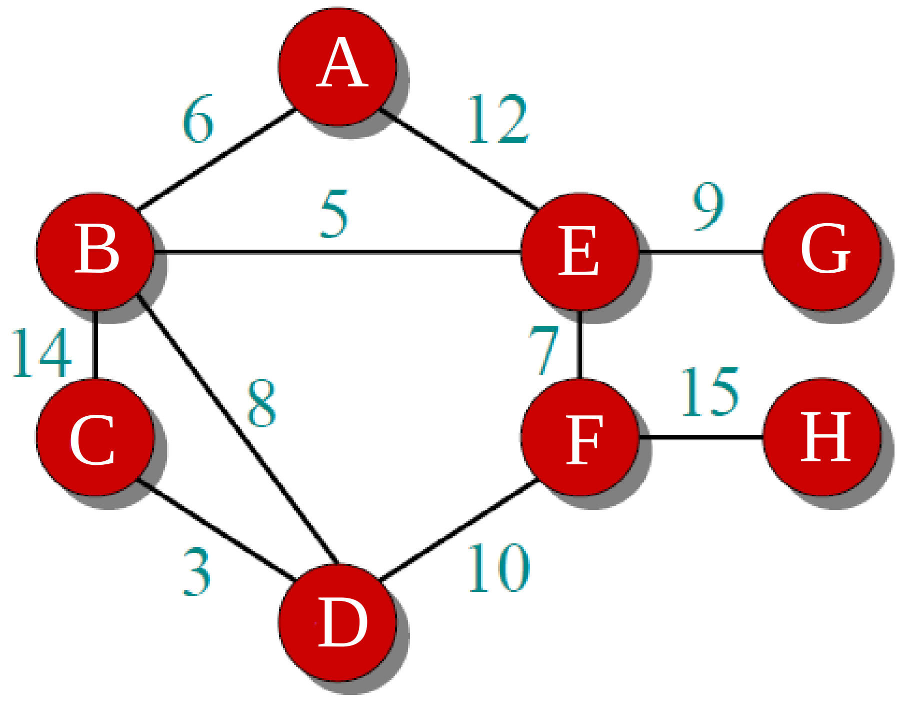


Although taking min is quick, it requires reorganization to put the rest of the elements in min-heap form again, takes *O*(lg *n*), i.e. height of the tree.

For reorganization example, see https://courses.cs.washington.edu/courses/cse373/06sp/handouts/lecture10.pdf

Kruskal's algorithm (also greedy)

1) Consider edges in ascending order of weights.
2) Add next edge to MST unless
   doing so creates a cycle.

