Design and Analysis of Algorithms

Dynamic Programming and Greedy Techniques

Dynamic problem(overlapping sub problem)

- **Computing a Binomial Coefficient**
- **The Knapsack Problems and Memory Function**
- **Warshall’s and Floyd Algorithms**
  - Warshall’s Algorithms
  - Floyd Algorithms for all pairs Shortest-paths Problems
- **Optimization of Binary Tree**

Dynamic Programming

D*ynamic Programming* is a general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping subinstances

- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS

- “Programming” here means “planning”

- Main idea:
  - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
  - solve smaller instances once
  - record solutions in a table
  - extract solution to the initial instance from that table

Example: Fibonacci numbers

- Recall definition of Fibonacci numbers:

$$F(n) = F(n\text{-}1) + F(n\text{-}2)$$
$$F(0) = 0$$
$$F(1) = 1$$

- Computing the n^{th} Fibonacci number recursively (top-down):

$$F(n)$$

$$F(n\text{-}1) \quad + \quad F(n\text{-}2)$$

$$F(n\text{-}2) \quad + \quad F(n\text{-}3) \qquad F(n\text{-}3) \quad + \quad F(n\text{-}4)$$

...

Example: Fibonacci numbers (cont.)

Computing the n^{th} Fibonacci number using bottom-up iteration and recording results:

$F(0) = 0$
$F(1) = 1$
$F(2) = 1+0 = 1$
…
$F(n-2) =$
$F(n-1) =$
$F(n) = F(n-1) + F(n-2)$

0	1	1	. . .	$F(n-2)$	$F(n-1)$	$F(n)$

Efficiency:
- **time** n
- **space** n

What if we solve it recursively?

Examples of DP algorithms

- Computing a binomial coefficient
- Longest common subsequence
- Warshall’s algorithm for transitive closure
- Floyd’s algorithm for all-pairs shortest paths
- Constructing an optimal binary search tree
- Some instances of difficult discrete optimization problems:
  - traveling salesman
  - knapsack

Computing a binomial coefficient by DP

Binomial coefficients are coefficients of the binomial formula:

$$(a + b)^n = C(n,0)a^n b^0 + \ldots + C(n,k)a^{n-k}b^k + \ldots + C(n,n)a^0 b^n$$

Recurrence: $C(n,k) = C(n\text{-}1,k) + C(n\text{-}1,k\text{-}1)$ **for** $n > k > 0$

$C(n,0) = 1, \quad C(n,n) = 1$ **for** $n \geq 0$

Value of $C(n,k)$ **can be computed by filling a table:**

	0	1	2	. . .	k-1	k
0	1					
1	1	1				
.						
.						
.						
n-1					$C(n\text{-}1,k\text{-}1)$	$C(n\text{-}1,k)$
n						$C(n,k)$

Computing *C*(*n,k*): pseudocode and analysis

ALGORITHM *Binomial*(n, k)
    //Computes $C(n, k)$ by the dynamic programming algorithm
    //Input: A pair of nonnegative integers $n \geq k \geq 0$
    //Output: The value of $C(n, k)$
    for $i \leftarrow 0$ **to** n **do**
        for $j \leftarrow 0$ **to** $\min(i, k)$ **do**
            if $j = 0$ **or** $j = i$
                $C[i, j] \leftarrow 1$
            else $C[i, j] \leftarrow C[i-1, j-1] + C[i-1, j]$
    return $C[n, k]$

Space efficiency: $\Theta(nk)$

Knapsack Problem by DP

Given n items of

integer weights: $w_1 \ w_2 \ \ldots \ w_n$

values: $v_1 \ v_2 \ \ldots \ v_n$

a knapsack of integer capacity W

find most valuable subset of the items that fit into the knapsack

Consider instance defined by first i items and capacity j ($j \le W$).

Let $V[i,j]$ be optimal value of such an instance. Then

$$V[i,j] = \begin{cases} \max\{V[i-1,j],\ v_i + V[i-1,j-w_i]\} & \text{if } j-w_i \ge 0 \\ V[i-1,j] & \text{if } j-w_i < 0 \end{cases}$$

Knapsack Problem by DP (example)

Example: Knapsack of capacity $W = 5$

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity j

	i	0	1	2	3	4	5
	0	0	0	0			
$w_1 = 2, V_1 = 12$	1	0	0	12			
$w_2 = 1, V_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, V_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2, V_4 = 15$	4	0	10	15	25	30	37

Backtracing finds the actual optimal subset, i.e. solution.

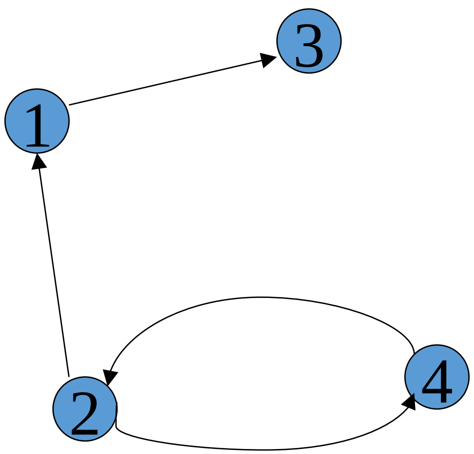
Knapsack Problem by DP (pseudocode)

```
Algorithm DPKnapsack(w[1..n], v[1..n], W)
  var V[0..n,0..W], P[1..n,1..W]: int
  for j := 0 to W do
       V[0,j] := 0
    for i := 0 to n do
         V[i,0] := 0
    for i := 1 to n do
       for j := 1 to W do
           if  w[i] ≤ j and v[i] + V[i-1,j-w[i]] > V[i-1,j] then
                V[i,j] := v[i] + V[i-1,j-w[i]]; P[i,j] := j-w[i]
           else
                V[i,j] := V[i-1,j]; P[i,j] := j
  return V[n,W] and the optimal subset by backtracing
```

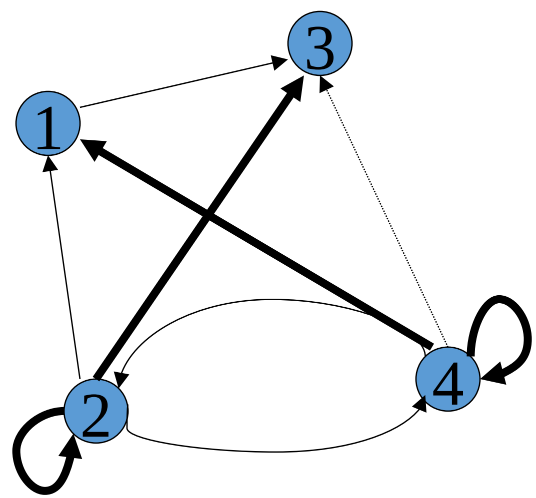
Running time and space: $O(nW)$.

Warshall's Algorithm: Transitive Closure

- **Computes the transitive closure of a relation(Reachabilty)**
- **Alternatively: existence of all nontrivial paths in a digraph**
- **Example of transitive closure:**



$$\begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$

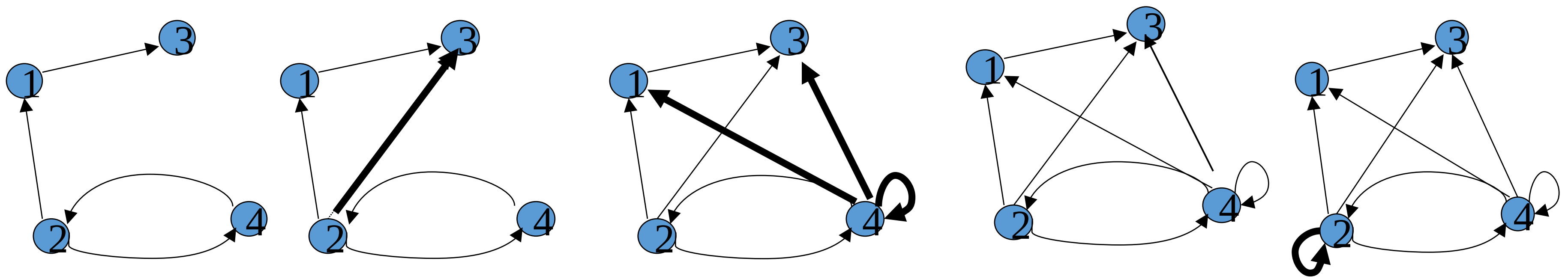


$$\begin{matrix} 0 & 0 & 1 & 0 \\ 1 & \mathbf{1} & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & \mathbf{1} & \mathbf{1} \end{matrix}$$

Warshall's Algorithm

Constructs transitive closure T as the last matrix in the sequence of n-by-n matrices $R^{(0)}, \ldots, R^{(k)}, \ldots, R^{(n)}$ where $R^{(k)}[i,j] = 1$ iff there is nontrivial path from i to j with only the first k vertices allowed as intermediate

Note that $R^{(0)} = A$ (adjacency matrix), $R^{(n)} = T$ (transitive closure)

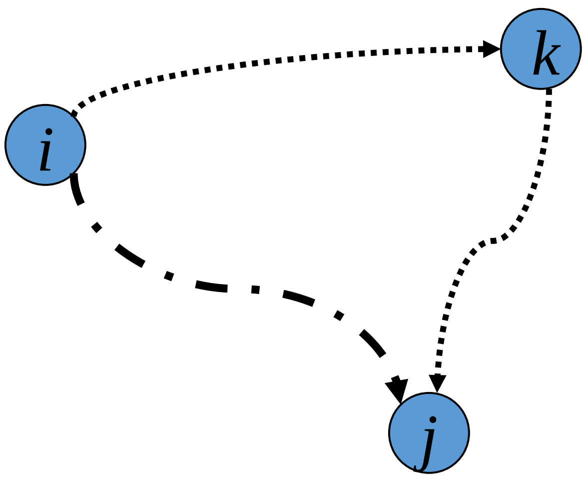


$$R^{(0)} \quad \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix} \qquad R^{(1)} \quad \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix} \qquad R^{(2)} \quad \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & \mathbf{1} & \mathbf{1} \end{matrix} \qquad R^{(3)} \quad \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix} \qquad R^{(4)} \quad \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & \mathbf{1} & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix}$$

Warshall's Algorithm (recurrence)

On the k-th iteration, the algorithm determines for every pair of vertices i, j if a path exists from i and j with just vertices $1,\ldots,k$ allowed as intermediate

$$R^{(k)}[i,j] = \begin{cases} R^{(k-1)}[i,j] & \text{(path using just } 1,\ldots,k\text{-1)} \\ \quad\text{or} & \\ R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j] & \text{(path from } i \text{ to } k \\ & \quad \text{and from } k \text{ to } j \\ & \quad \text{using just } 1,\ldots,k\text{-1)} \end{cases}$$



Warshall's Algorithm (matrix generation)

Recurrence relating elements $R^{(k)}$ to elements of $R^{(k-1)}$ is:

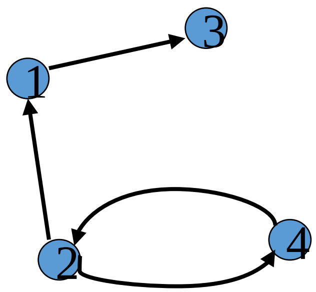
$$R^{(k)}[i,j] = R^{(k-1)}[i,j] \text{ or } (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j])$$

It implies the following rules for generating $R^{(k)}$ from $R^{(k-1)}$:

Rule 1 If an element in row i and column j is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$

Rule 2 If an element in row i and column j is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row i and column k and the element in its column j and row k are both 1's in $R^{(k-1)}$

Warshall's Algorithm (example)



$$R^{(0)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$

$$R^{(1)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$

$$R^{(2)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & \mathbf{1} & \mathbf{1} \end{matrix}$$

$$R^{(3)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix}$$

$$R^{(4)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & \mathbf{1} & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix}$$

Warshall's Algorithm (pseudocode and analysis)

ALGORITHM *Warshall*($A[1..n, 1..n]$)

//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
$R^{(0)} \leftarrow A$
for $k \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** $(R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j])$
return $R^{(n)}$

Time efficiency: Θ(n^3)

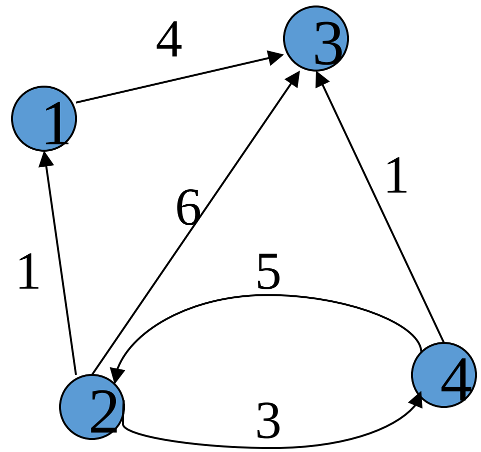
Space efficiency: Matrices can be written over their predecessors (with some care), so it's Θ(n^2).

