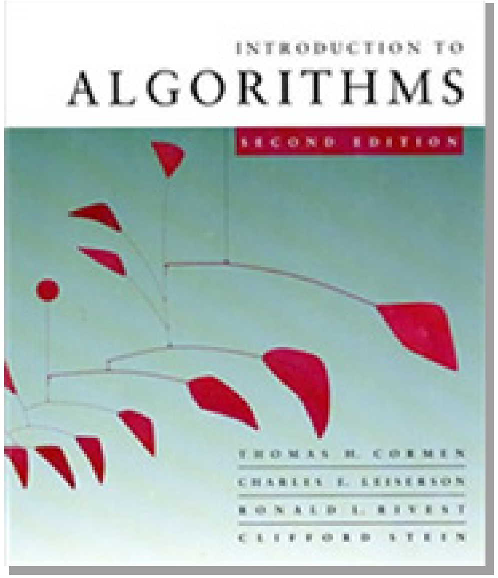
Introduction to Algorithms

6.046J/18.401J

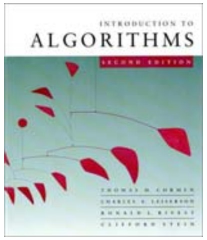


LECTURE 16

Greedy Algorithms (and Graphs)

- Graph representation
- Minimum spanning trees
- Optimal substructure
- Greedy choice
- Prim's greedy MST algorithm

Prof. Charles E. Leiserson



Graphs (review)

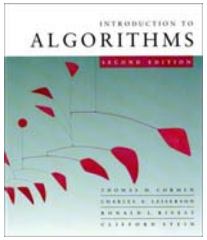
Definition. A ***directed graph (digraph)*** $G = (V, E)$ is an ordered pair consisting of

- 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.

In either case, we have $|E| = O(V^2)$. Moreover, if G is connected, then $|E| \geq |V| - 1$, which implies that $\lg |E| = \Theta(\lg V)$.

(Review CLRS, Appendix B.)



Adjacency-matrix representation

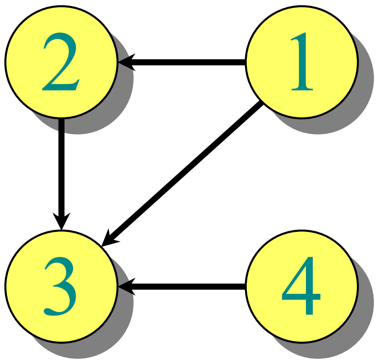
The ***adjacency matrix*** of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, is the matrix $A[1 \,..\, n, 1 \,..\, n]$ given by

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

Adjacency-matrix representation

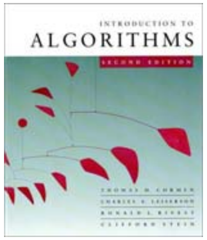
The ***adjacency matrix*** of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, is the matrix $A[1 \,.\,.\, n, 1 \,.\,.\, n]$ given by

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



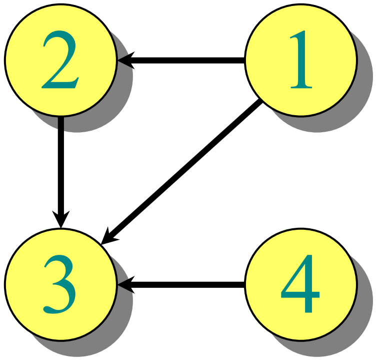
$$\begin{array}{c|cccc} A & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ 2 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 1 & 0 \end{array}$$

$\Theta(V^2)$ storage $\Rightarrow$ ***dense*** representation.

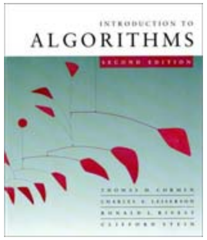


Adjacency-list representation

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

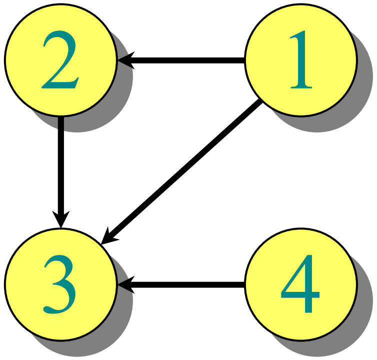


$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$



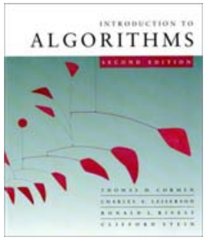
Adjacency-list representation

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



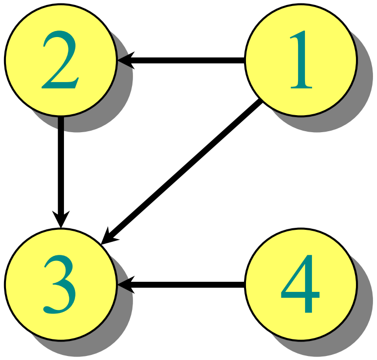
$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

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



Adjacency-list representation

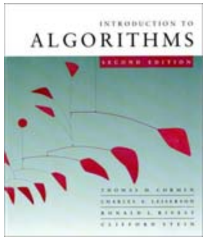
An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.



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

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

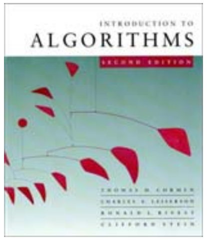
Handshaking Lemma: $\sum_{v \in V} = 2|E|$ for undirected graphs $\Rightarrow$ adjacency lists use $\Theta(V + E)$ storage — a ***sparse*** representation (for either type of graph).



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.)



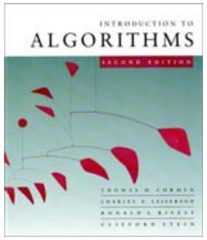
Minimum spanning trees

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

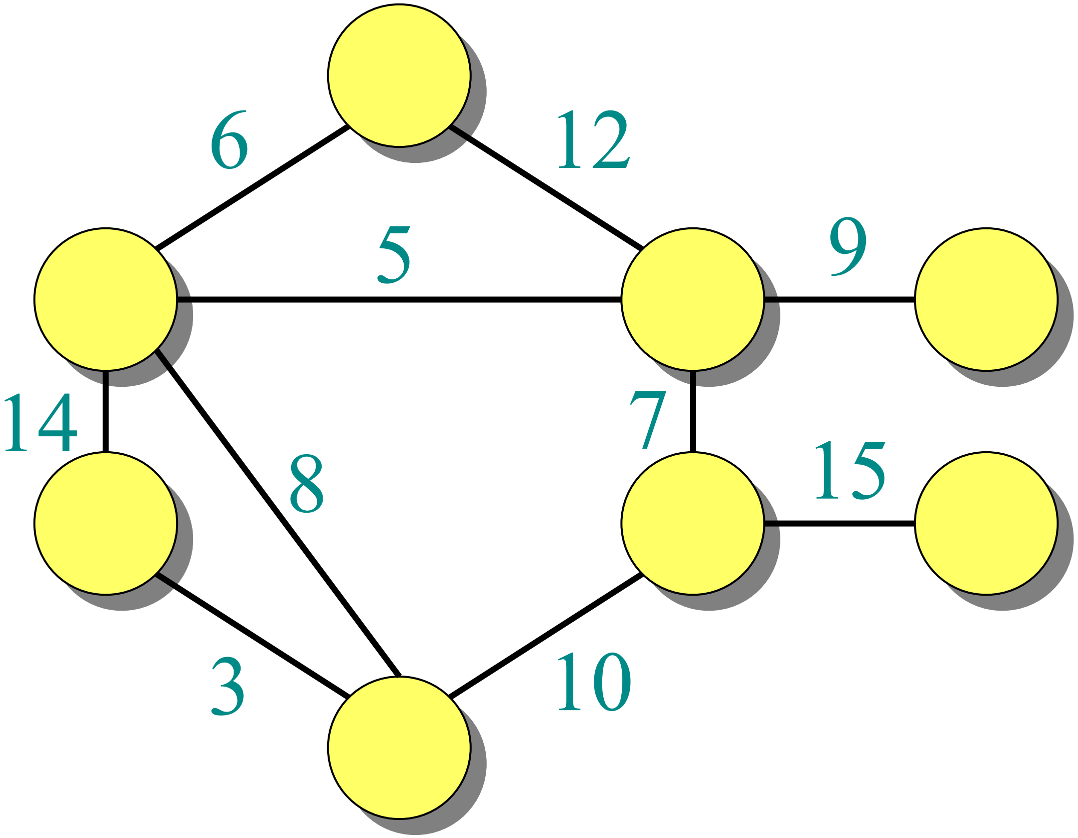
- For simplicity, assume that all edge weights are distinct. (CLRS covers the general case.)

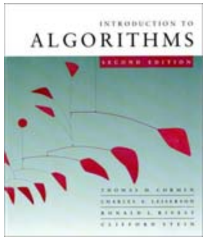
Output: A ***spanning tree*** T — a tree that connects all vertices — of minimum weight:

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

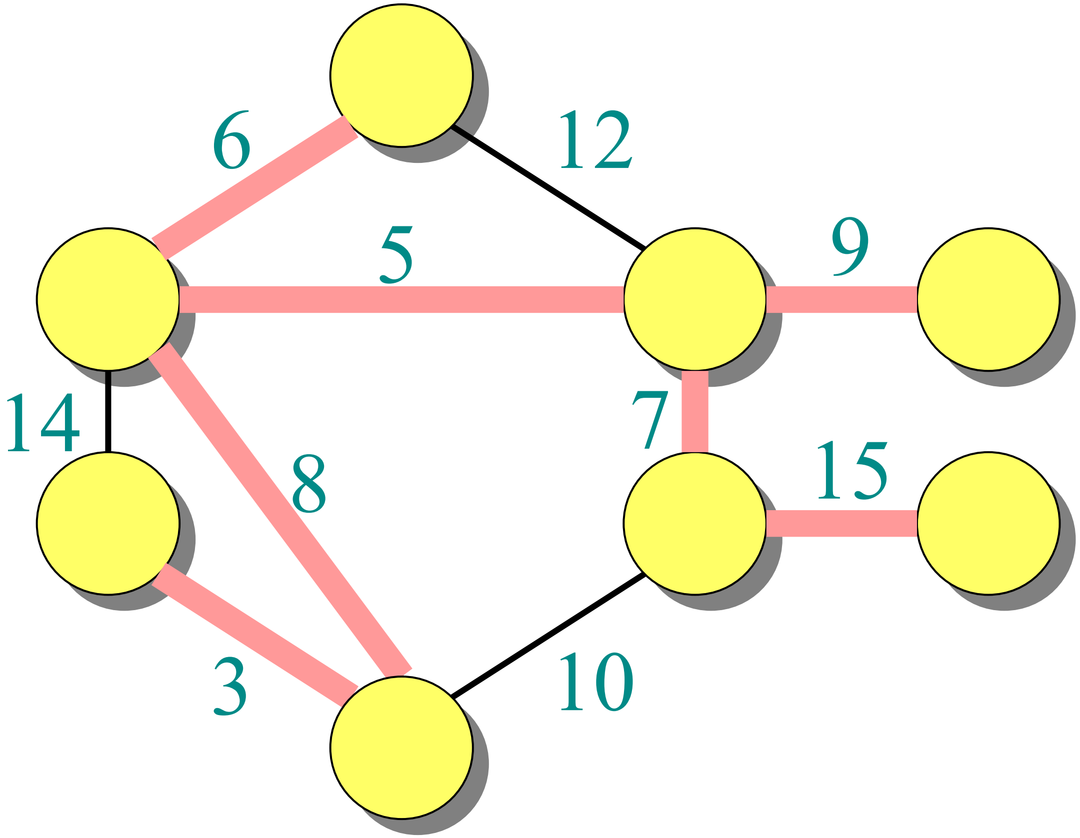


Example of MST





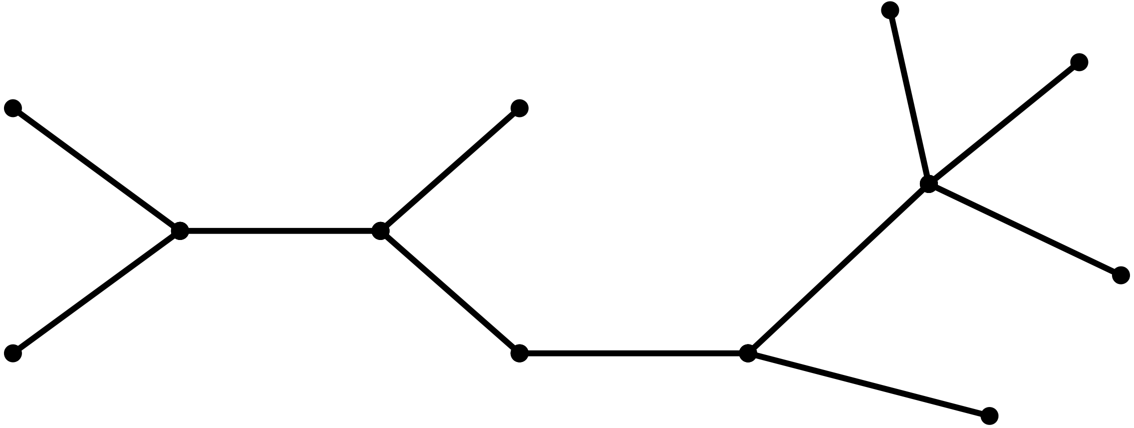
Example of MST



Optimal substructure

MST T:

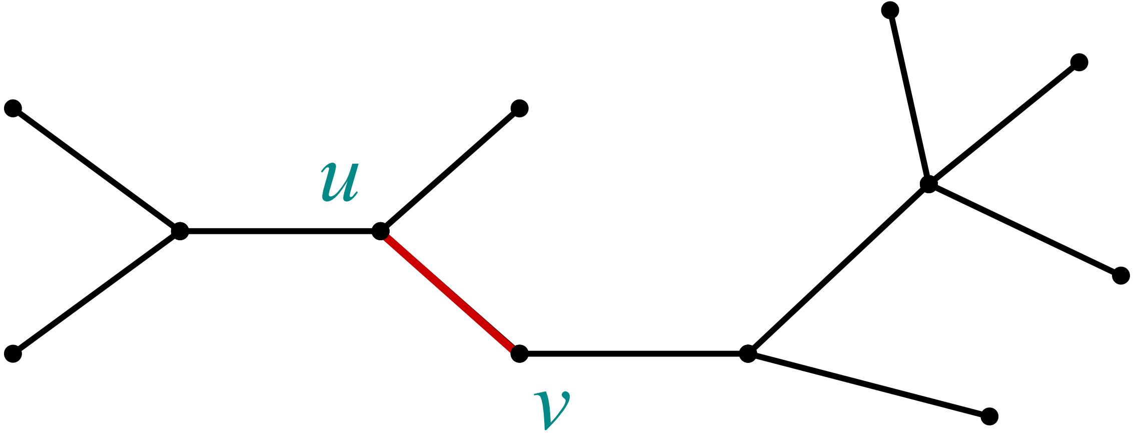
(Other edges of G are not shown.)



Optimal substructure

MST T:

(Other edges of G are not shown.)

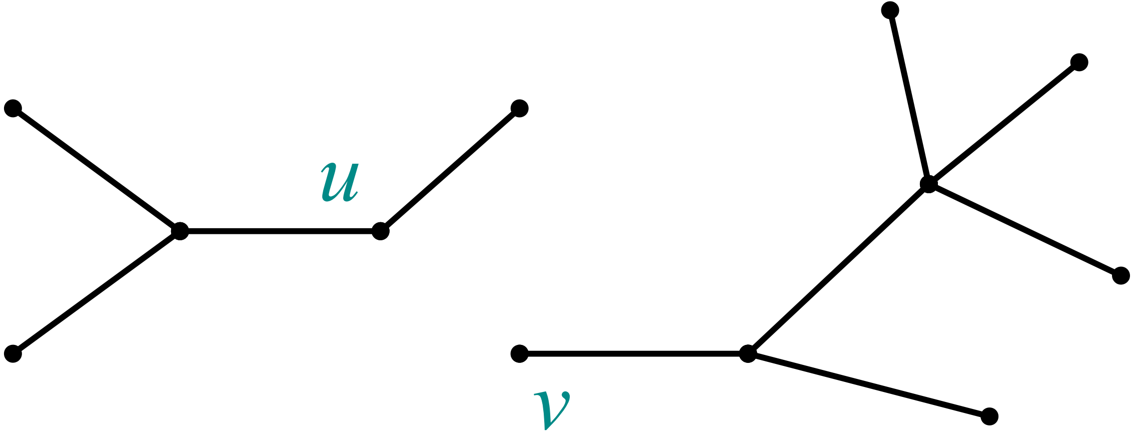


Remove any edge $(u, v) \in T$.

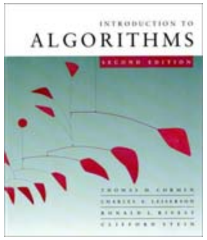
Optimal substructure

MST T:

(Other edges of G are not shown.)



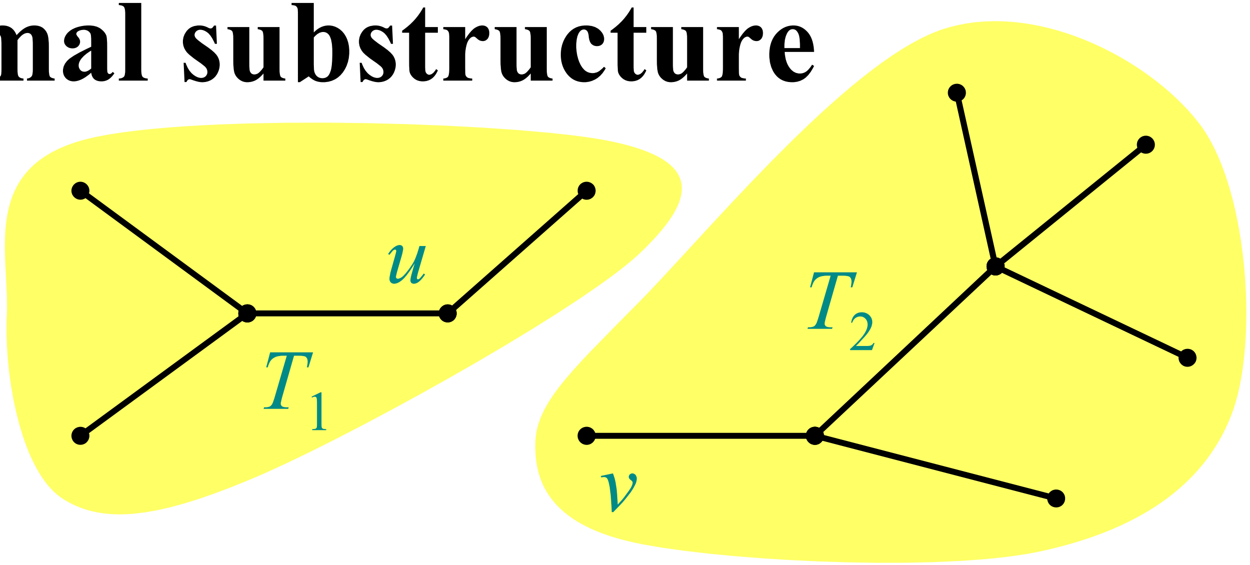
Remove any edge $(u, v) \in T$.



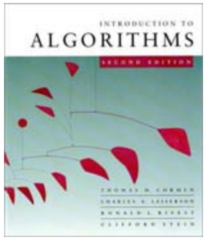
Optimal substructure

MST T:

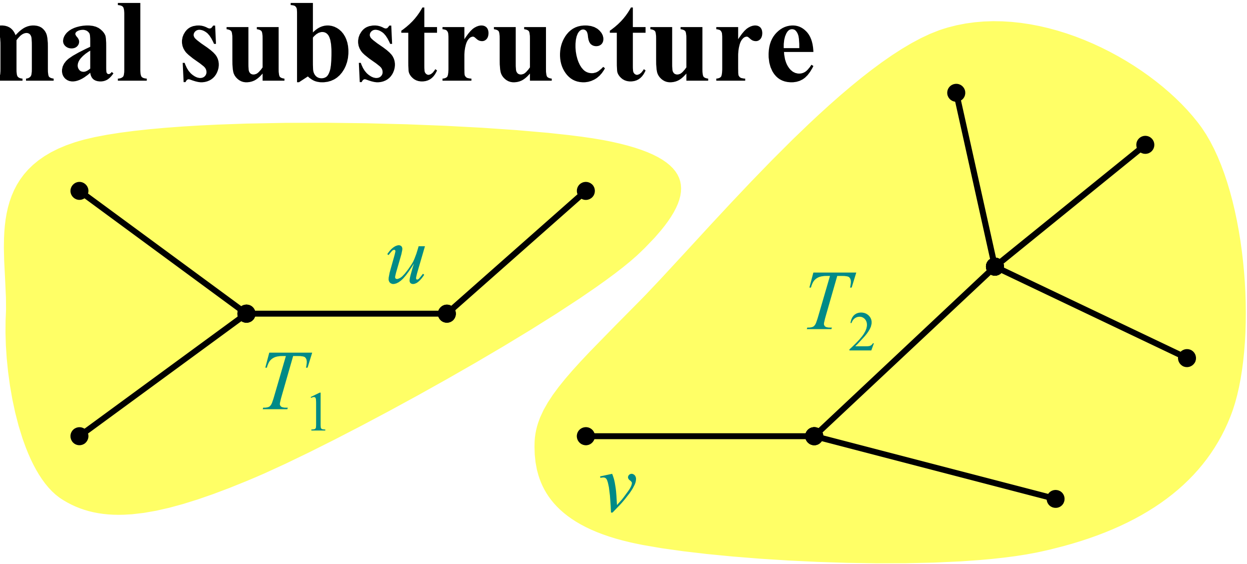
(Other edges of G are not shown.)



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



Optimal substructure



MST T:

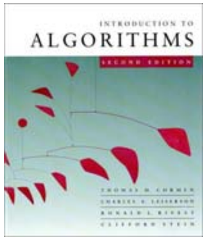
(Other edges of G are not shown.)

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.

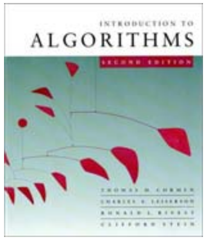


Proof of optimal substructure

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

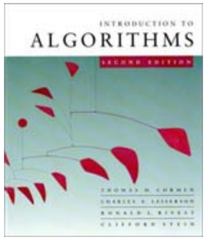
Proof. Cut and paste:

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

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:

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

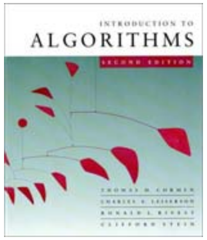
If T_1' were a lower-weight spanning tree than T_1 for G_1, then $T' = \{(u, v)\} \cup T_1' \cup T_2$ would be a lower-weight spanning tree than T for G. □

Do we also have overlapping subproblems?

- Yes.

Great, then dynamic programming may work!

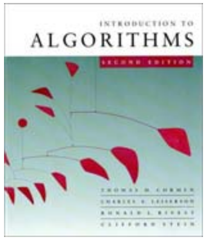
- Yes, but MST exhibits another powerful property which leads to an even more efficient algorithm.



Hallmark for “greedy” algorithms

Greedy-choice property

A locally optimal choice is globally optimal.