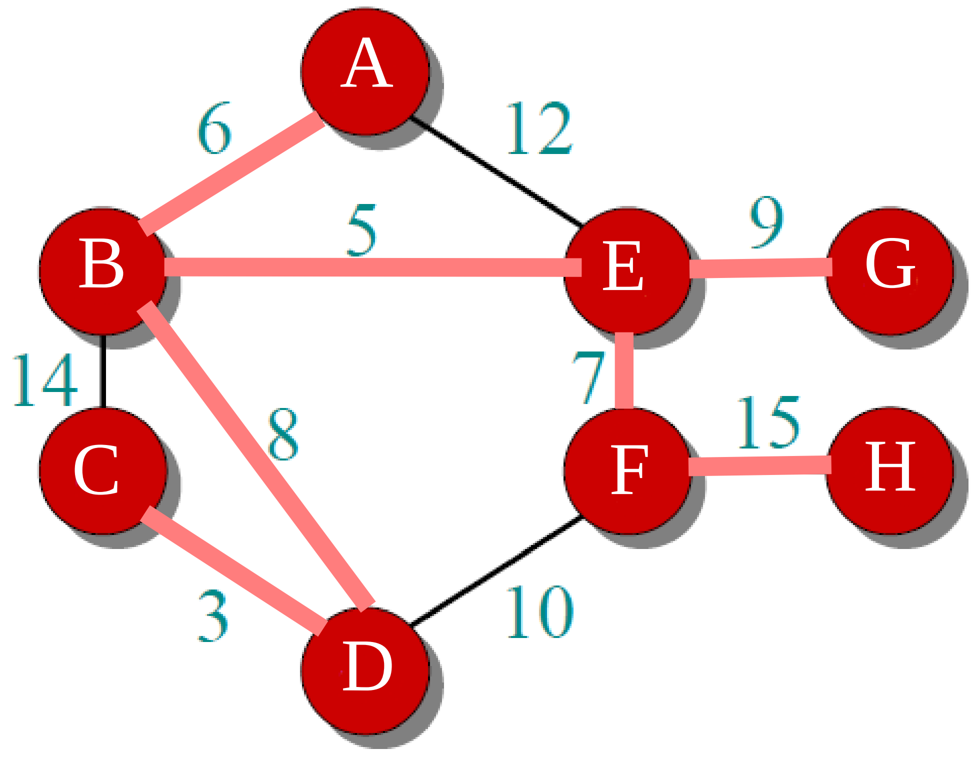


edges sorted
by weight

C-D	3
B-E	5
B-A	6
F-E	7
B-D	8
E-G	9
D-F	10
A-E	12
B-C	14
F-H	15

Kruskal's algorithm

Result of Kruskal's (not surprisingly, same as Prim's).



edges sorted by weight

C-D	3
B-E	5
B-A	6
F-E	7
B-D	8
E-G	9
~~D-F~~	~~10~~
~~A-E~~	~~12~~
~~B-C~~	~~14~~
F-H	15

Kruskal’s algorithm

Q. How to check if a cycle occurs?

A. Can use depth-first-search.

For every visited vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not parent of v, then there is a cycle in graph. If we don’t find such an adjacent for any vertex, we say that there is no cycle.

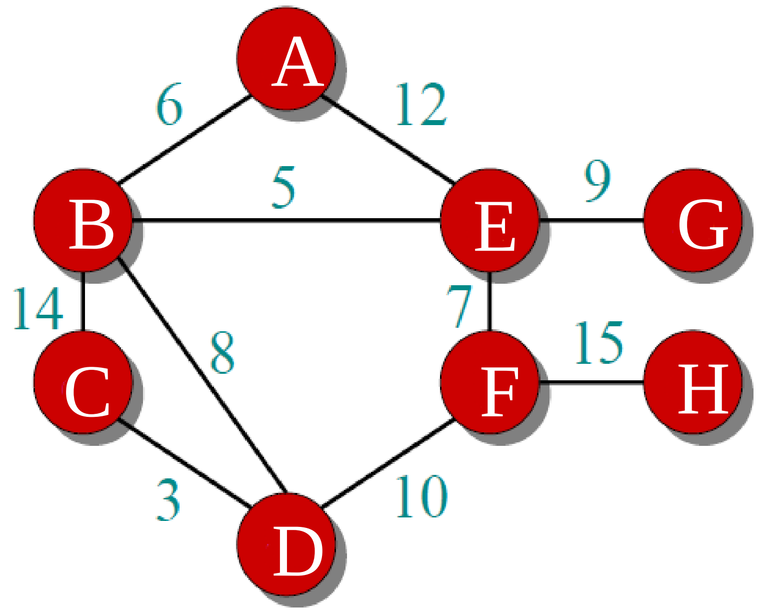
It takes $O(V)$ time, total running time becomes $O(E \cdot V)$

$O(E \lg V)$ is also possible for Kruskal’s algorithm with a different implementation (detailed explanation in textbook).

Other MST algorithms

Boruvka (Sollin) Algorithm:

1) For each vertex, choose lowest weight edge connected to that vertex and add it to a set.
2) Result may not be a connected graph. Add lowest weight edges that connect subgraphs. It is $O(E \lg V)$



Other MST algorithms

Reverse-delete (Reverse Kruskal) Algorithm:
Subtract highest weight edge from graph unless doing so creates a not-connected graph.

Best to date:

- Karger, Klein, and Tarjan [1993].
- Randomized algorithm.
- $O(V + E)$ expected time.

The End

Textbook Section 16.1, 16.2 (Greedy algorithms).
Section 23 (MST)