Floyd's Algorithm: All pairs shortest paths

Problem: **In a weighted (di)graph, find shortest paths between every pair of vertices**

Same idea: construct solution through series of matrices $D^{(0)}, \ldots, D^{(n)}$ using increasing subsets of the vertices allowed as intermediate

Example:

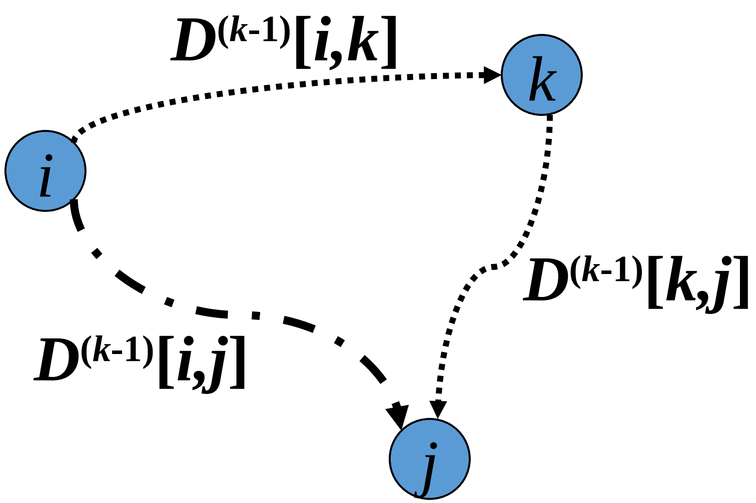


0	∞	4	∞
1	0	4	3
∞	∞	0	∞
6	5	1	0

Floyd's Algorithm (matrix generation)

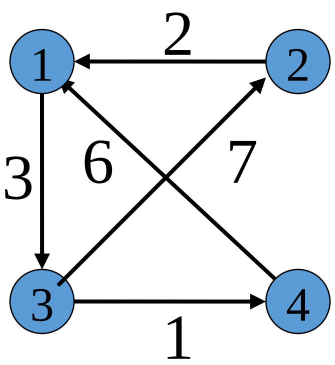
On the *k*-th iteration, the algorithm determines shortest paths between every pair of vertices *i, j* that use only vertices among 1, …,*k* as intermediate

$$D^{(k)}[i,j] = \min\{D^{(k-1)}[i,j],\ D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}$$



Initial condition?

Floyd’s Algorithm (example)



$$D^{(0)} = \begin{matrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{matrix}$$

$$D^{(1)} = \begin{matrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{matrix}$$

$$D^{(2)} = \begin{matrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{matrix}$$

$$D^{(3)} = \begin{matrix} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{matrix}$$

$$D^{(4)} = \begin{matrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ \mathbf{7} & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{matrix}$$

Floyd's Algorithm (pseudocode and analysis)

ALGORITHM *Floyd*($W[1..n, 1..n]$)

//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
$D \leftarrow W$ //is not necessary if W can be overwritten
for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
        for $j \leftarrow 1$ **to** n **do**
            $D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$
return D

Time efficiency: Θ(n^3)

Since the superscripts *k* or *k-1* make no difference to $D[i,k]$ and $D[k,j]$.

Space efficiency: Matrices can be written over their predecessors

Note: Works on graphs with negative edges but without negative cycles. Shortest paths themselves can be found, too.

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.

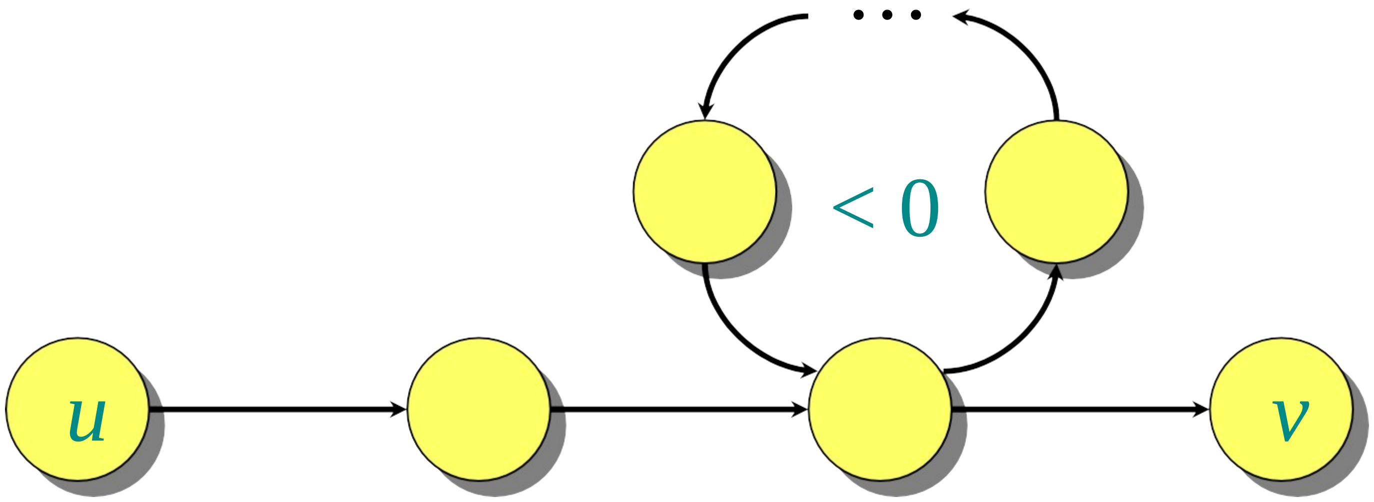
Well-definedness of shortest paths

If a graph G contains a negative-weight cycle, then some shortest paths may not exist.

Well-definedness of shortest paths

If a graph G contains a negative-weight cycle, then some shortest paths may not exist.

Example:



Single-source shortest paths

Problem. From a given source vertex $s \in V$, find the shortest-path weights $\delta(s, v)$ for all $v \in V$.

If all edge weights $w(u, v)$ are *nonnegative*, all shortest-path weights must exist.

IDEA: Greedy.

1. Maintain a set S of vertices whose shortest-path distances from s are known.
2. At each step add to S the vertex $v \in V - S$ whose distance estimate from s is minimal.
3. Update the distance estimates of vertices adjacent to v.

Dijkstra's algorithm

$d[s] \leftarrow 0$

for each $v \in V - \{s\}$

do $d[v] \leftarrow \infty$

$S \leftarrow \varnothing$

$Q \leftarrow V$ $\triangleright$ Q is a priority queue maintaining $V - S$

Dijkstra's algorithm

$d[s] \leftarrow 0$
for each $v \in V - \{s\}$
do $d[v] \leftarrow \infty$
$S \leftarrow \emptyset$
$Q \leftarrow V$ ⊳ Q is a priority queue maintaining $V - S$
while $Q \neq \emptyset$
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 $d[v] \leftarrow d[u] + w(u, v)$

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 $v \in Adj[u]$

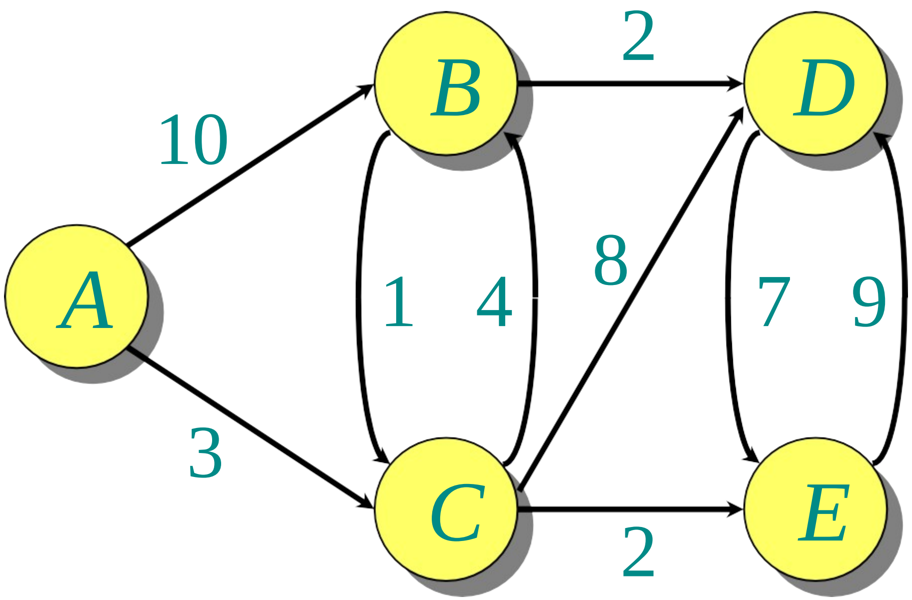
do if $d[v] > d[u] + w(u, v)$

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

relaxation step

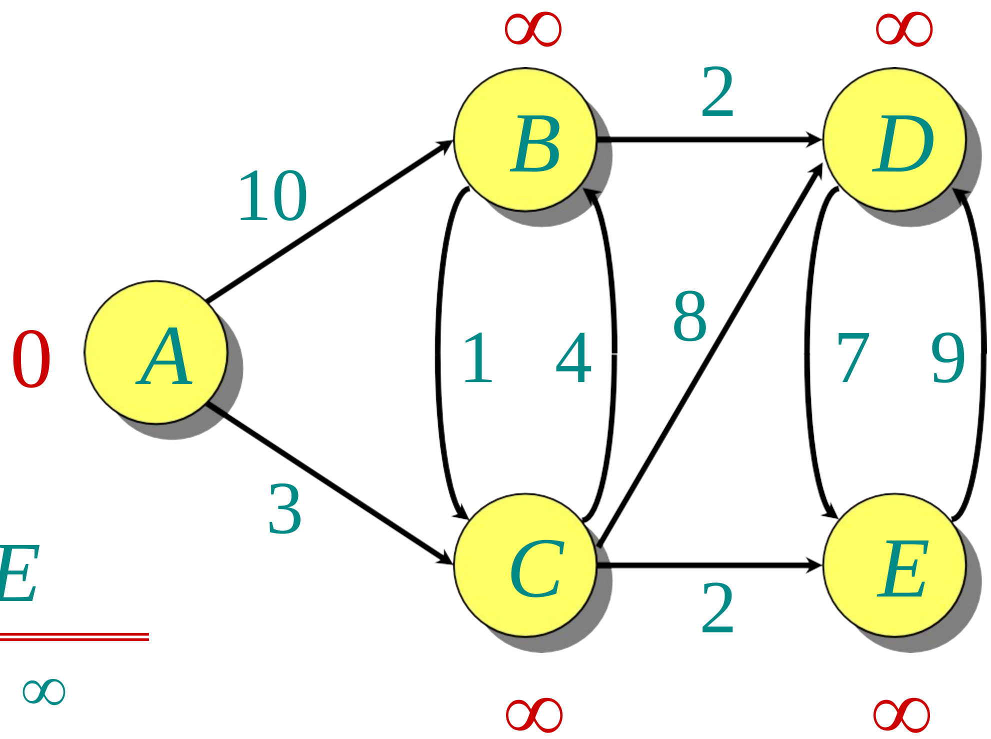
Example of Dijkstra's algorithm

Graph with nonnegative edge weights:



Example of Dijkstra's algorithm

Initialize:



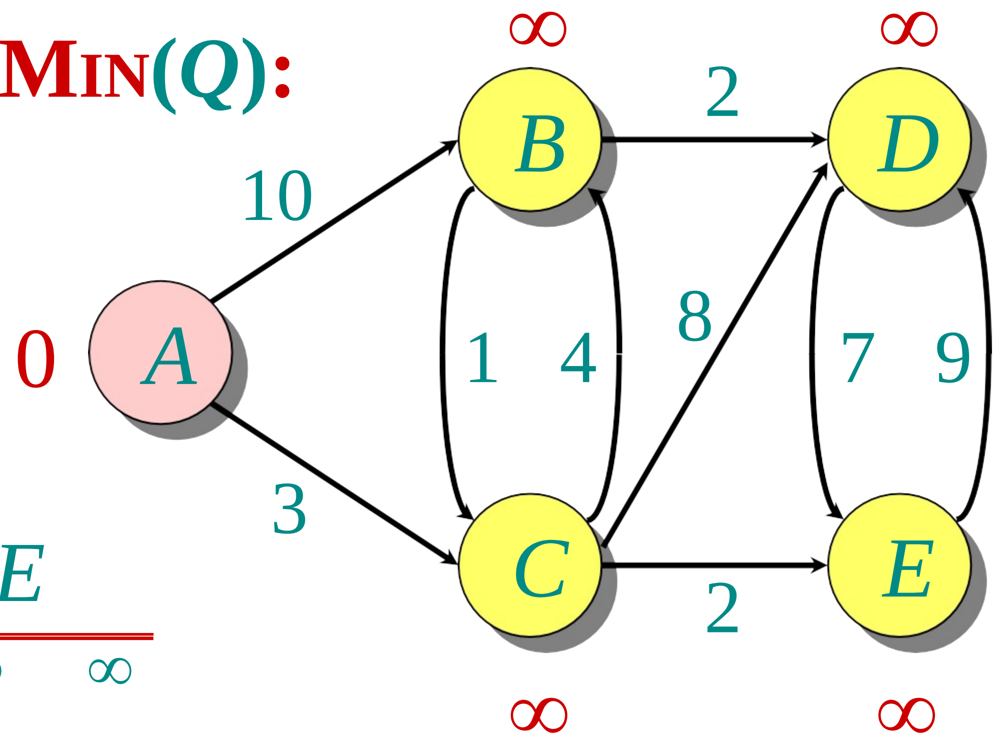
Q: *A B C D E*

0 ∞ ∞ ∞ ∞

S: { }

Example of Dijkstra’s algorithm

“*A*” ← EXTRACT-MIN(*Q*):