Hallmark for "greedy" algorithms

Greedy-choice property

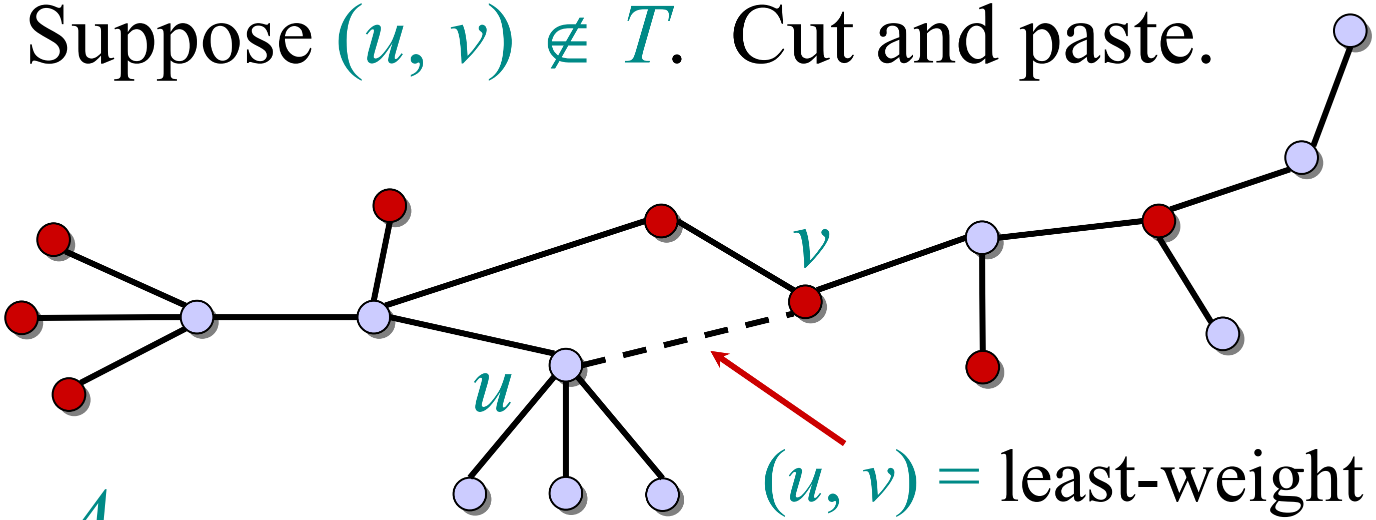
A locally optimal choice is globally optimal.

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.

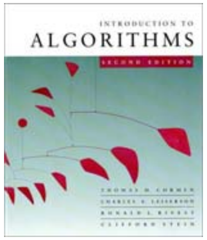
T:



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

○ $\in A$

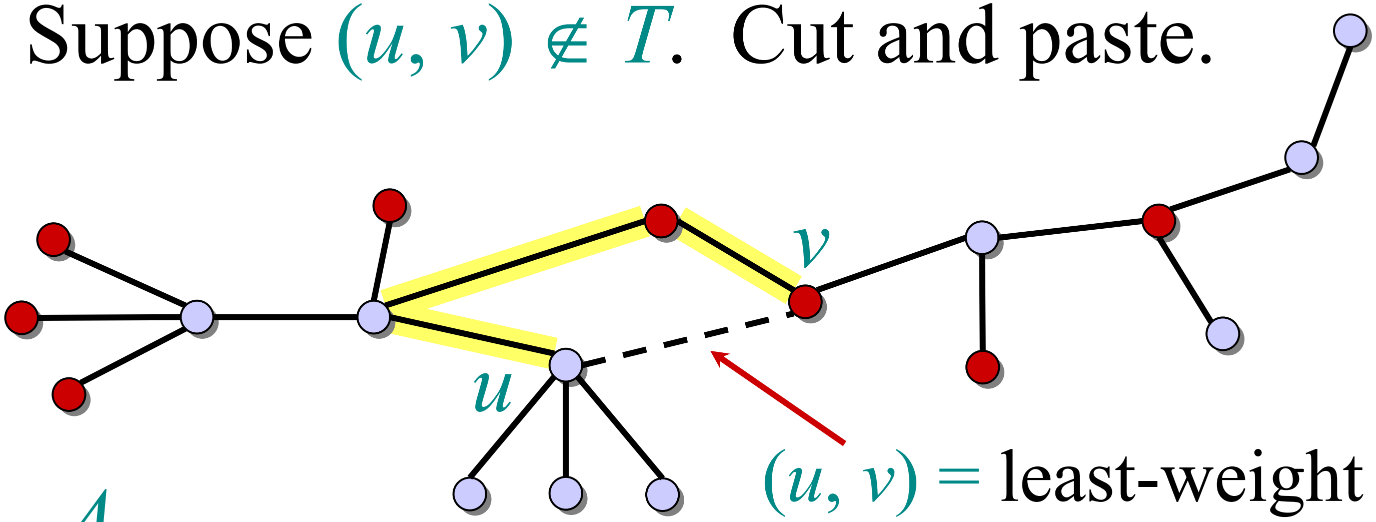
● $\in V - A$



Proof of theorem

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

T:

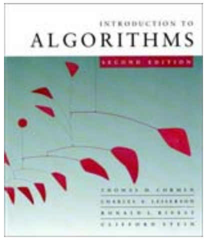


○ $\in A$

● $\in V - A$

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

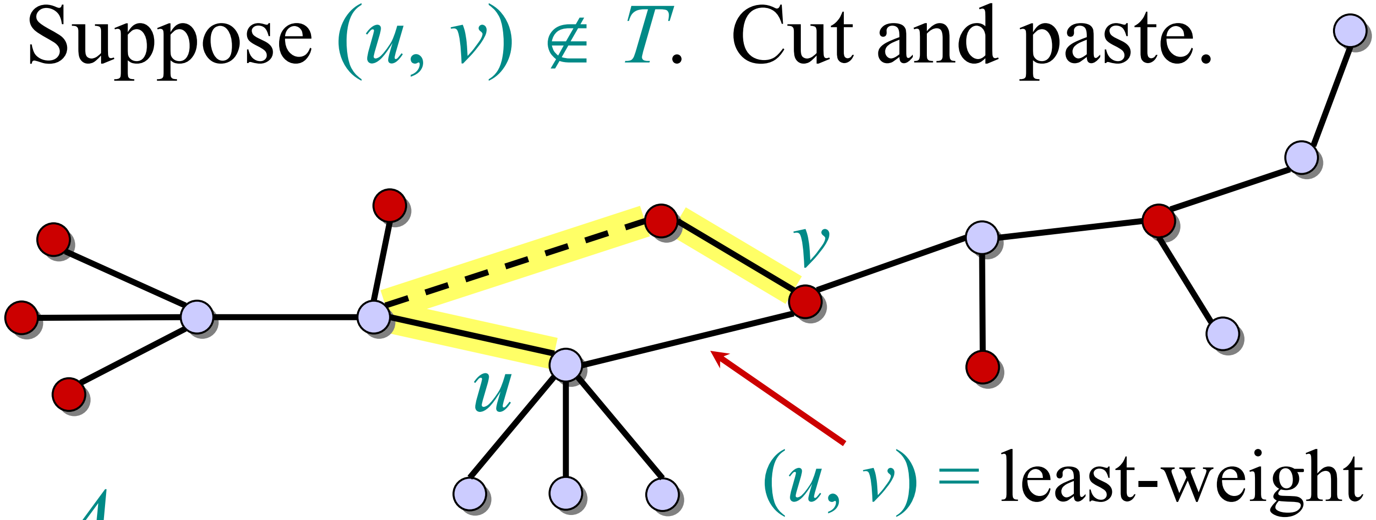
Consider the unique simple path from u to v in T.



Proof of theorem

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

T:



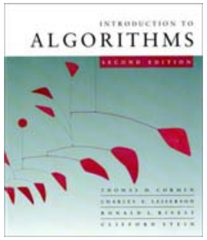
● $\in A$

● $\in V - A$

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

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

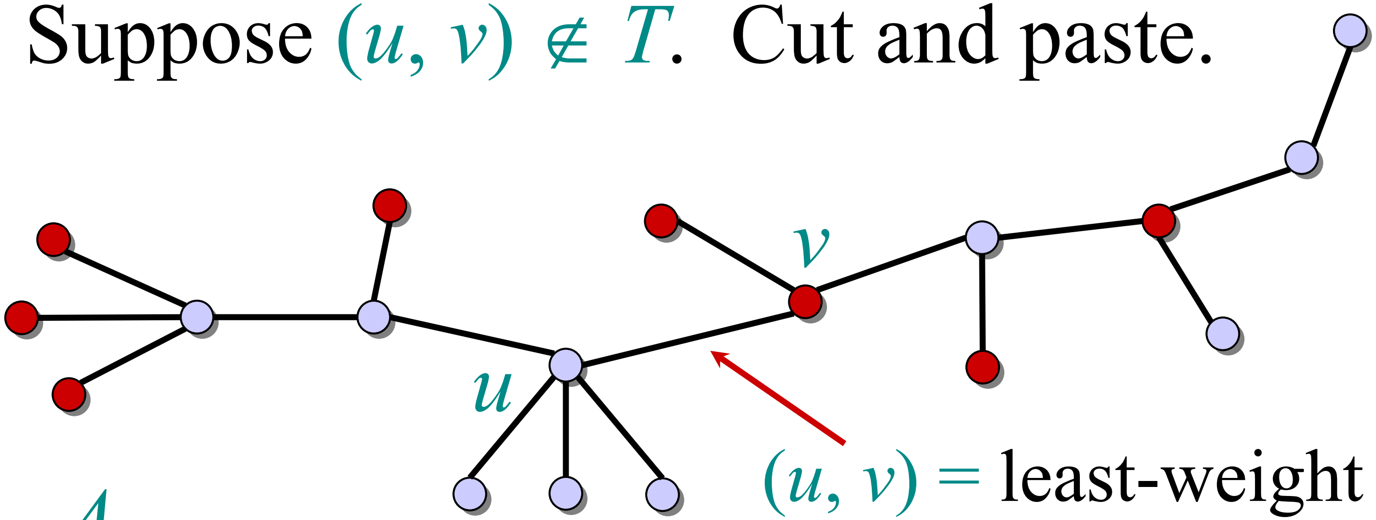
Swap (u, v) with the first edge on this path that connects a vertex in A to a vertex in $V - A$.



Proof of theorem

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

T':



○ $\in A$

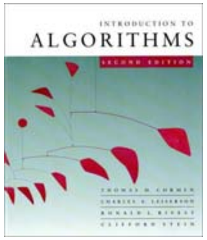
● $\in V - A$

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

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

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. ☐



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.