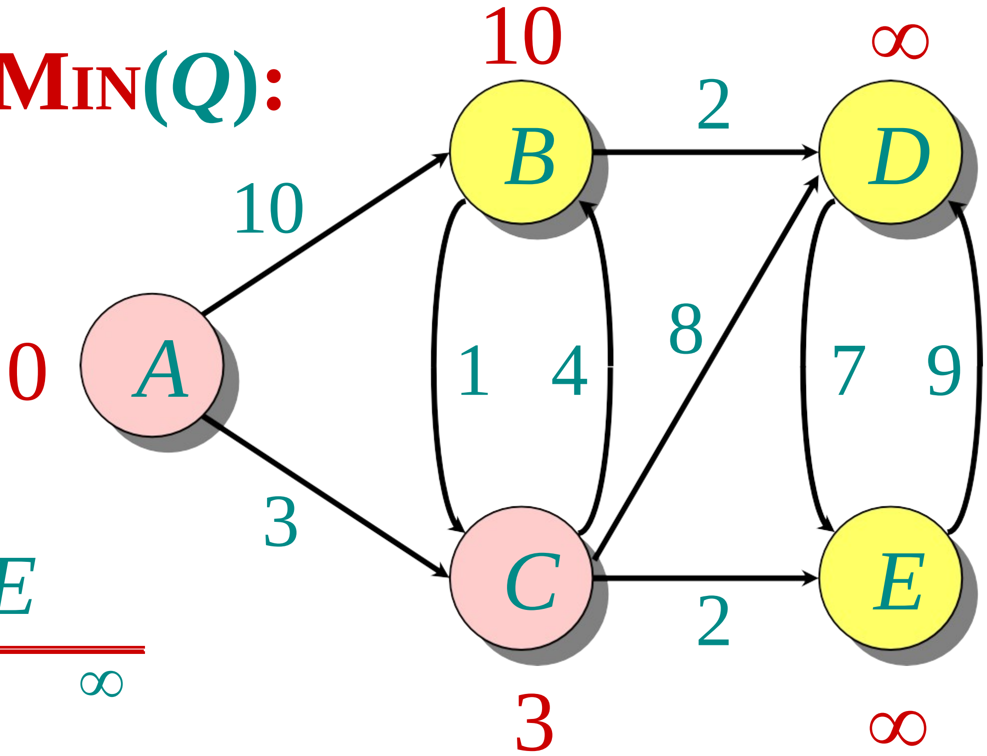


Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: { A }

Example of Dijkstra's algorithm

"*C*" ← EXTRACT-MIN(*Q*):

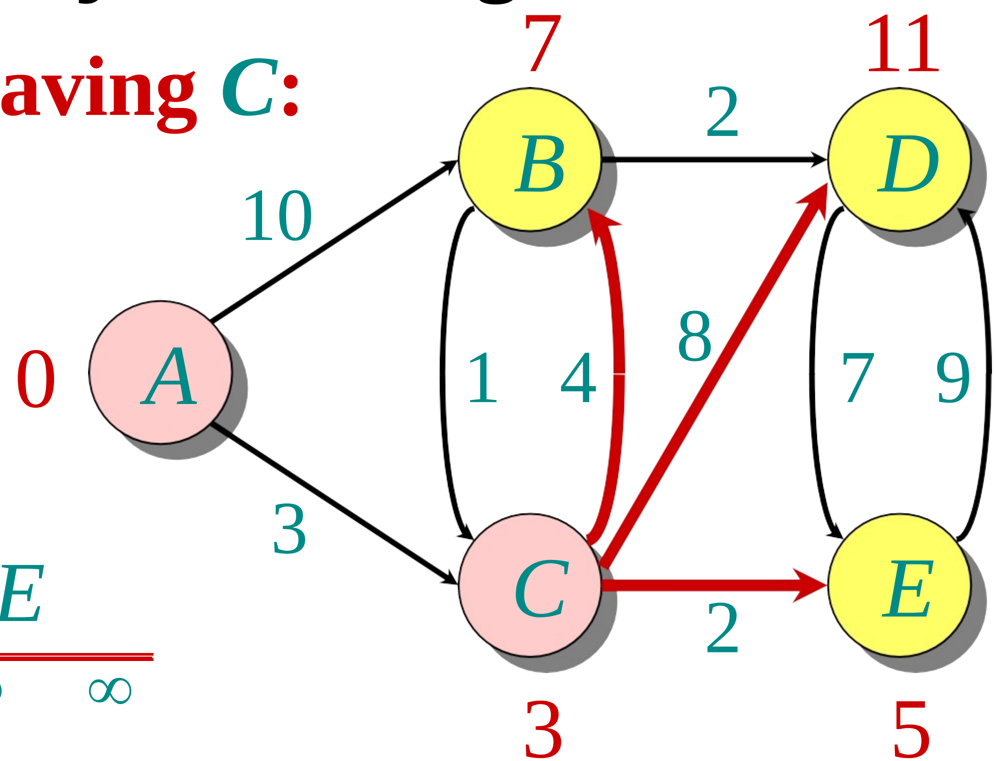


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { *A*, *C* }

Example of Dijkstra’s algorithm

Relax all edges leaving *C*:

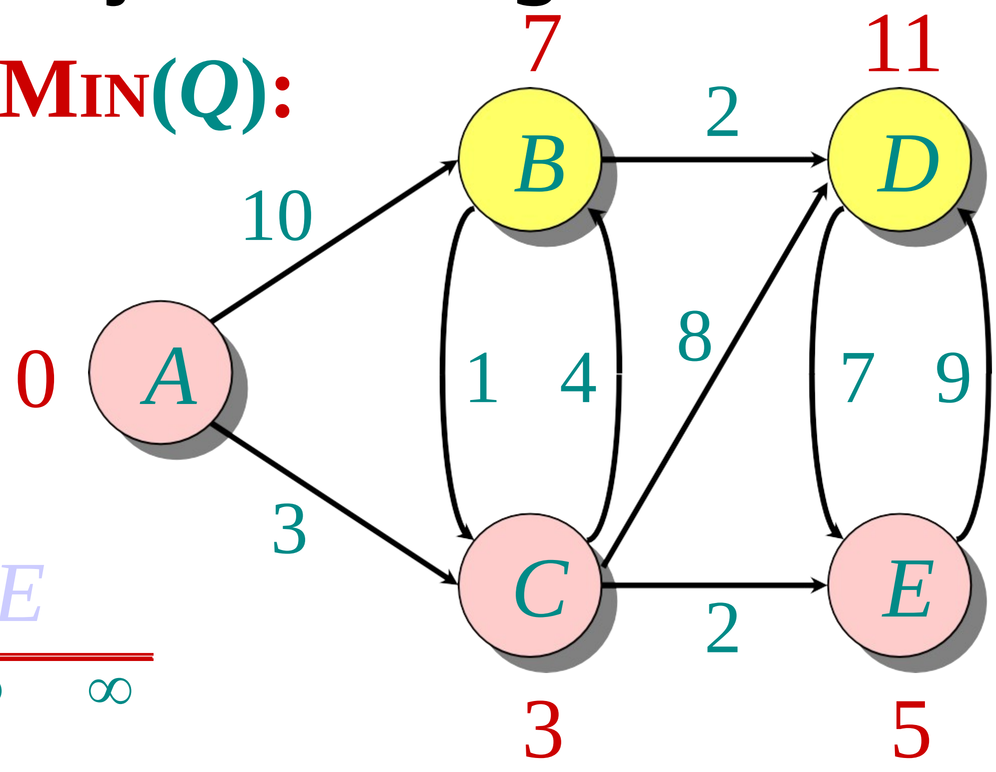


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { *A*, *C* }

Example of Dijkstra’s algorithm

“E” ← **EXTRACT-MIN(*Q*):**

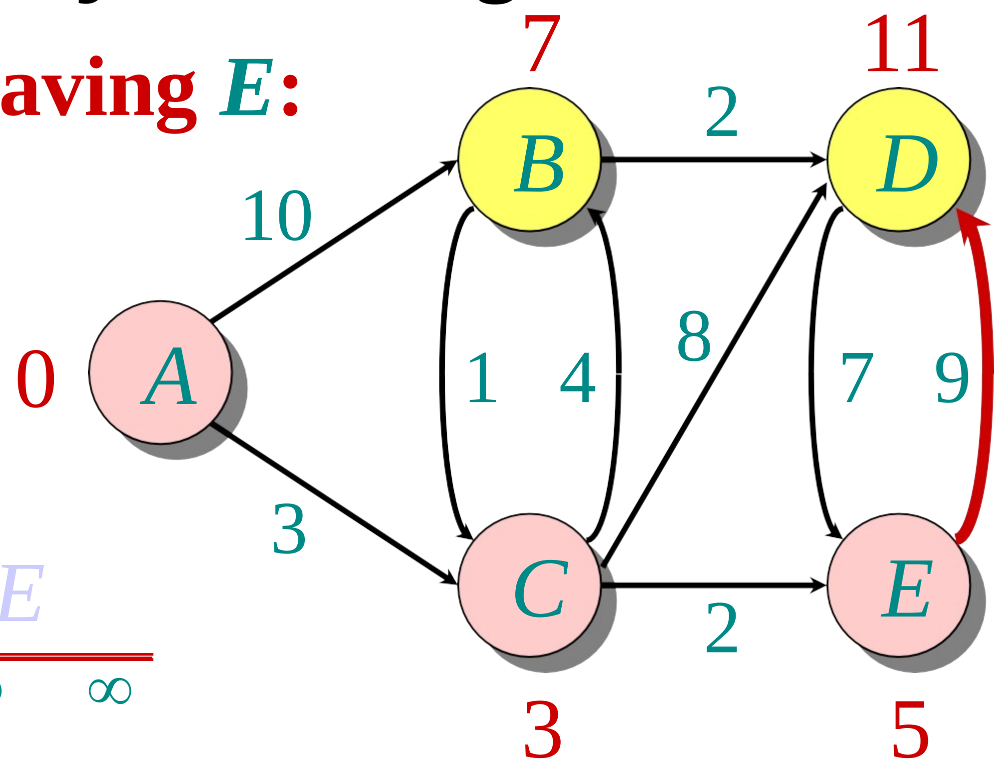


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { *A*, *C*, *E* }

Example of Dijkstra's algorithm

Relax all edges leaving *E*:

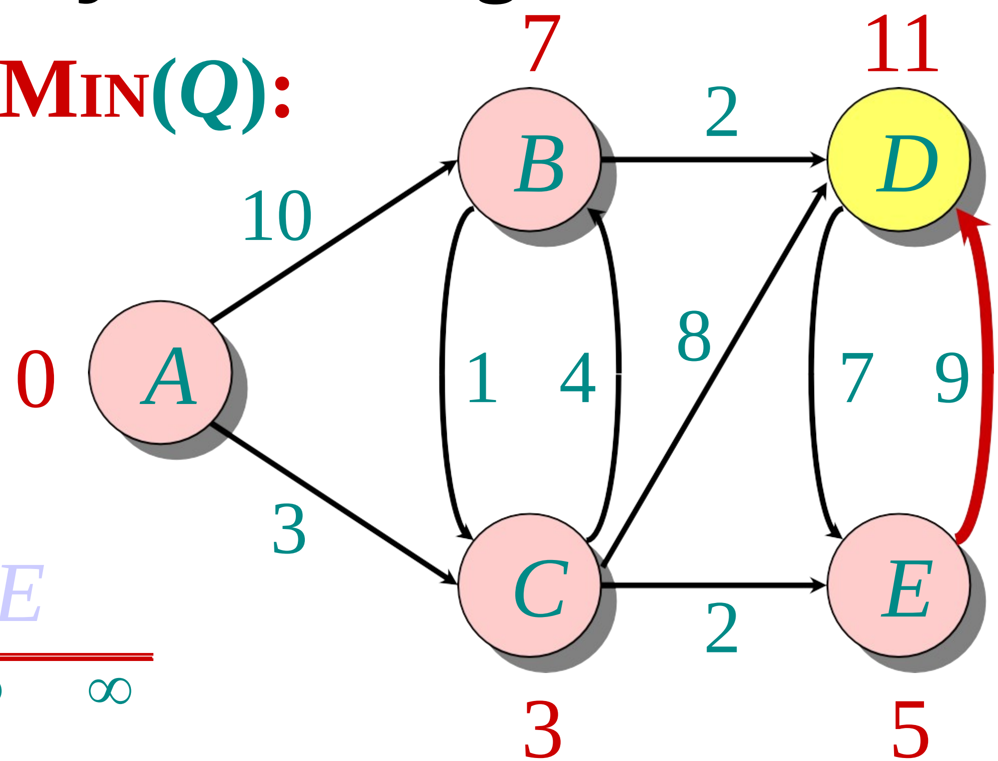


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E }

Example of Dijkstra's algorithm

"*B*" ← EXTRACT-MIN(*Q*):