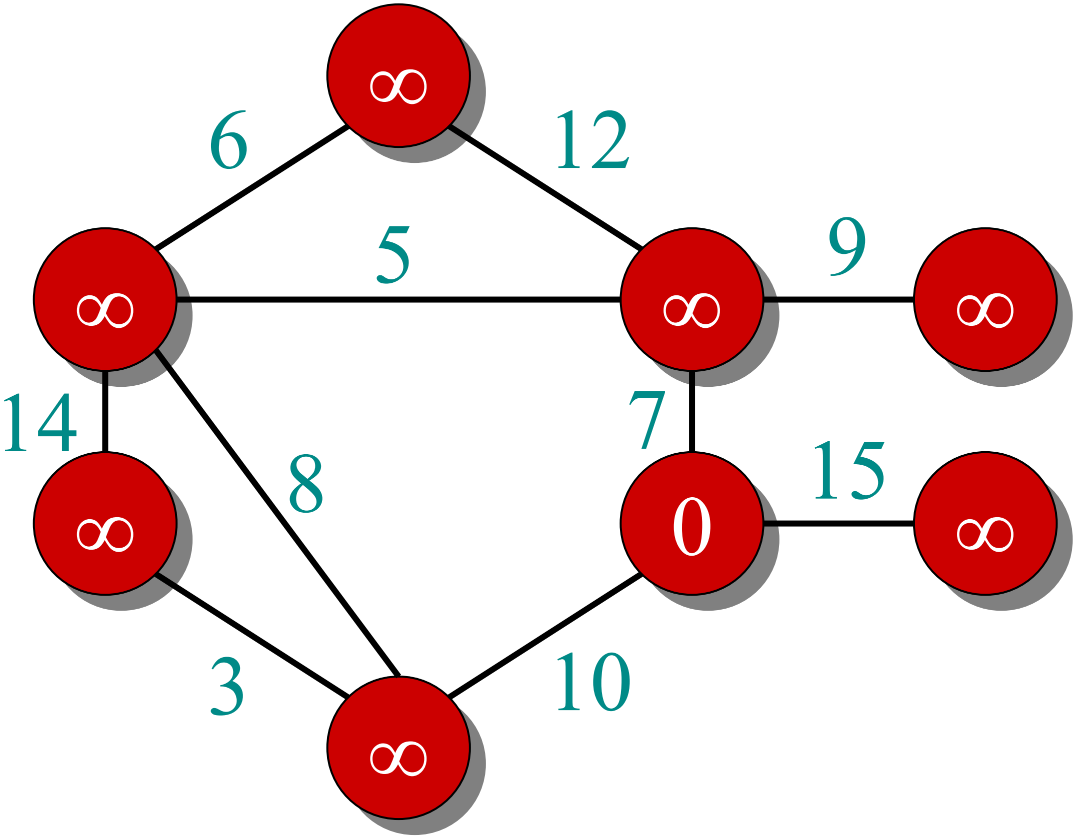
```
$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN$(Q)$
        for each $v \in Adj[u]$
            do if $v \in Q$ and $w(u, v) < key[v]$
                then $key[v] \leftarrow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

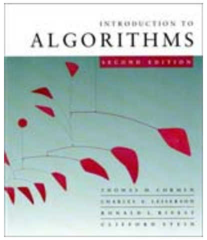
At the end, $\{(v, \pi[v])\}$ forms the MST.

Example of Prim's algorithm

○ $\in A$

● $\in V - A$

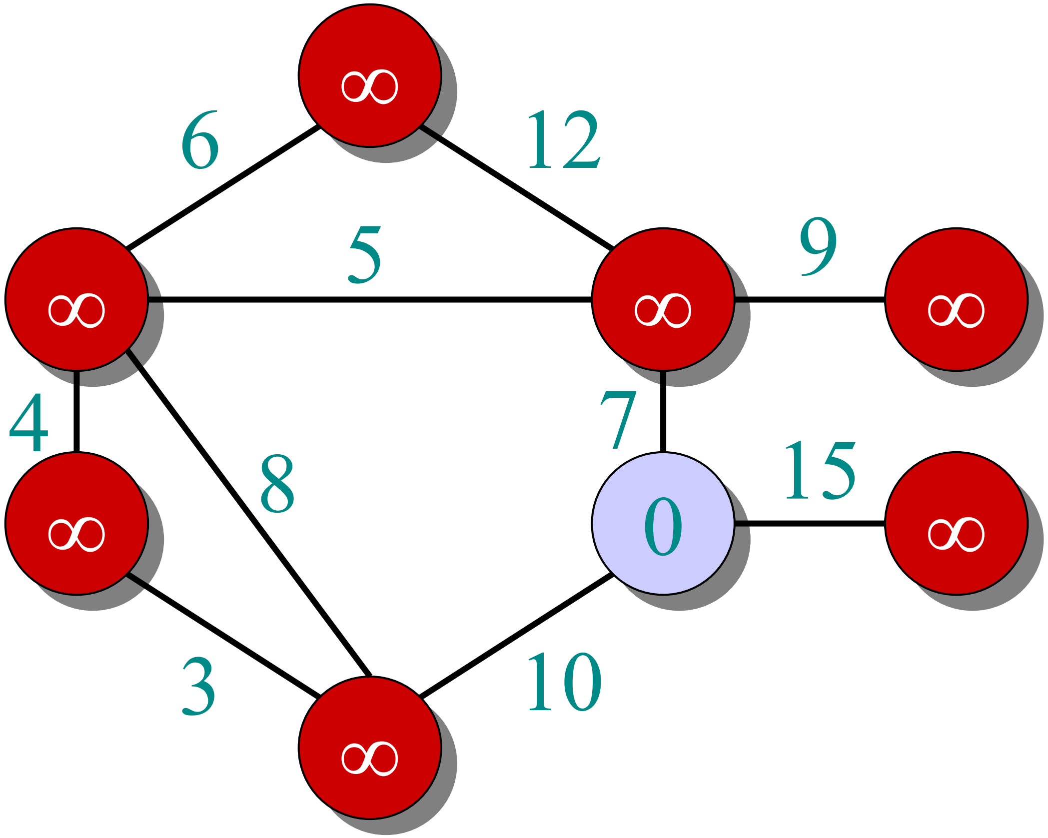




Example of Prim's algorithm

○ $\in A$

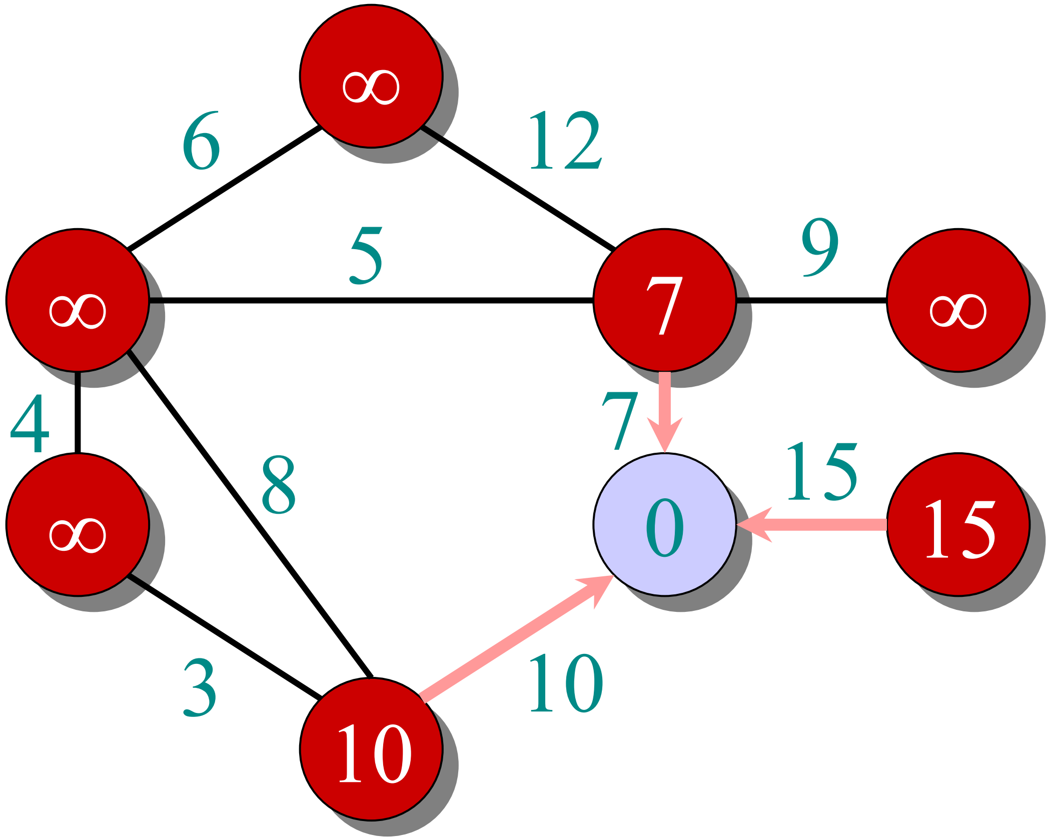
● $\in V - A$



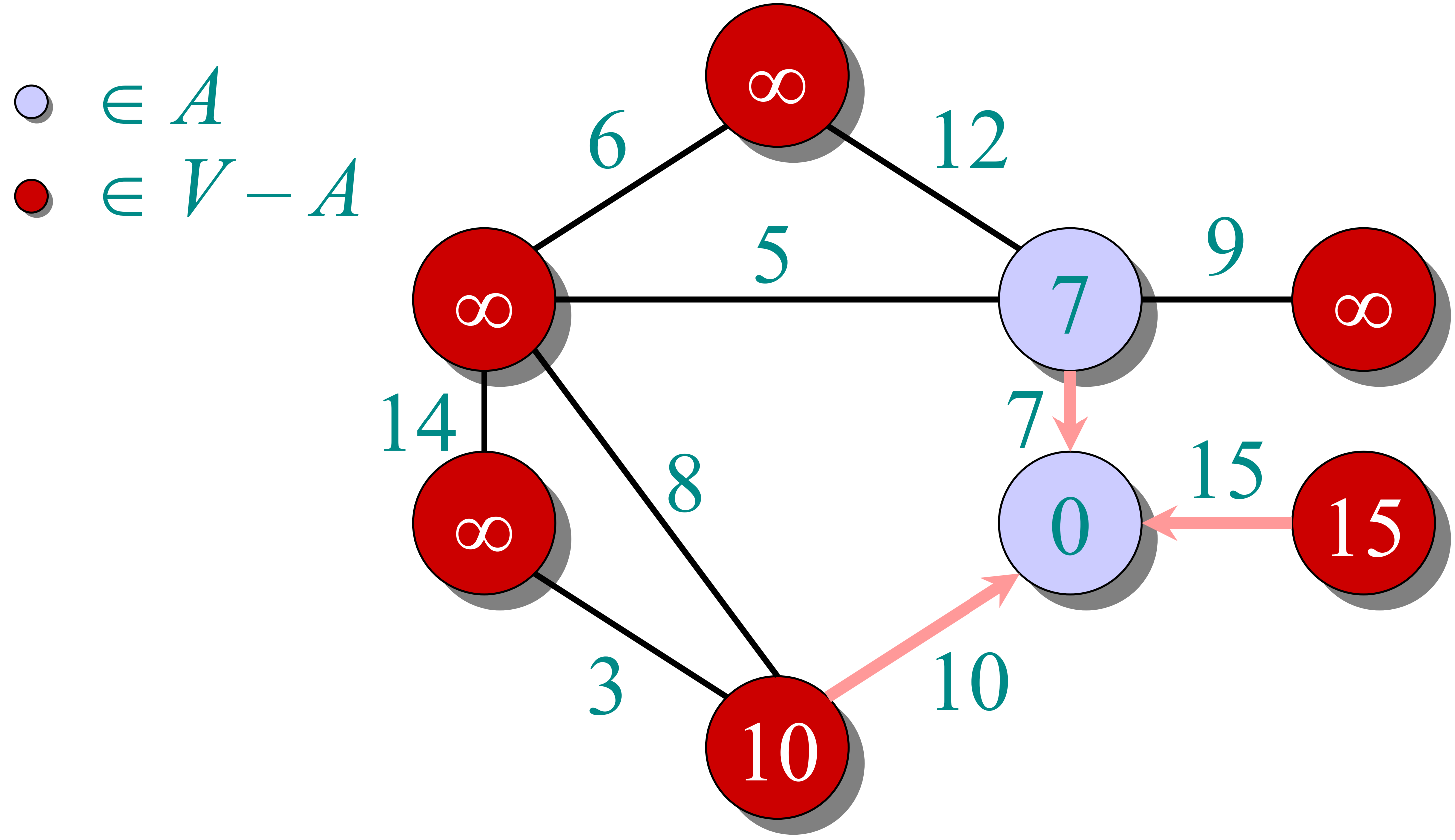
Example of Prim's algorithm

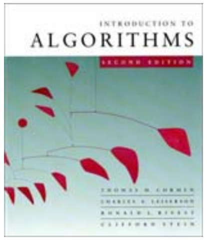
○ $\in A$

● $\in V - A$



Example of Prim's algorithm

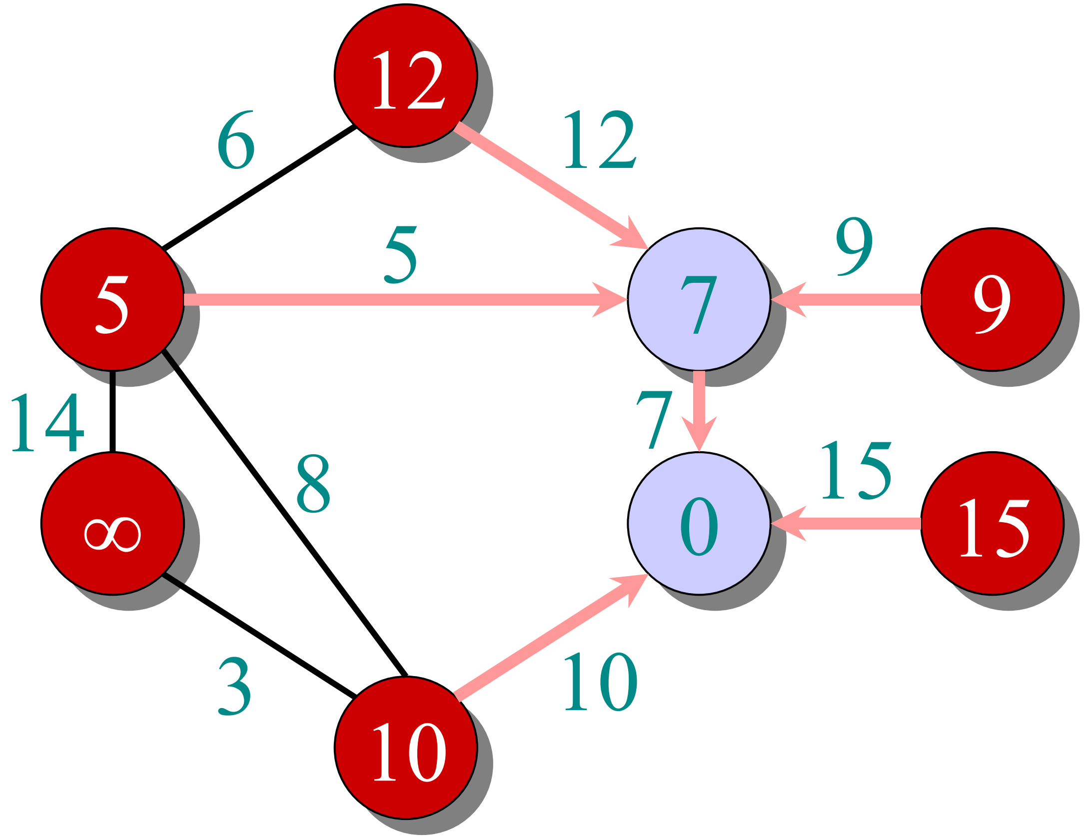


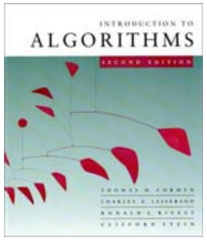


Example of Prim’s algorithm

○ $\in A$

● $\in V - A$

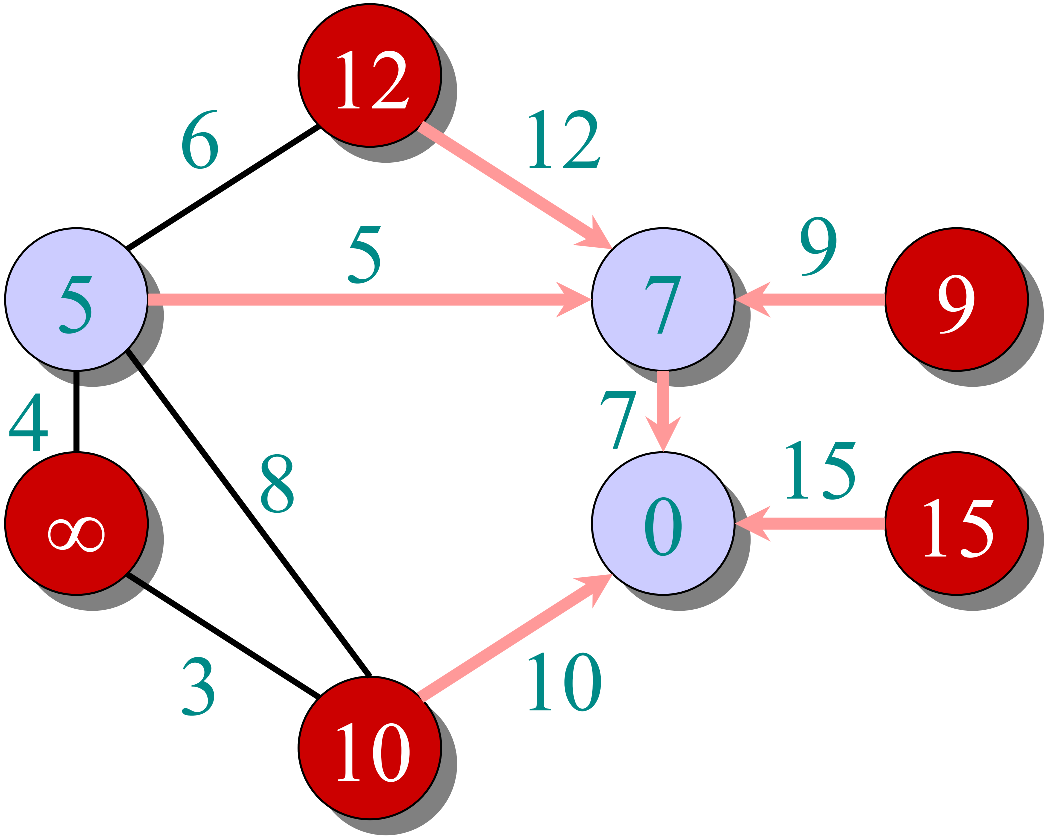


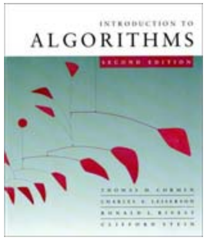


Example of Prim’s algorithm

○ $\in A$

● $\in V - A$

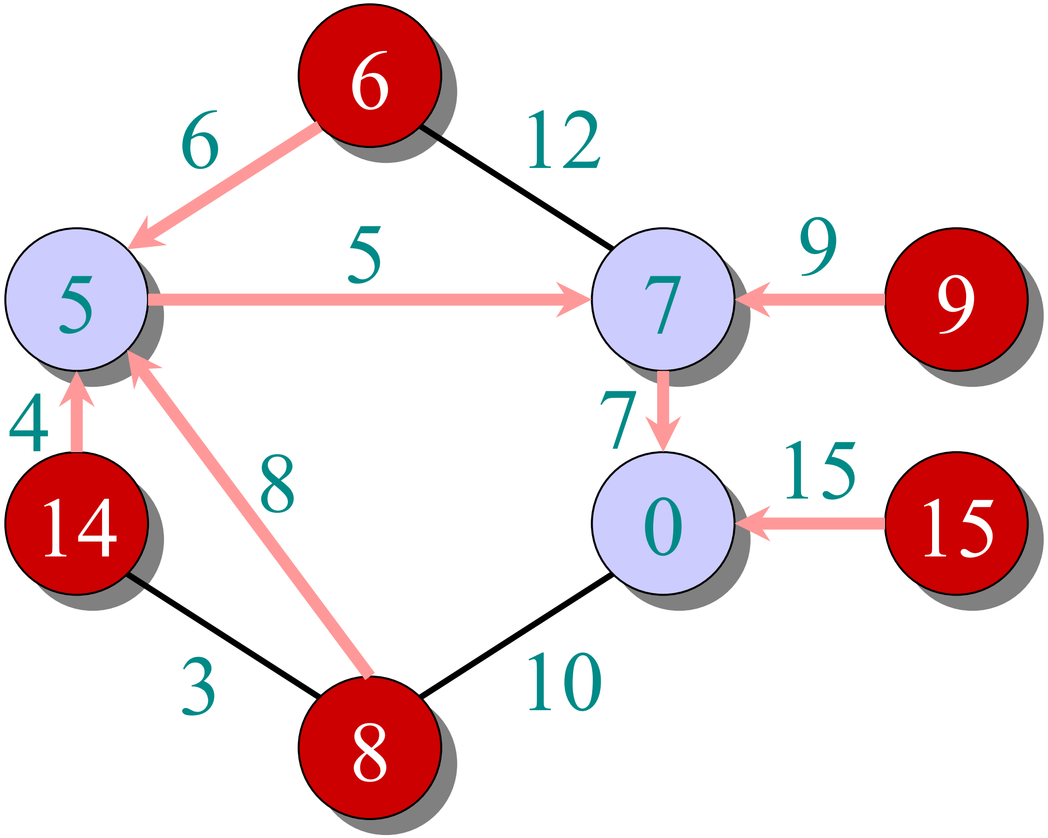


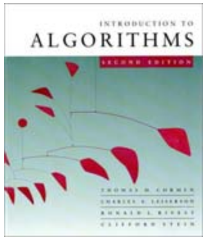


Example of Prim’s algorithm

○ $\in A$

● $\in V - A$

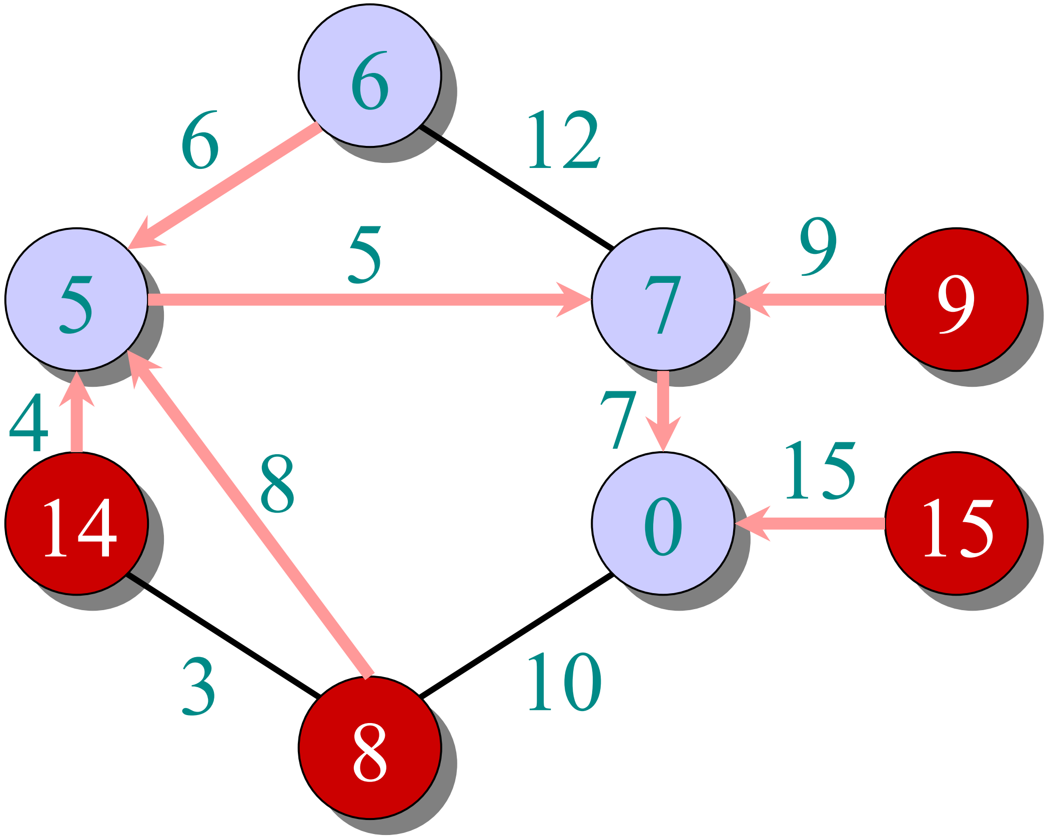


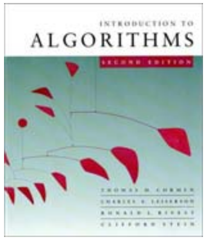


Example of Prim’s algorithm

○ $\in A$

● $\in V - A$

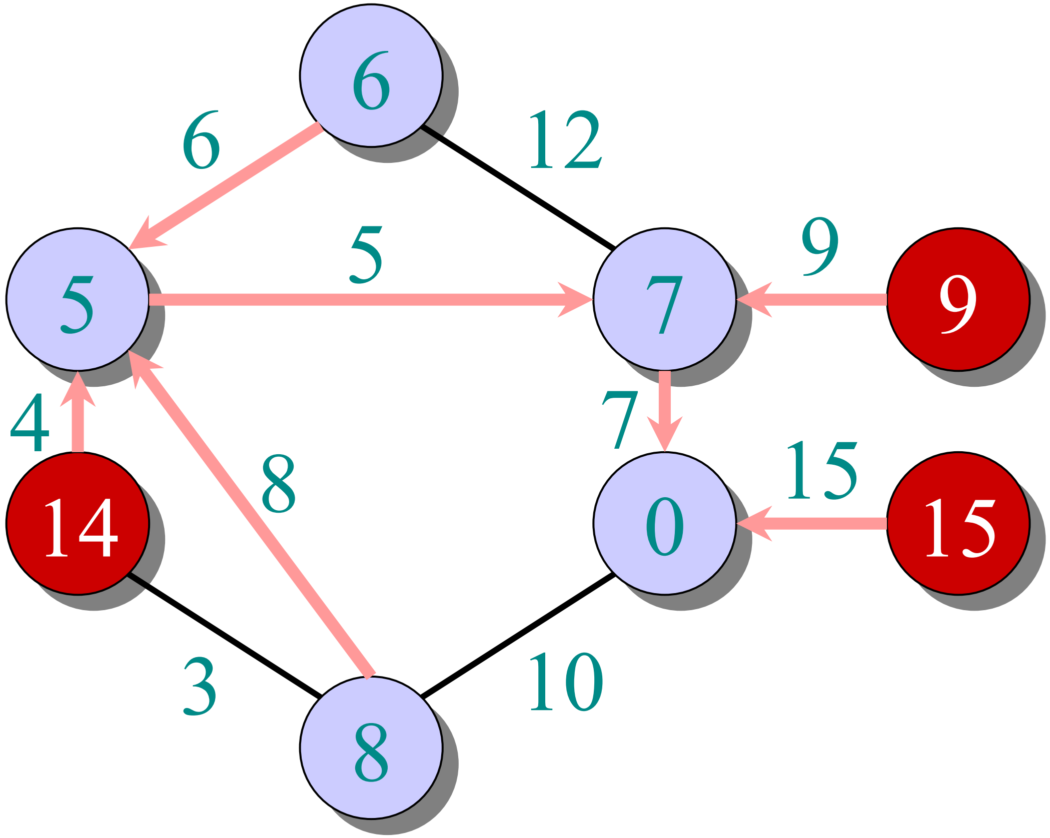


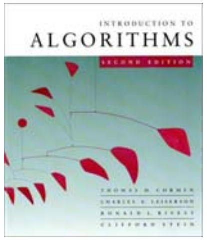


Example of Prim's algorithm

○ $\in A$

● $\in V - A$

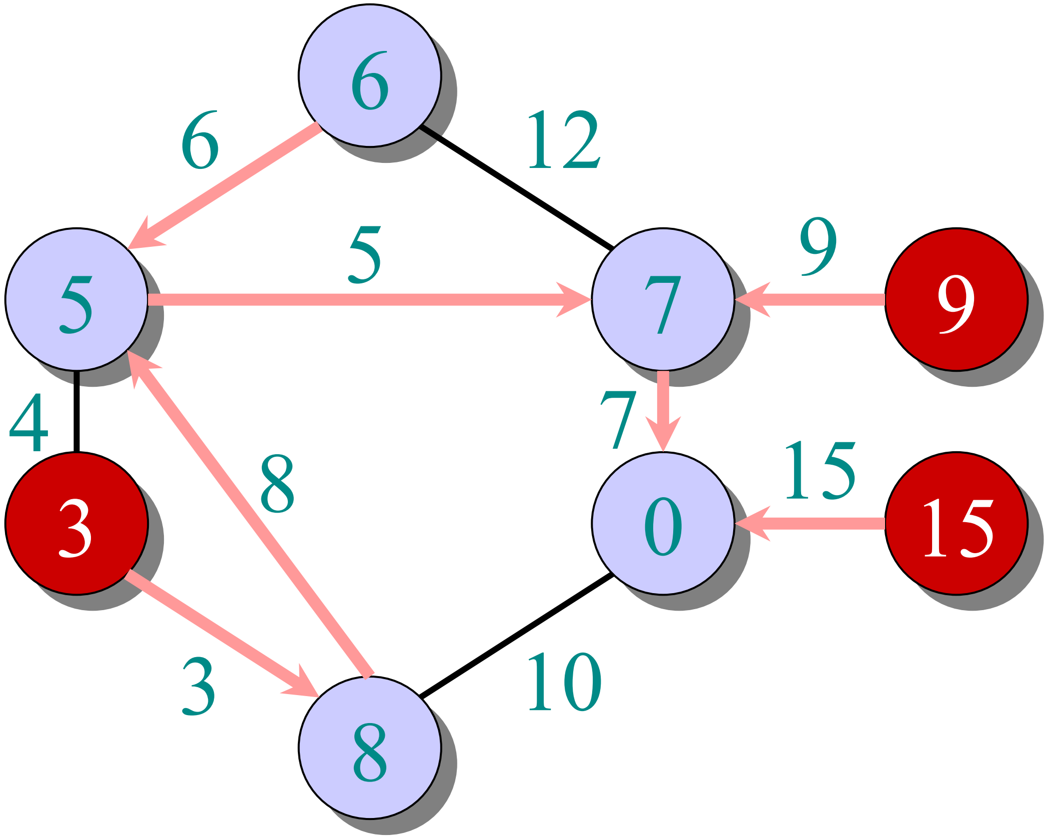


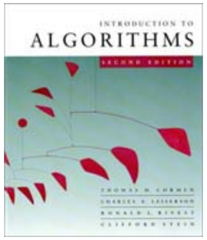


Example of Prim's algorithm