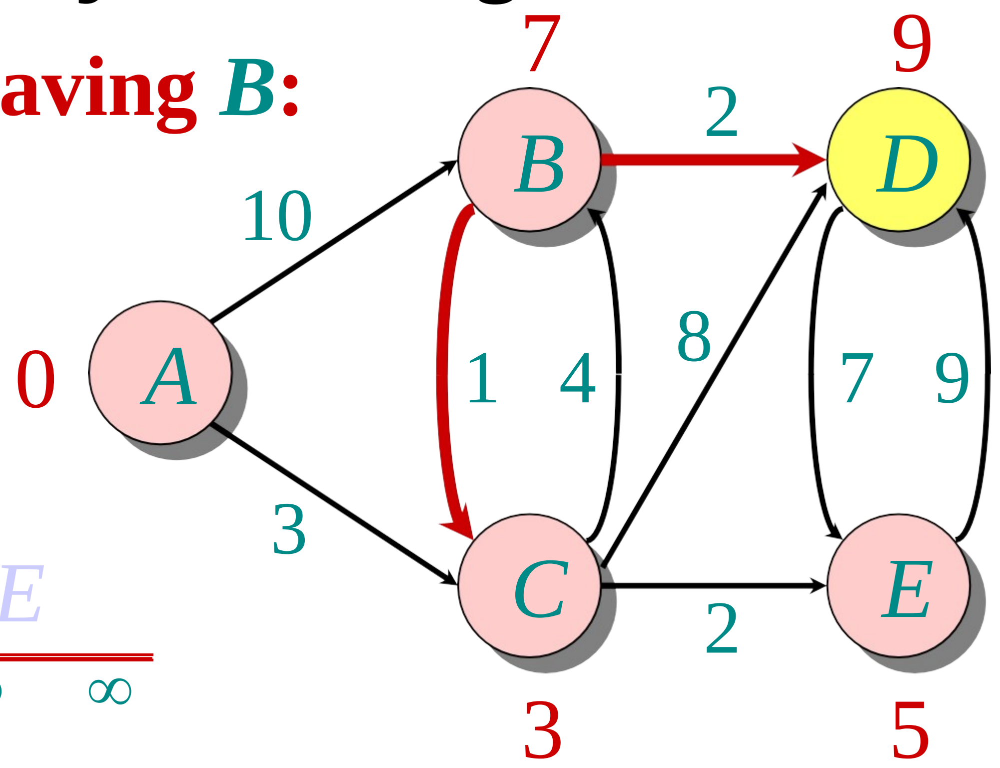


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { *A*, *C*, *E*, *B* }

Example of Dijkstra's algorithm

Relax all edges leaving *B*:

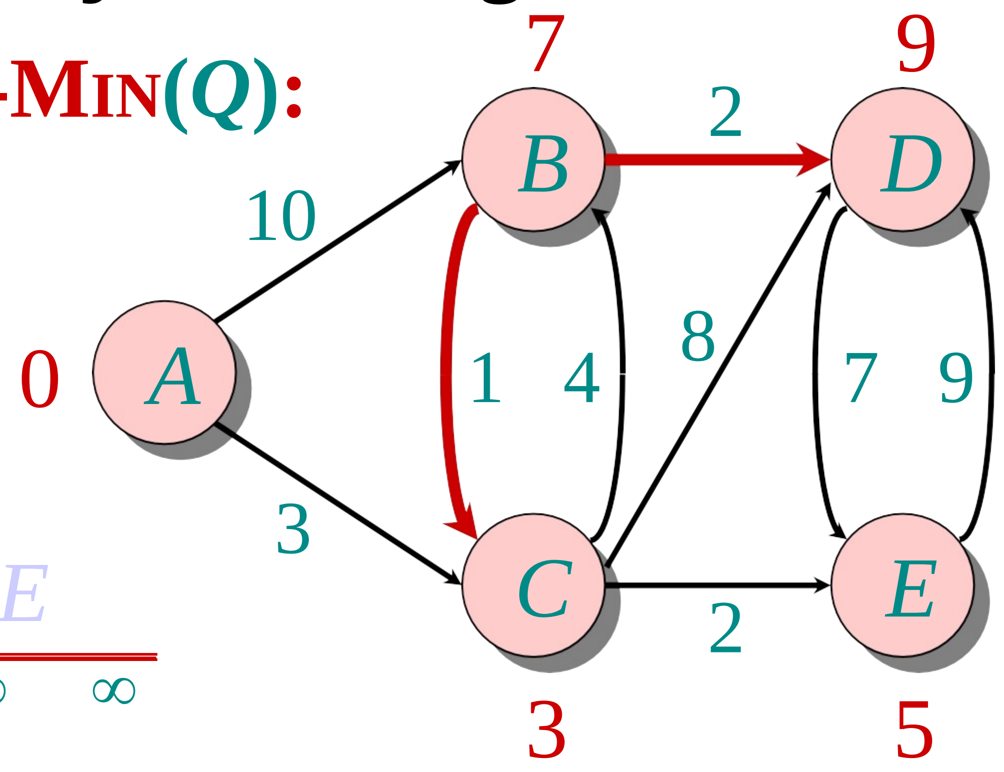


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { *A*, *C*, *E*, *B* }

Example of Dijkstra’s algorithm

“D” ← **EXTRACT-MIN(*Q*):**



Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { *A*, *C*, *E*, *B*, *D* }

Analysis of Dijkstra

while $Q \neq \emptyset$
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 $d[v] \leftarrow$
$d[u] + w(u, v)$

Analysis of Dijkstra

$|V|$ times

while $Q \neq \emptyset$
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 $d[v] \leftarrow d[u] + w(u, v)$

Analysis of Dijkstra

$|V|$ times

- while $Q \neq \emptyset$
  - do $u \leftarrow$ EXTRACT-MIN(Q)
  - $S \leftarrow S \cup \{u\}$
  - for each $v \in Adj[u]$

$degree(u)$ times

do if $d[v] > d[u] + w(u, v)$

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

Analysis of Dijkstra

$|V|$ times

- while $Q \neq \emptyset$
  - do $u \leftarrow$ EXTRACT-MIN(Q)
  - $S \leftarrow S \cup \{u\}$
  - for each $v \in Adj[u]$

$degree(u)$ times

do if $d[v] > d[u] + w(u, v)$

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

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

Analysis of Dijkstra

$|V|$ times

- while $Q \neq \emptyset$
  - do $u \leftarrow \text{EXTRACT-MIN}(Q)$
  - $S \leftarrow S \cup \{u\}$
  - for each $v \in Adj[u]$

$degree(u)$ times

do if $d[v] > d[u] + w(u, v)$

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

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

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

Note: Same formula as in the analysis of Prim's minimum spanning tree algorithm.

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.

Minimum spanning trees

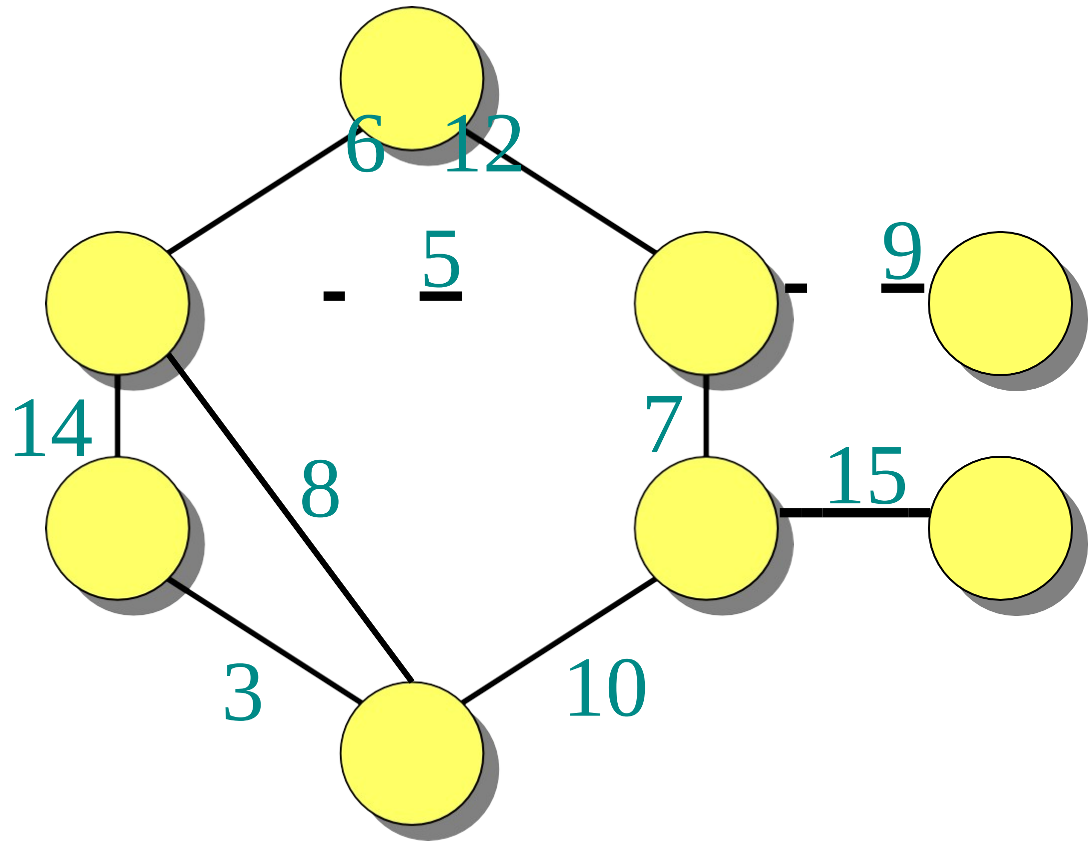
Input: A connected, undirected graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$.

- For simplicity, assume that all edge weights are distinct.

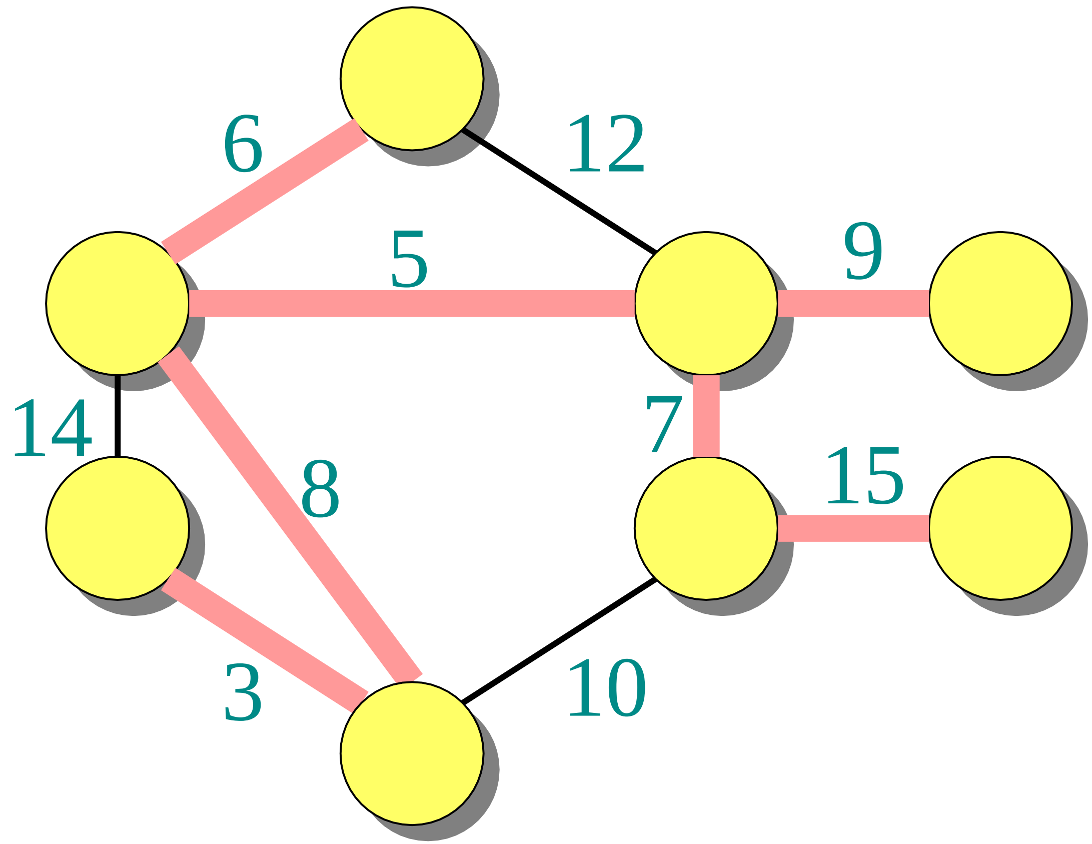
Output: A ***spanning tree*** T — a tree that connects all vertices — of minimum weight:

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

Example of MST



Example of MST



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

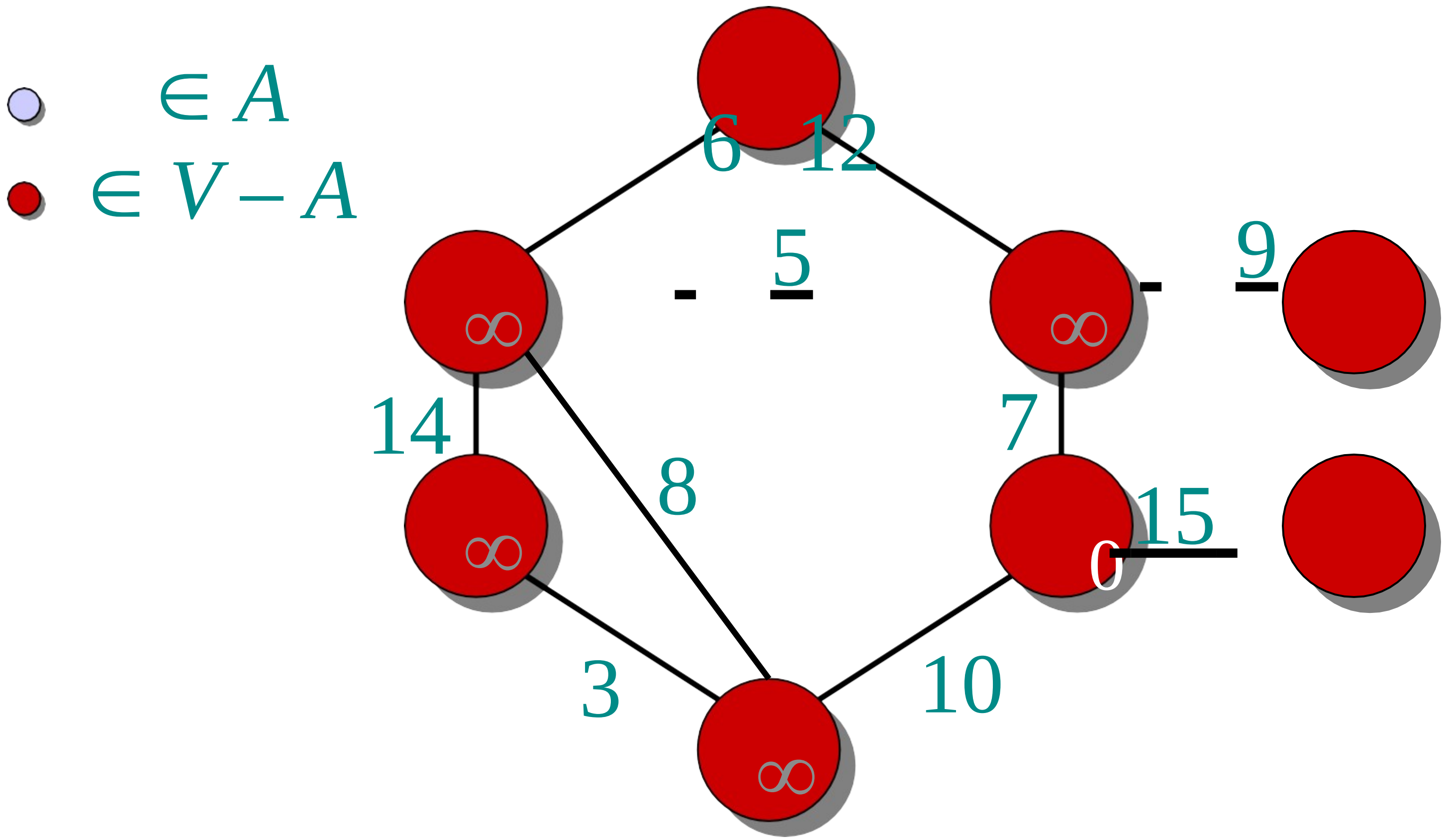
Prim's algorithm

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.

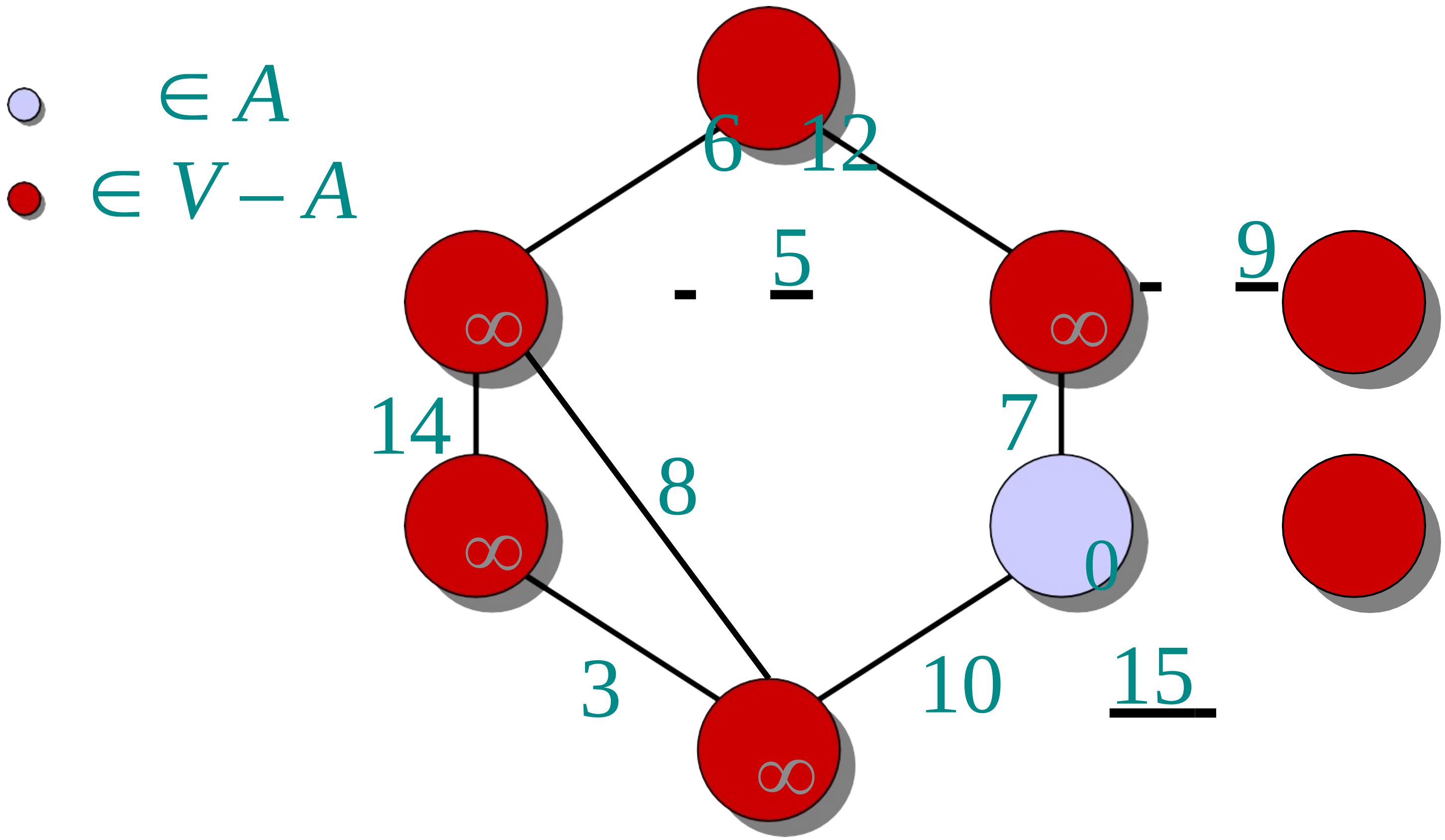
$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$
while $Q \neq \emptyset$
do $u \leftarrow$ EXTRACT-MIN(Q)
  for each $v \in Adj[u]$
    do if $v \in Q$ and $w(u, v) < key[v]$
      then $key[v] \leftarrow w(u, v)$ ⊳ DECREASE-KEY
      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