○ $\in A$

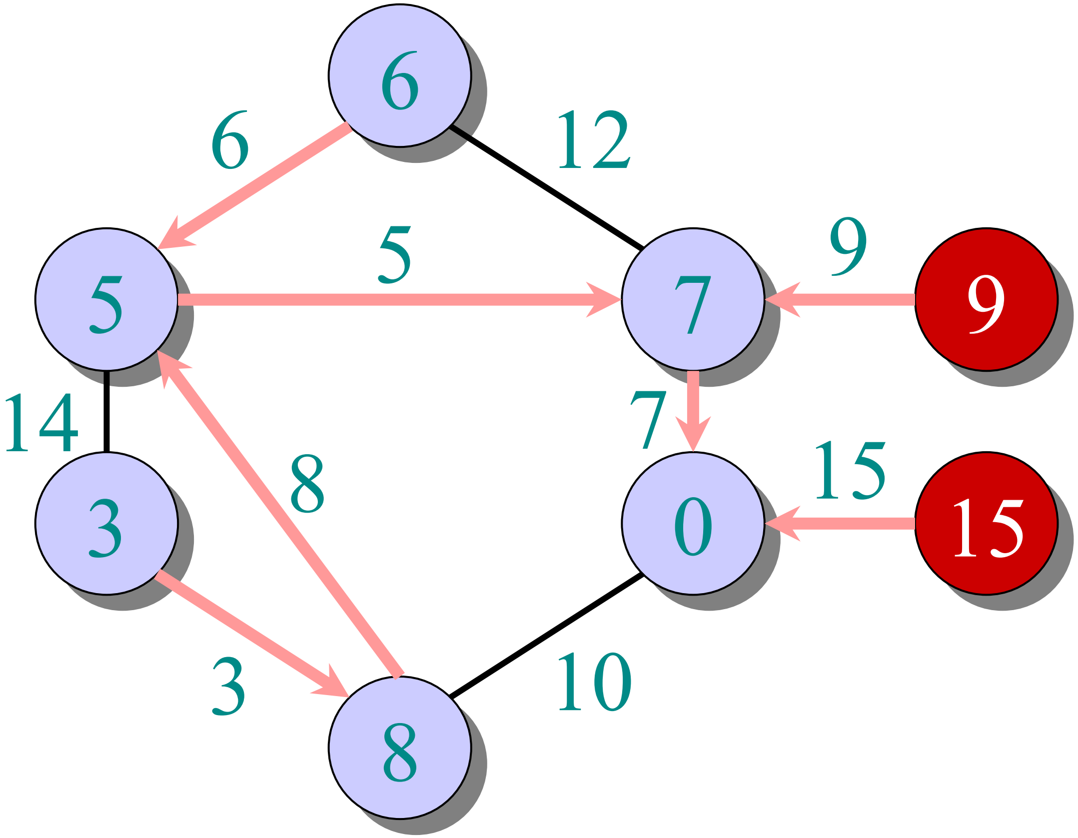
● $\in V - A$





Example of Prim's algorithm

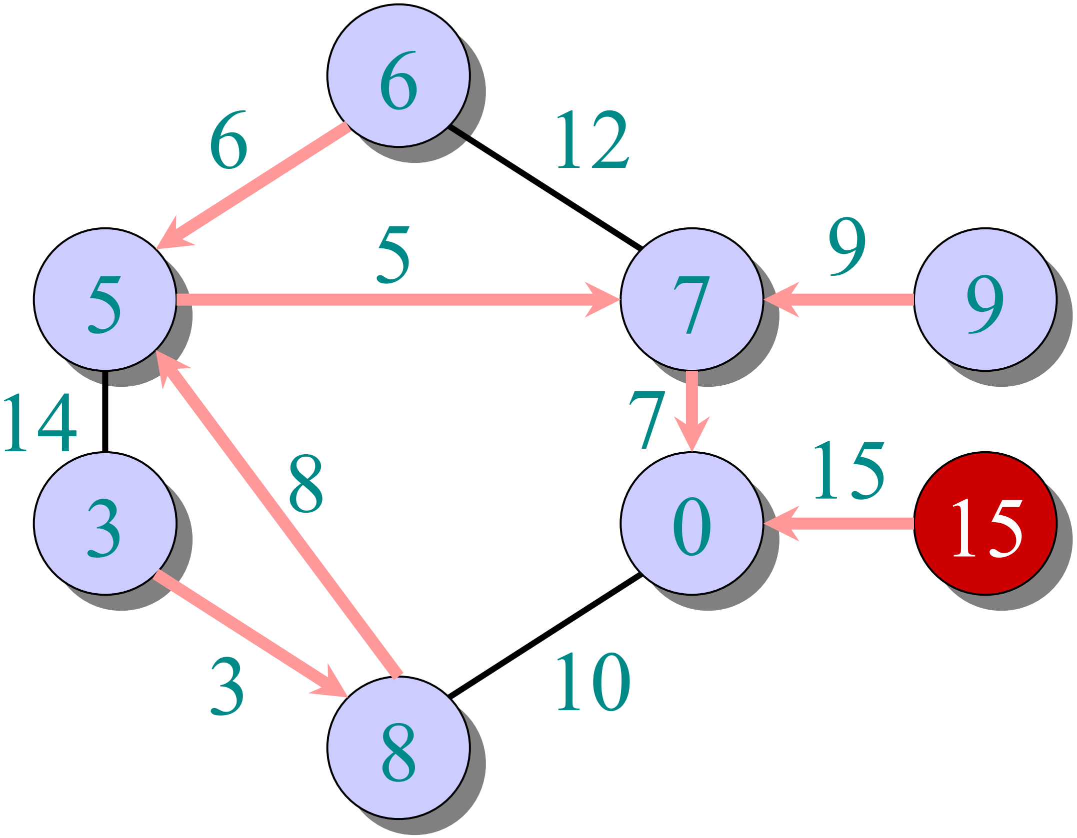
- ○ $\in A$
- ● $\in V - A$

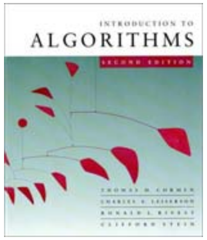


Example of Prim's algorithm

○ $\in A$

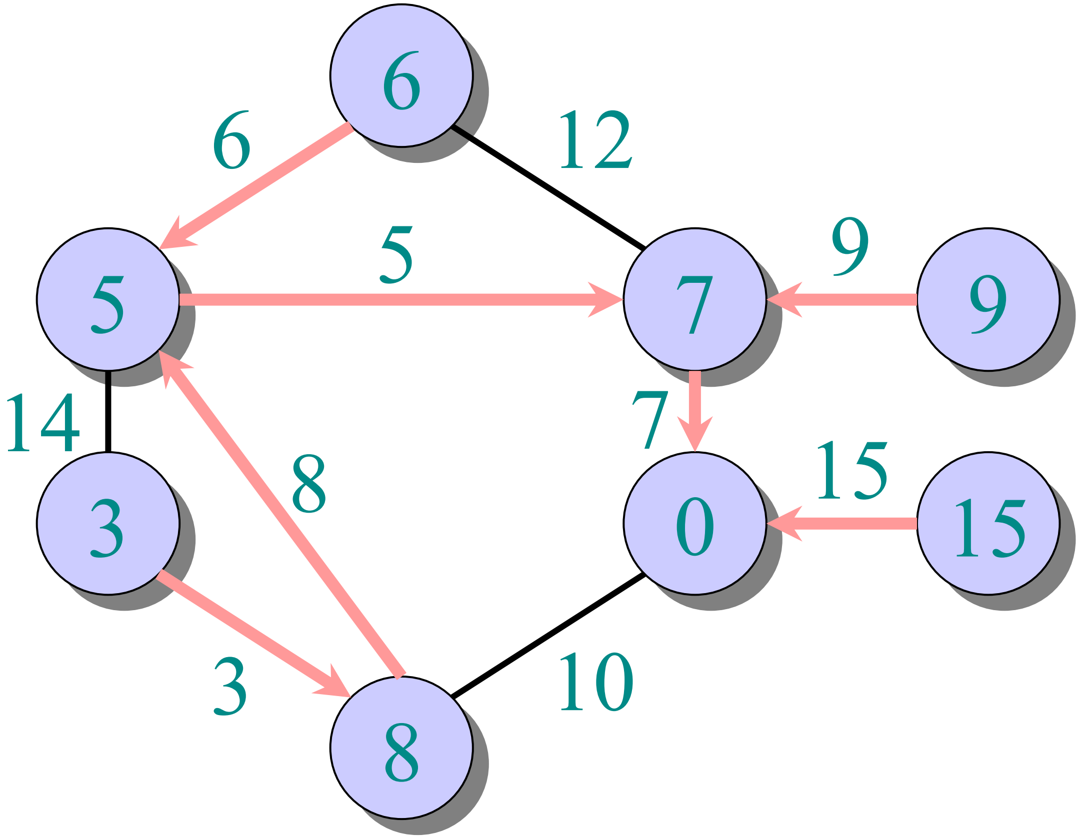
● $\in V - A$

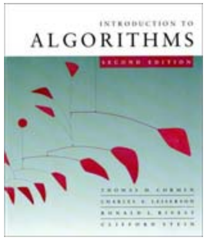




Example of Prim's algorithm

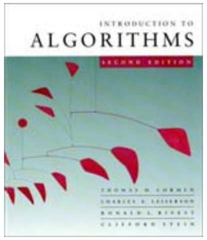
- ○ $\in A$
- ● $\in V - A$





Analysis of Prim

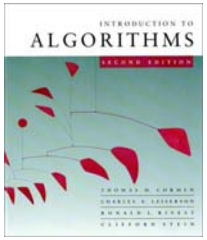
```
$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN$(Q)$
        for each $v \in Adj[u]$
            do if $v \in Q$ and $w(u, v) < key[v]$
                then $key[v] \leftarrow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