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

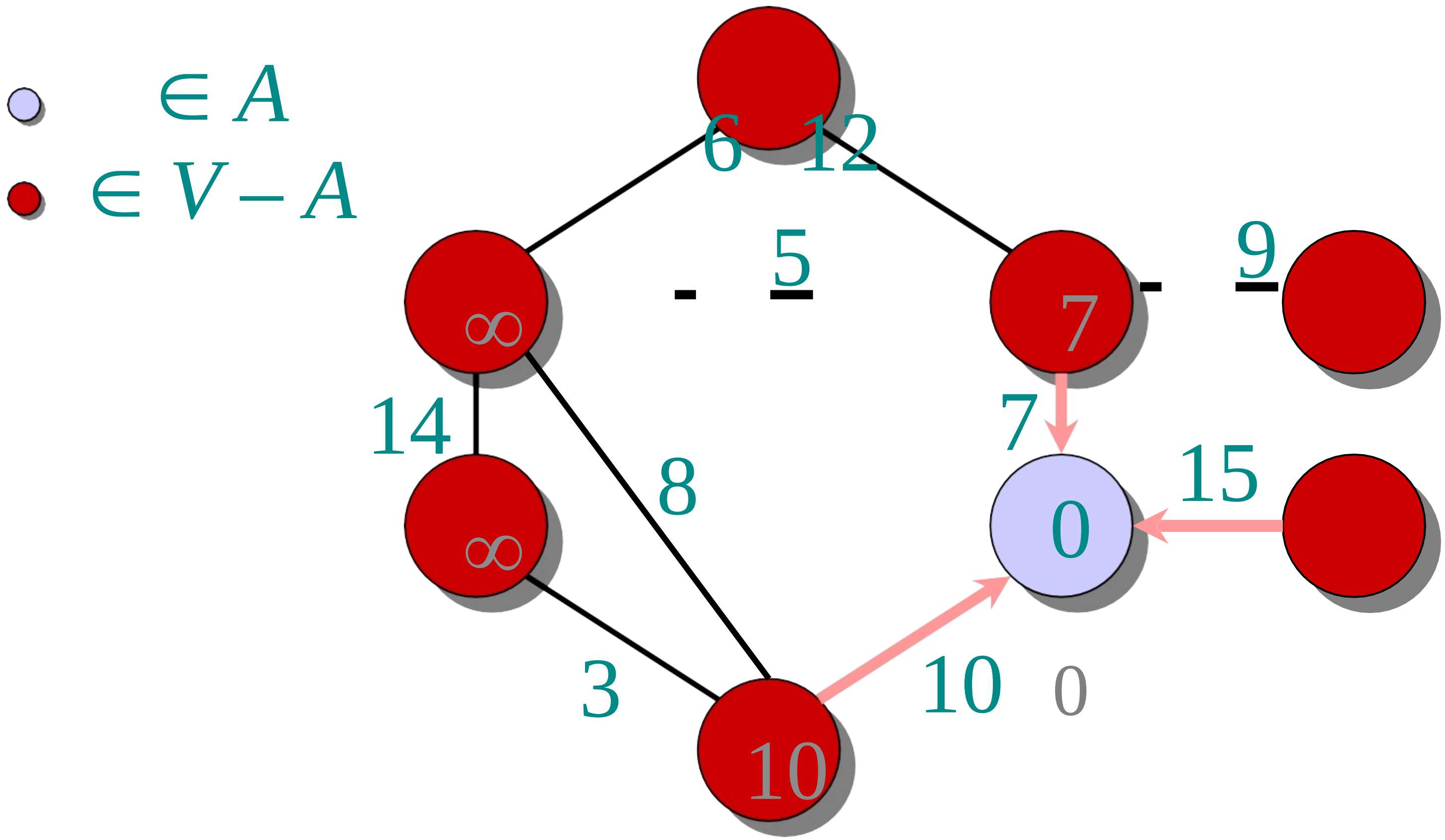
Example of Prim's algorithm



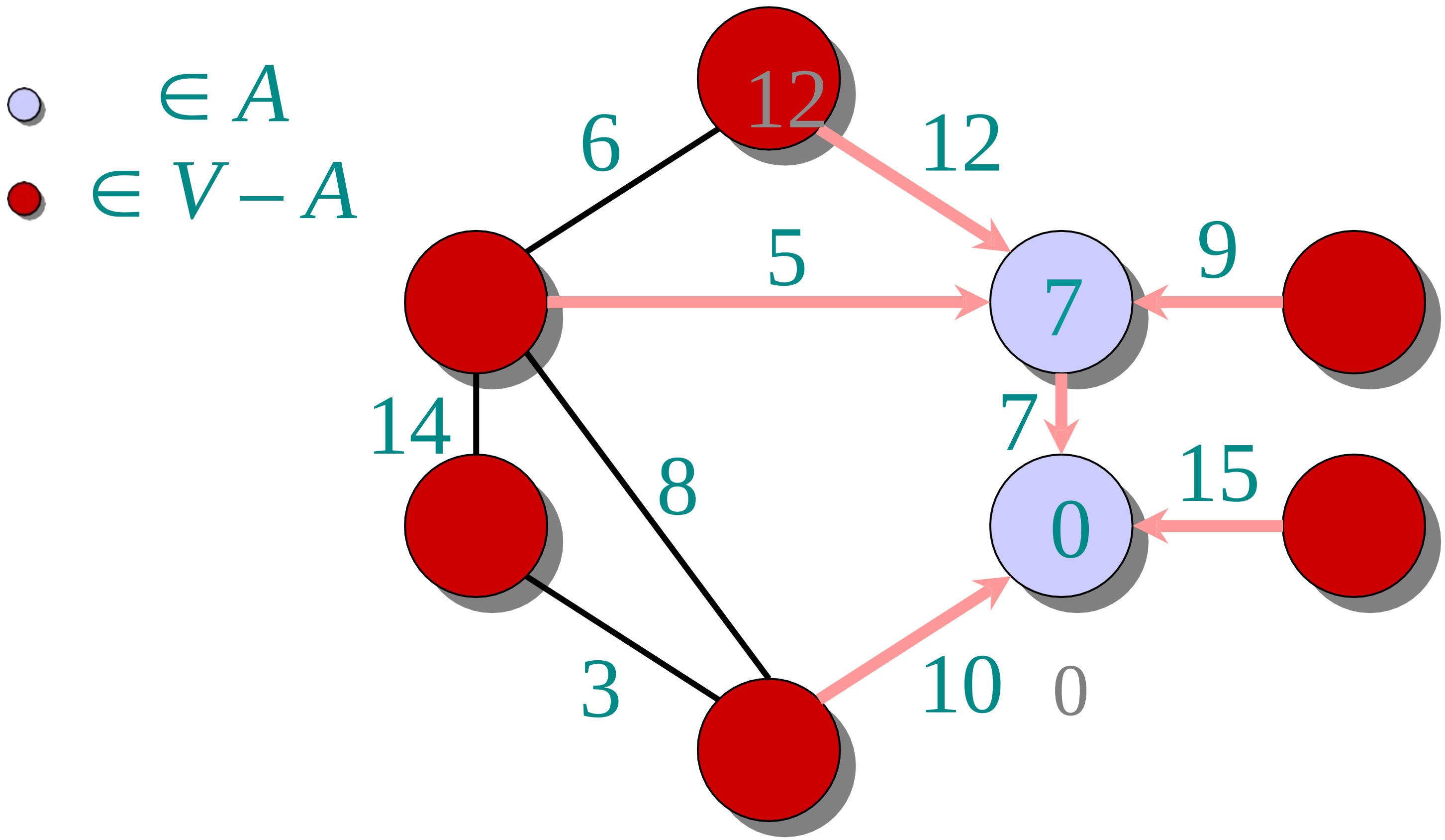
Example of Prim's algorithm



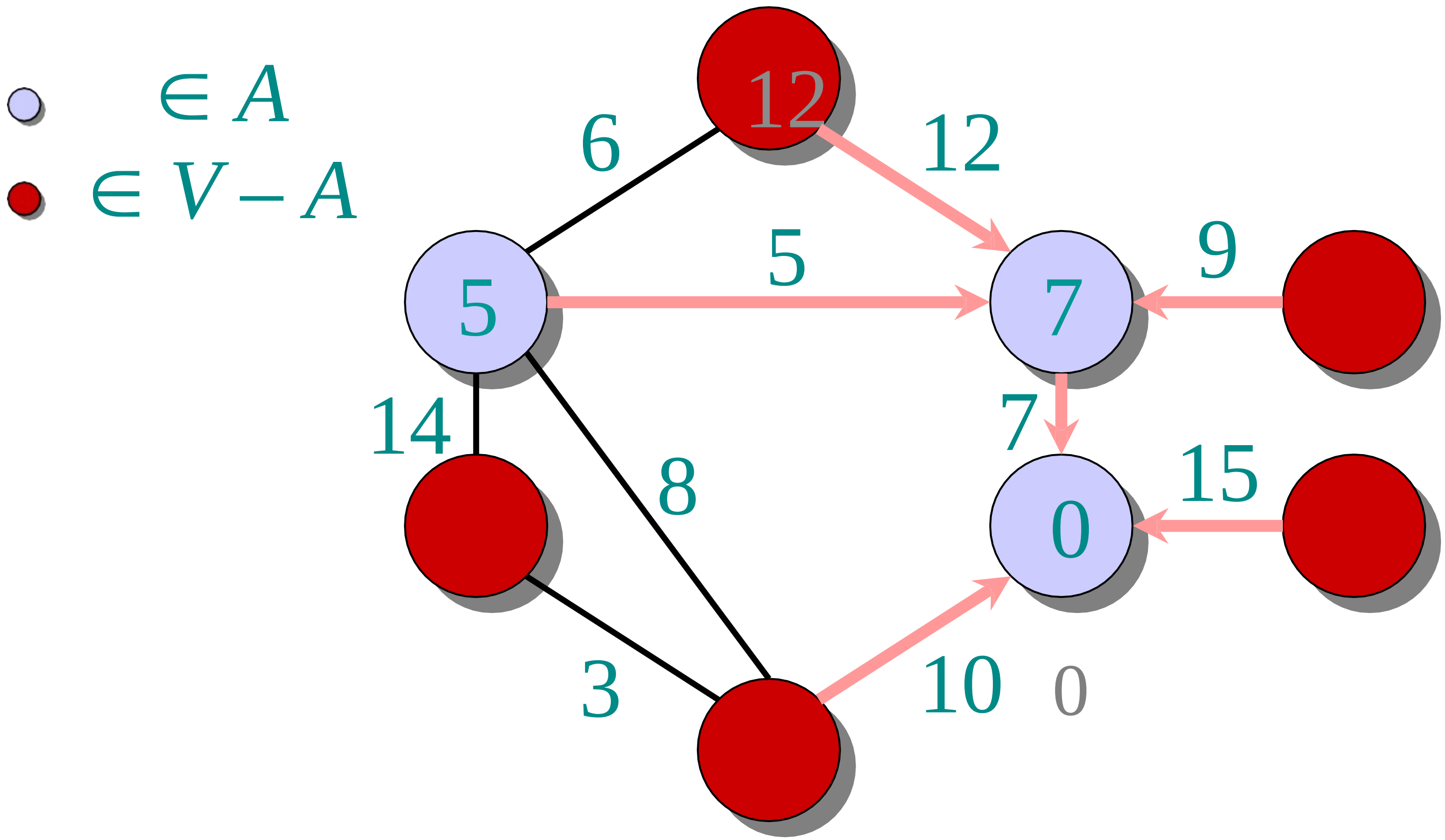
Example of Prim's algorithm



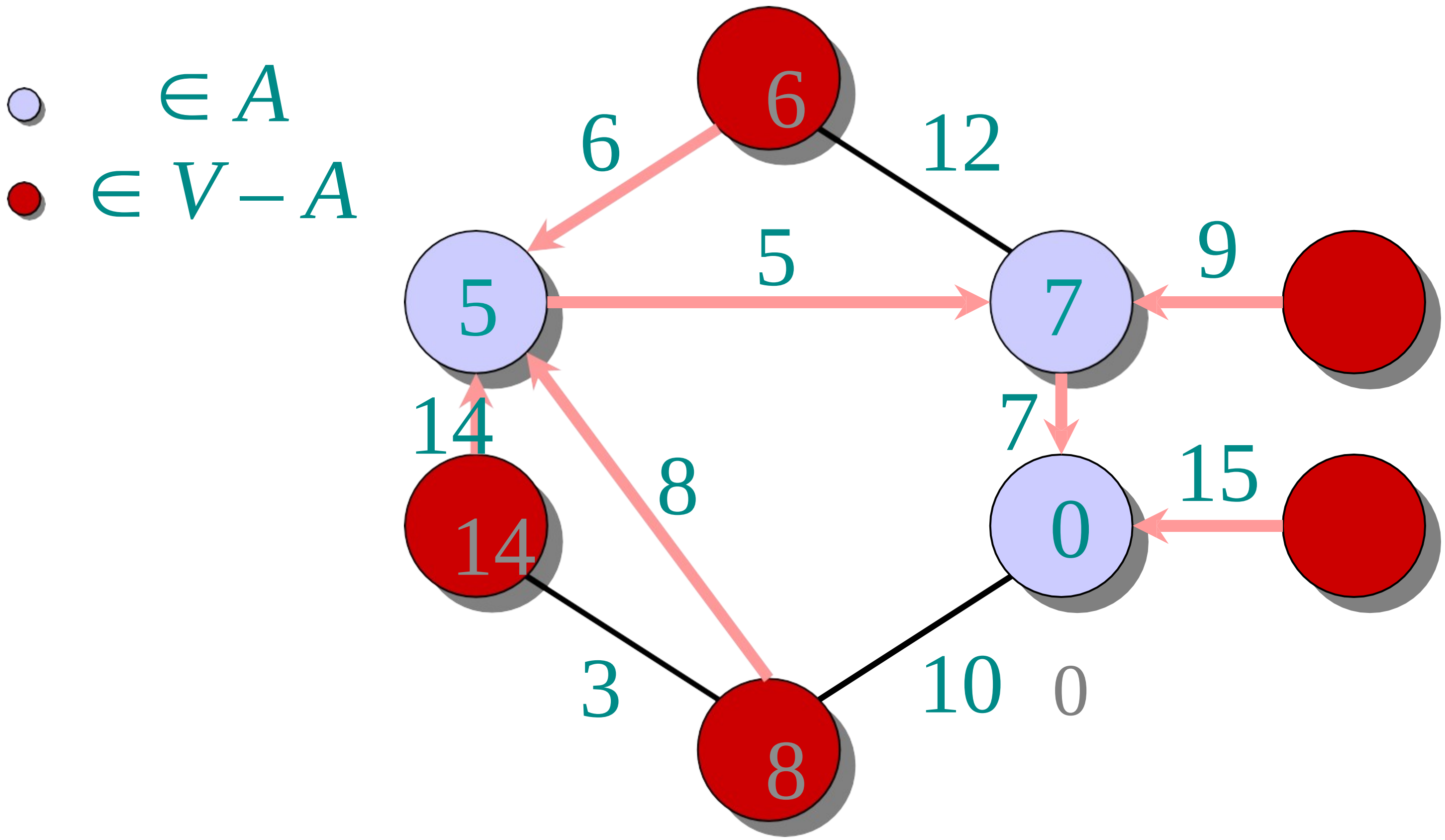
Example of Prim's algorithm



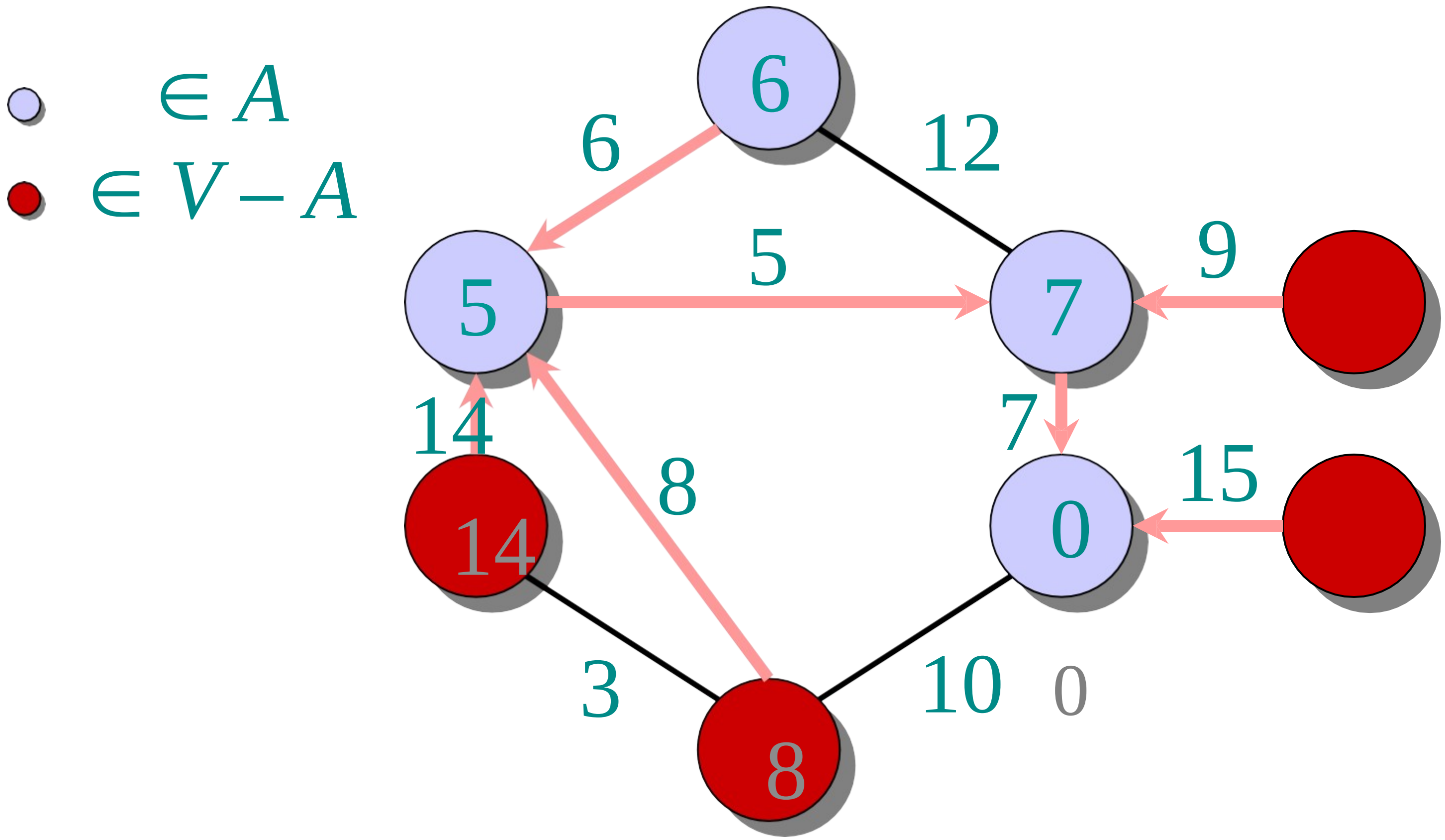
Example of Prim's algorithm



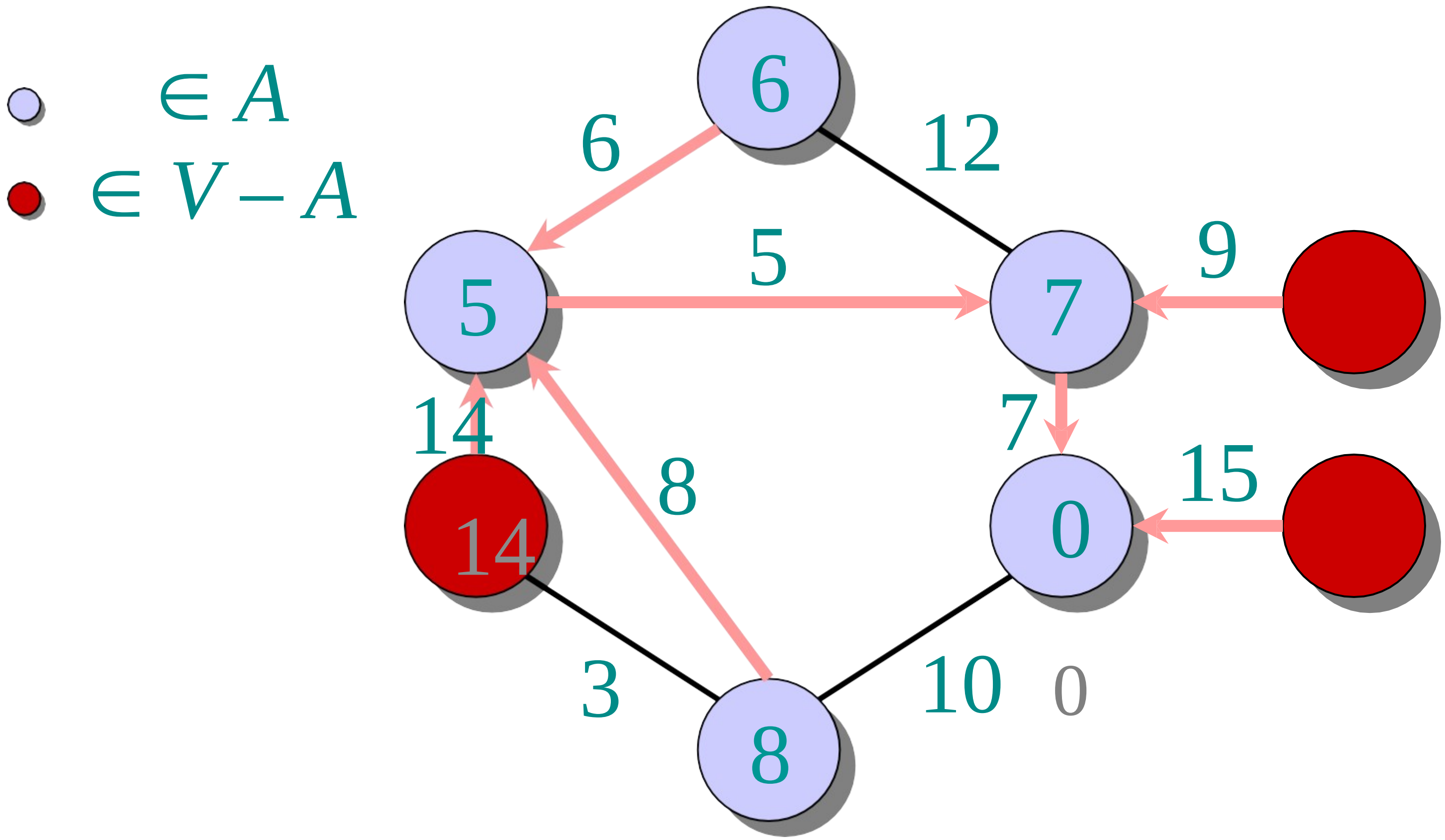
Example of Prim's algorithm



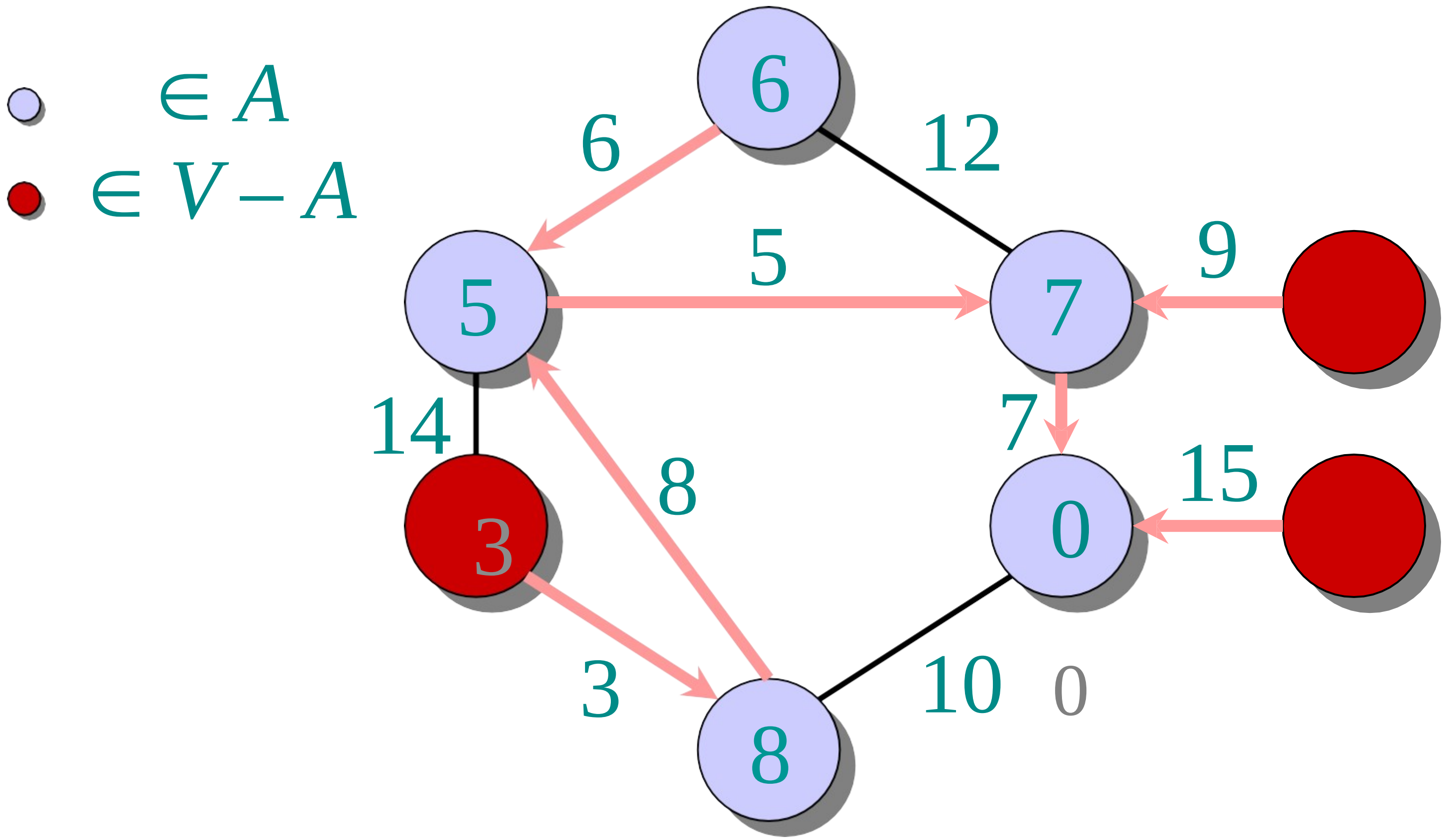
Example of Prim's algorithm



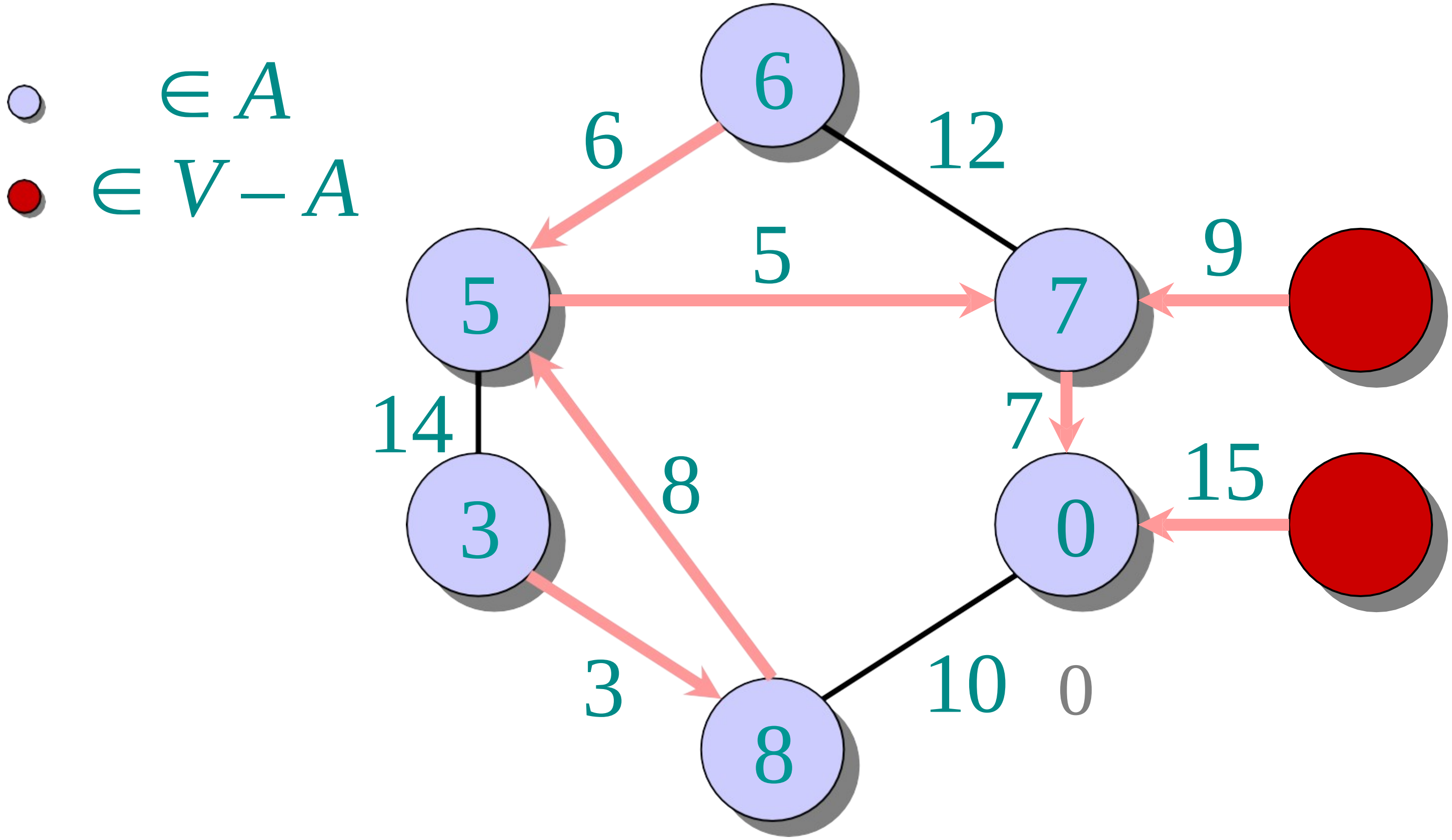
Example of Prim's algorithm



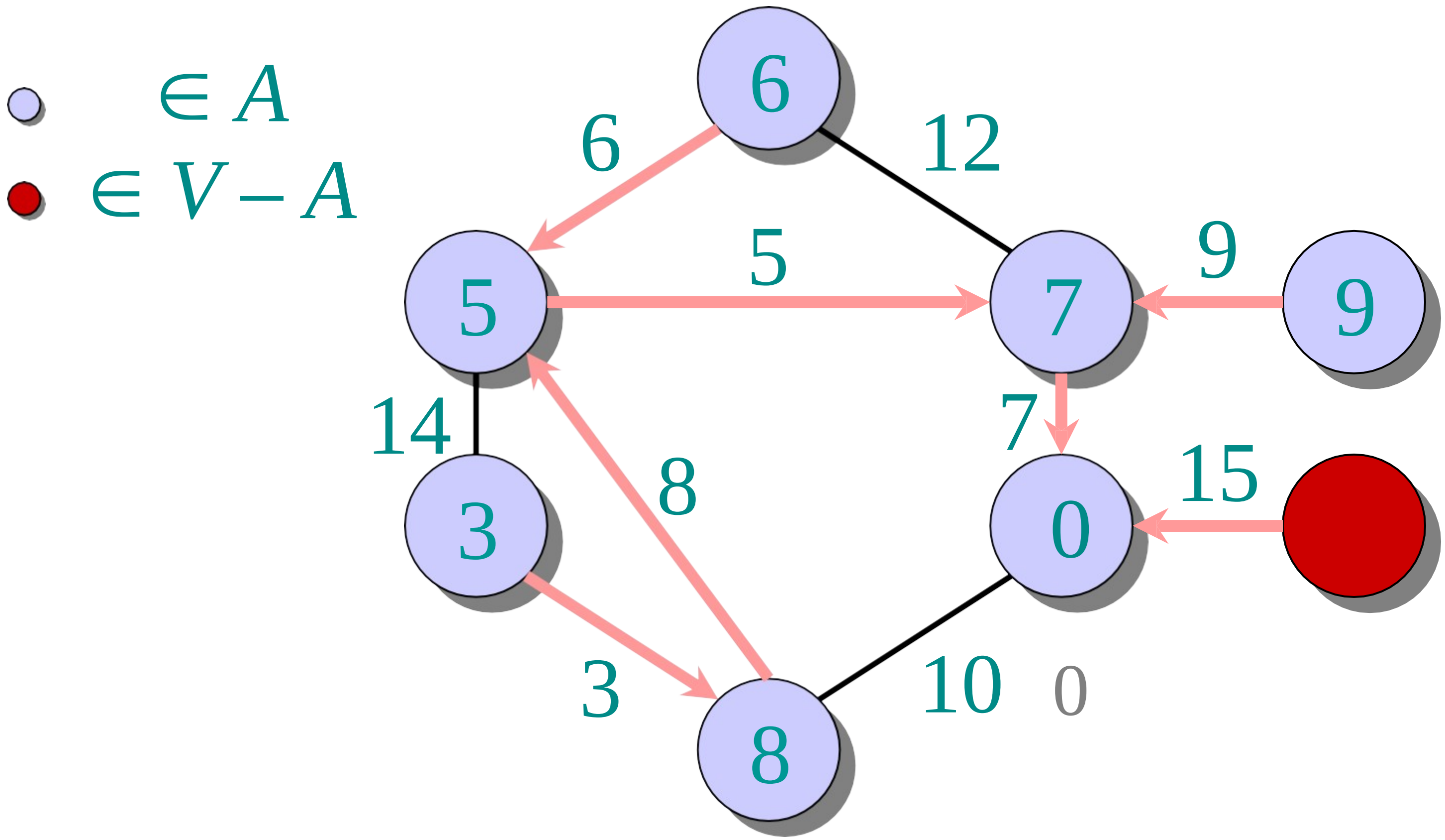
Example of Prim's algorithm



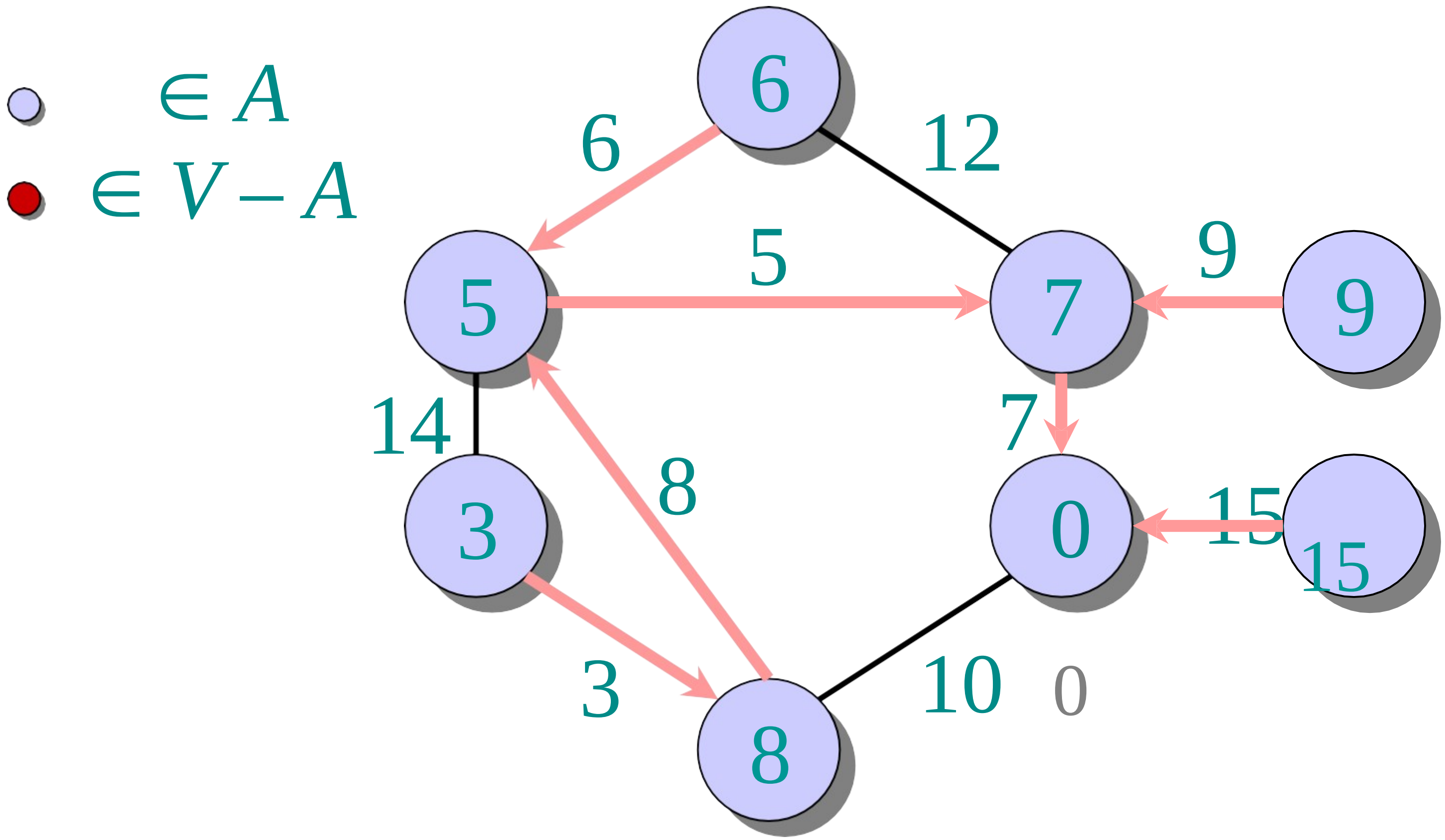
Example of Prim's algorithm



Example of Prim's algorithm



Example of Prim's algorithm



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

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

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