Analysis of Prim

$\Theta(V)$ total

```
$\left\{\begin{array}{l} Q \leftarrow V \\ key[v] \leftarrow \infty$ for all $v \in V \\ key[s] \leftarrow 0$ 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 $key[v] \leftarrow w(u, v)$
                    $\pi[v] \leftarrow u$
```



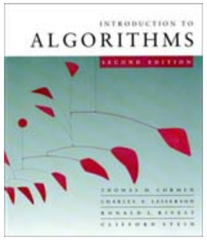
Analysis of Prim

$\Theta(V)$ 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

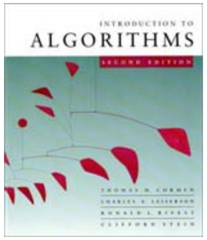
```
$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$
```

$|V|$ 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 $s \in V$
```

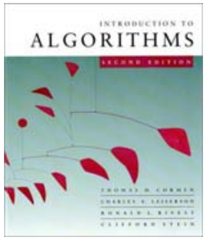
$|V|$ times

```
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 $s \in V$
```

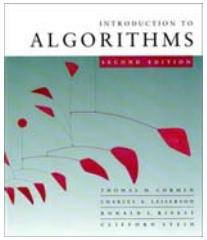
$|V|$ times

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

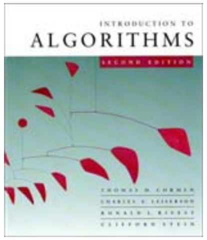
Handshaking Lemma $\Rightarrow$ $\Theta(E)$ implicit DECREASE-KEY's.

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



Analysis of Prim (continued)

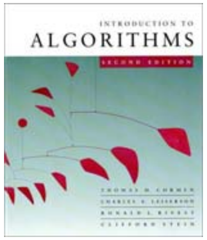
Time = $\Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$



Analysis of Prim (continued)

Time = $\Theta(V)\cdot T_{\textsc{Extract-Min}} + \Theta(E)\cdot T_{\textsc{Decrease-Key}}$

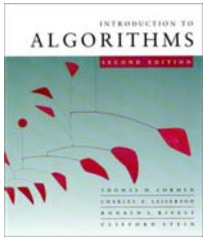
Q	$T_{\textsc{Extract-Min}}$	$T_{\textsc{Decrease-Key}}$	Total



Analysis of Prim (continued)

Time = $\Theta(V) \cdot T_{\textsc{Extract-Min}} + \Theta(E) \cdot T_{\textsc{Decrease-Key}}$

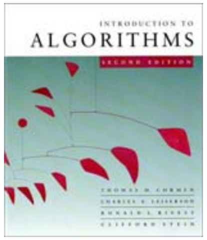
Q	$T_{\textsc{Extract-Min}}$	$T_{\textsc{Decrease-Key}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$



Analysis of Prim (continued)

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

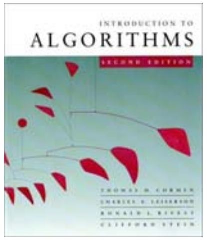
Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$



Analysis of Prim (continued)

Time = $\Theta(V)\cdot T_{\textsc{Extract-Min}} + \Theta(E)\cdot T_{\textsc{Decrease-Key}}$

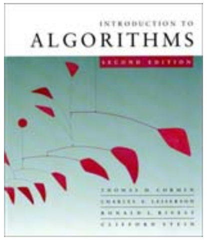
Q	$T_{\textsc{Extract-Min}}$	$T_{\textsc{Decrease-Key}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$ amortized	$O(1)$ amortized	$O(E + V \lg V)$ worst case



MST algorithms

Kruskal’s algorithm (see CLRS):

- Uses the ***disjoint-set data structure*** (Lecture 10).
- Running time = $O(E \lg V)$.



MST algorithms

Kruskal's algorithm (see CLRS):

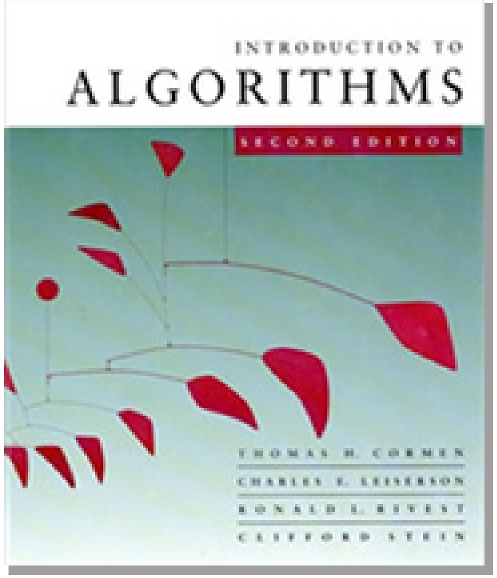
- Uses the ***disjoint-set data structure*** (Lecture 10).
- Running time = $O(E \lg V)$.

Best to date:

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

Introduction to Algorithms

6.046J/18.401J

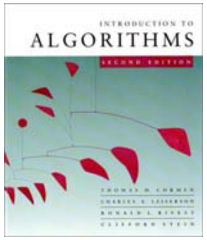


LECTURE 17

Shortest Paths I

- Properties of shortest paths
- Dijkstra's algorithm
- Correctness
- Analysis
- Breadth-first search

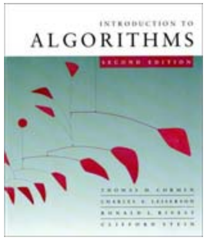
Prof. Erik Demaine



Paths in graphs

Consider a digraph $G = (V, E)$ with edge-weight function $w : E \to \mathbb{R}$. The ***weight*** of path $p = v_1 \to v_2 \to \cdots \to v_k$ is defined to be

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1}) .$$

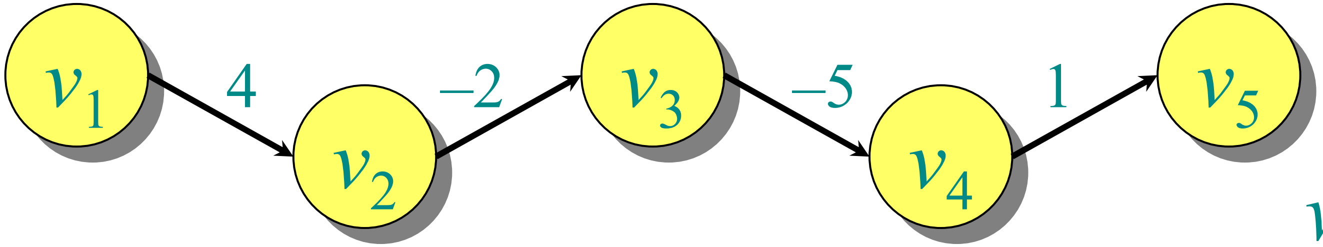


Paths in graphs

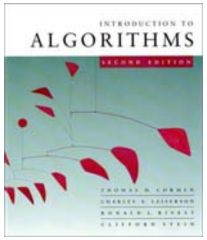
Consider a digraph $G = (V, E)$ with edge-weight function $w : E \to \mathbb{R}$. The ***weight*** of path $p = v_1 \to v_2 \to \cdots \to v_k$ is defined to be

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1}) .$$

Example:



$w(p) = –2$

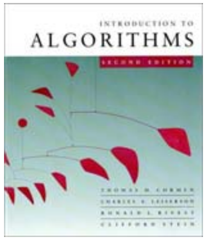


Shortest paths

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

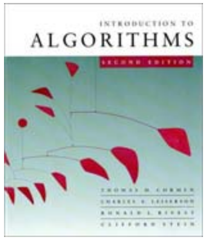
$$\delta(u, v) = \min\{w(p) : p \text{ is a path from } u \text{ to } v\}.$$

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



Optimal substructure

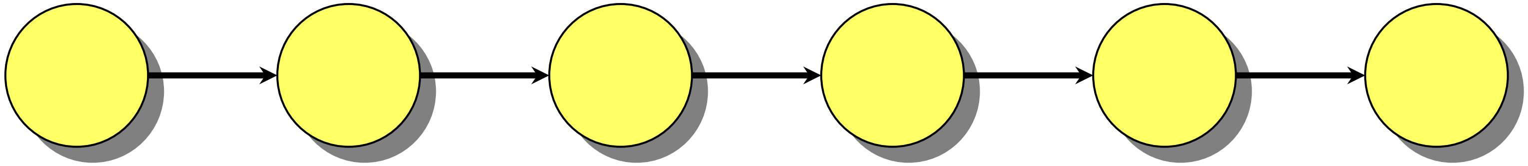
Theorem. A subpath of a shortest path is a shortest path.



Optimal substructure

Theorem. A subpath of a shortest path is a shortest path.

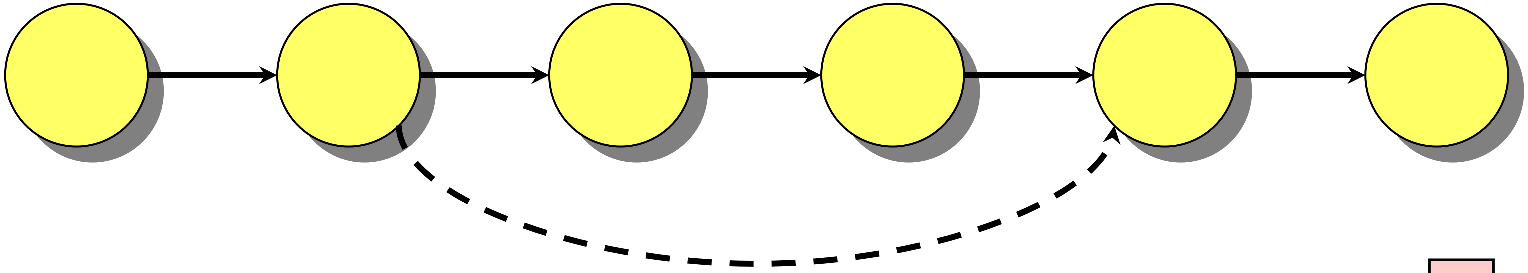
Proof. Cut and paste:



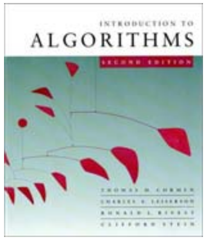
Optimal substructure

Theorem. A subpath of a shortest path is a shortest path.

Proof. Cut and paste:



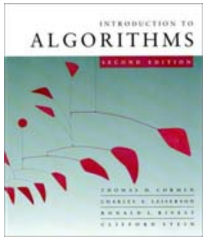
□



Triangle inequality

Theorem. For all $u, v, x \in V$, we have

$$\delta(u, v) \le \delta(u, x) + \delta(x, v).$$

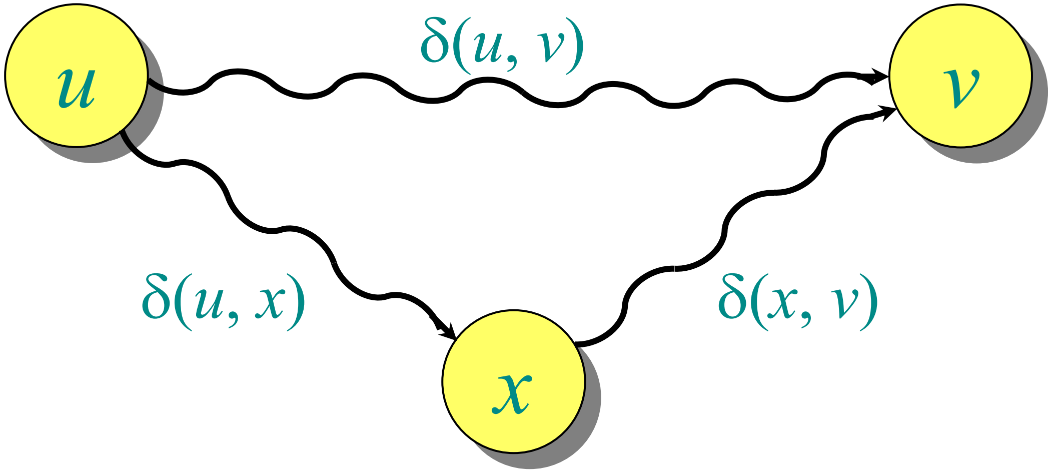


Triangle inequality

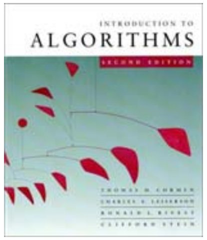
Theorem. For all $u, v, x \in V$, we have

$$\delta(u, v) \leq \delta(u, x) + \delta(x, v).$$

Proof.



□



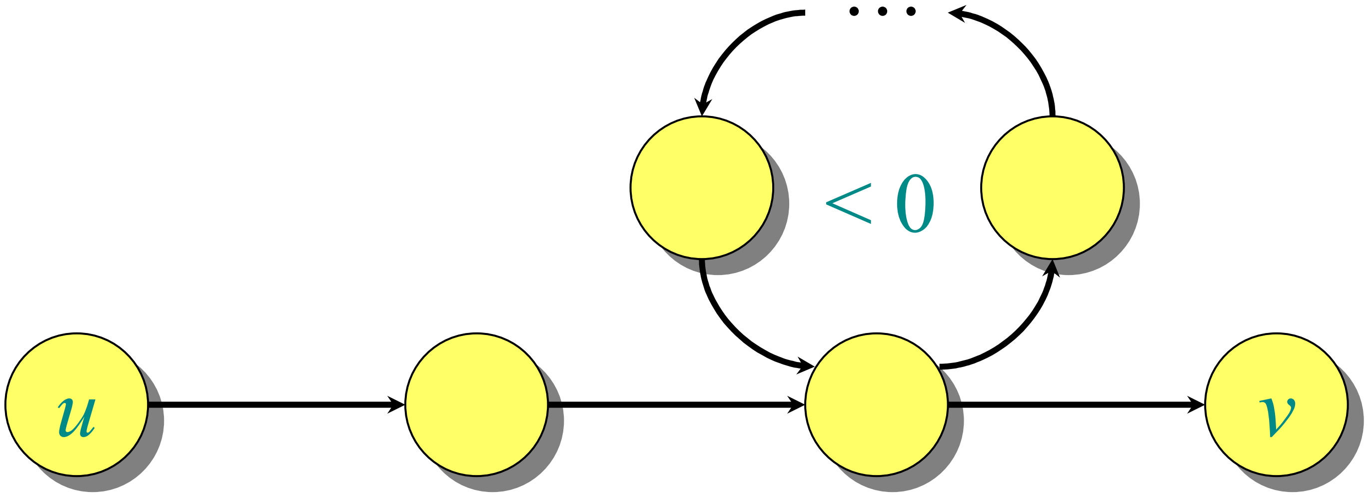
Well-definedness of shortest paths

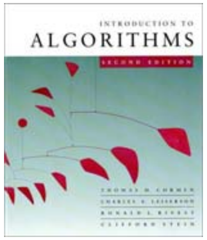
If a graph G contains a negative-weight cycle, then some shortest paths may not exist.

Well-definedness of shortest paths

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

Example:





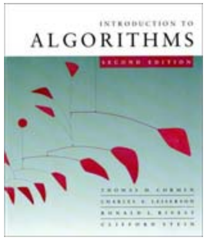
Single-source shortest paths

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.

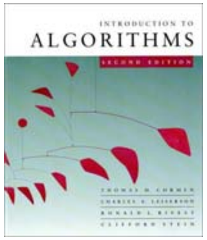
IDEA: Greedy.

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.



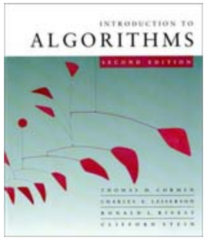
Dijkstra’s algorithm