total $\Theta(V)$ $\begin{cases} Q \leftarrow V \\ key[v] \leftarrow \infty \text{ for all } v \in V \\ key[s] \leftarrow 0 \text{ for some arbitrary } s \in V \end{cases}$

while $Q \neq \emptyset$

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

total $\Theta(V)$ $\begin{cases} Q \leftarrow V \\ key[v] \leftarrow \infty \text{ for all } v \in V \\ key[s] \leftarrow 0 \text{ for some arbitrary } s \in V \end{cases}$

while $Q \neq \emptyset$

$|V|$ times

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

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

while $Q \neq \emptyset$

$|V|$ times

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

Kruskal’s Algorithm

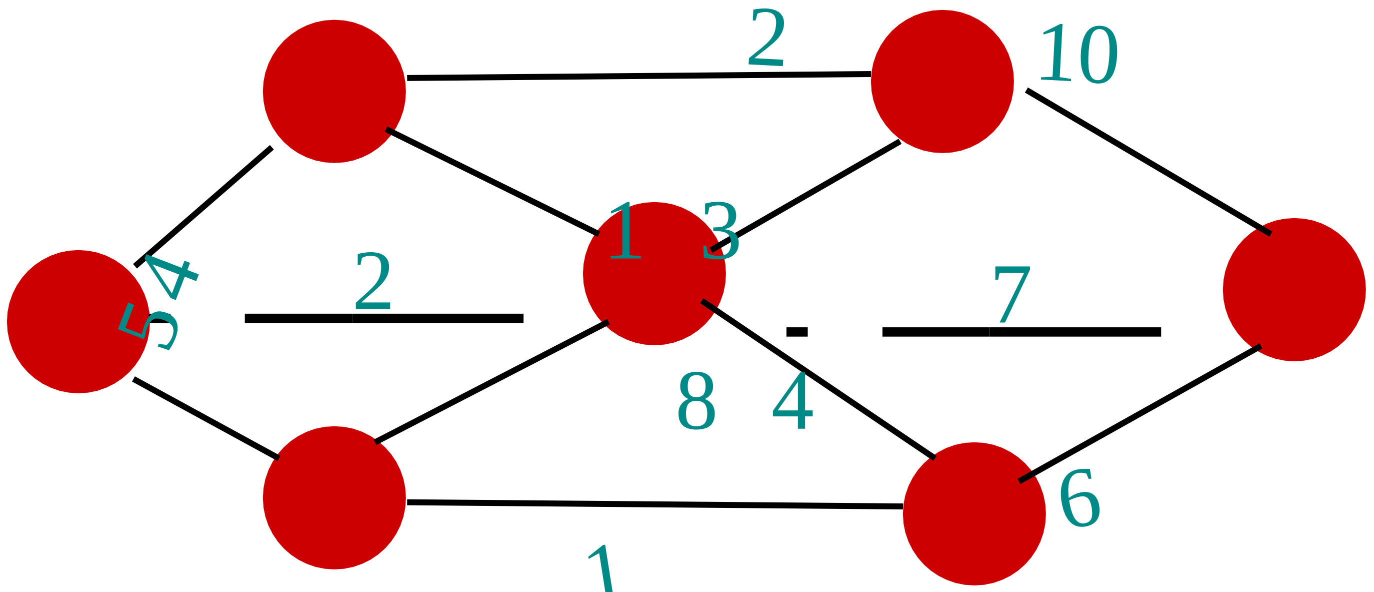
- **Idea: - Forming Minimum Spanning Tree**
- **Steps**

1. Include all the Vertices

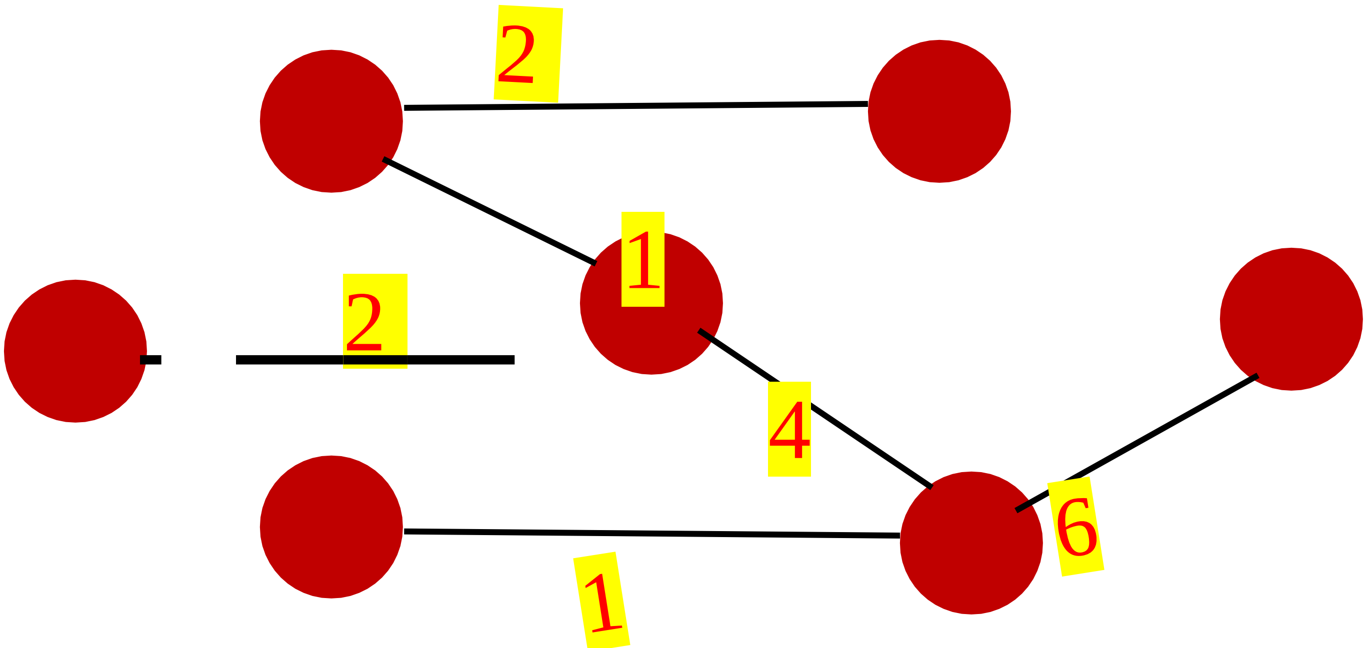
2. Select minimu Weighted Edges and Use those edges as long as they are not forming a Cycle.

3. Repeat Step 2 as till all the vertices are visited.

Example of Kruskal's Algorithm



Example of Kruskal's Algorithm



MST algorithms

Kruskal's algorithm

- Running time $= O(E \lg V)$.

Best to date:

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

Chapter -7 (Reading)
Limitations of Algorithm Power

- Information theoretic arguments
  - P vs NP
  - NP-hard and NP-Complete Problems
- Problem reduction

Example

Suppose we know that if one could travel faster than the speed of light, then one could travel back in time.

Using our language, the problem of traveling back to the past reduces to the problem of traveling faster than the speed of light

If we manage to build a faster-than-light vehicle, then we can go back to the past

But if we prove thatis impossible to travel back in time, then we immediately know it is impossible to build a faster-than-light vehicle.

Assignment II

1. Explain Optimal Binary Search Tree
2. Show the algorithm with example
3. Write the pseudocode
4. Perform the analysis