```
$d[s] \leftarrow 0$
for each $v \in V - \{s\}$
    do $d[v] \leftarrow \infty$
$S \leftarrow \emptyset$
$Q \leftarrow V$            ⊳ $Q$ is a priority queue maintaining $V - S$
```



Dijkstra's algorithm

```
$d[s] \leftarrow 0$
for each $v \in V - \{s\}$
    do $d[v] \leftarrow \infty$
$S \leftarrow \varnothing$
$Q \leftarrow V$          ▷ $Q$ 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 $d[v] \leftarrow d[u] + w(u, v)$
```



Dijkstra’s algorithm

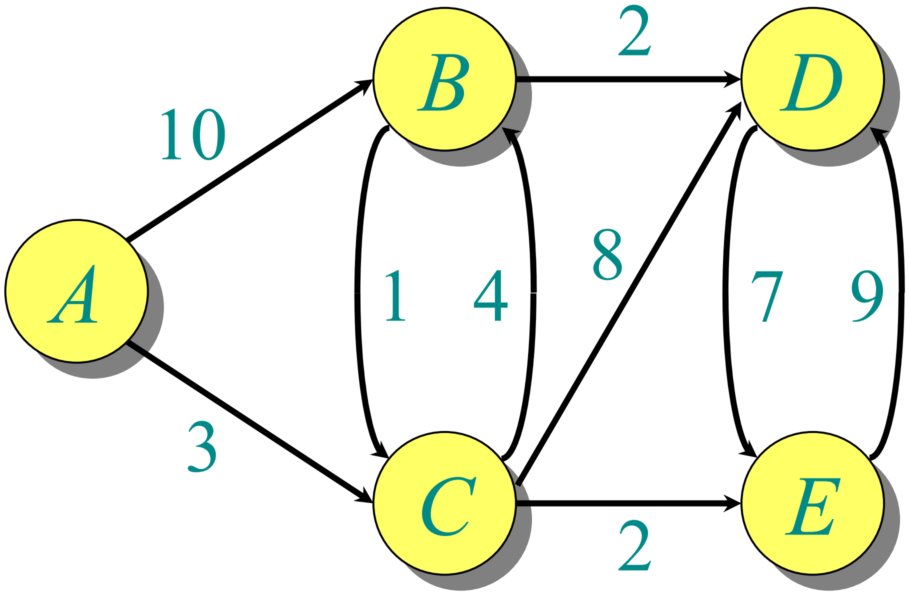
```
$d[s] \leftarrow 0$
for each $v \in V - \{s\}$
    do $d[v] \leftarrow \infty$
$S \leftarrow \varnothing$
$Q \leftarrow V$            ▷ $Q$ is a priority queue maintaining $V - S$
while $Q \neq \varnothing$
    do $u \leftarrow$ EXTRACT-MIN$(Q)$
       $S \leftarrow S \cup \{u\}$
       for each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

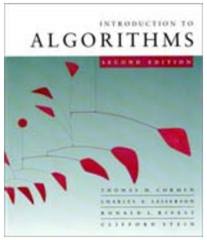
relaxation step

Implicit DECREASE-KEY

Example of Dijkstra's algorithm

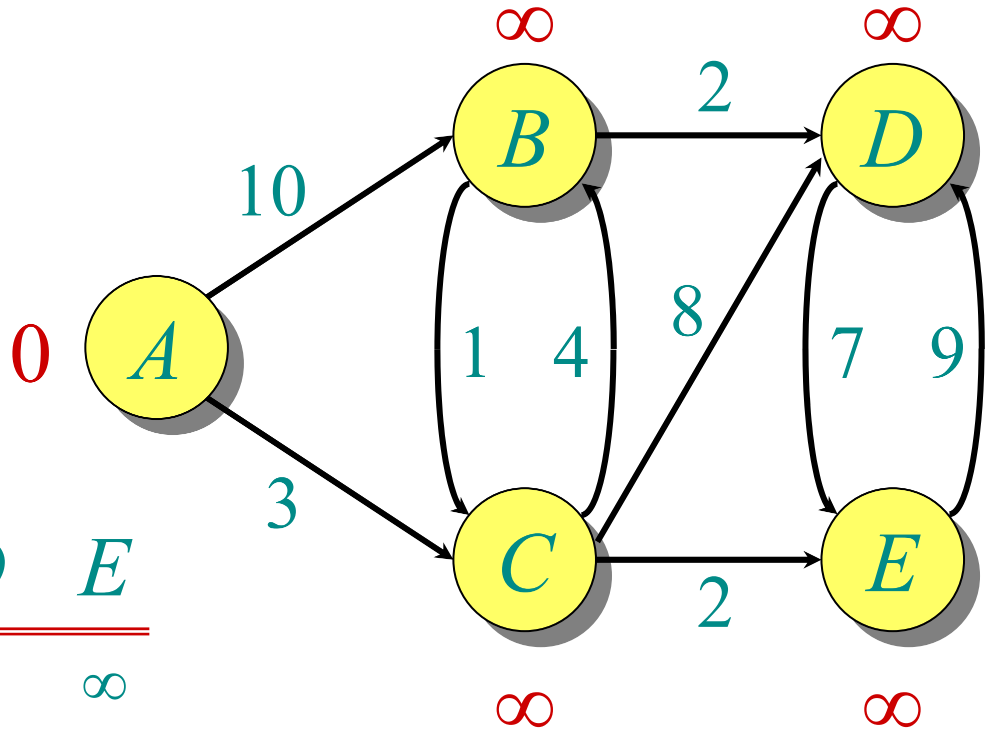
Graph with nonnegative edge weights:





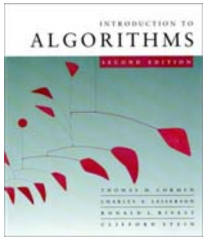
Example of Dijkstra’s algorithm

Initialize:



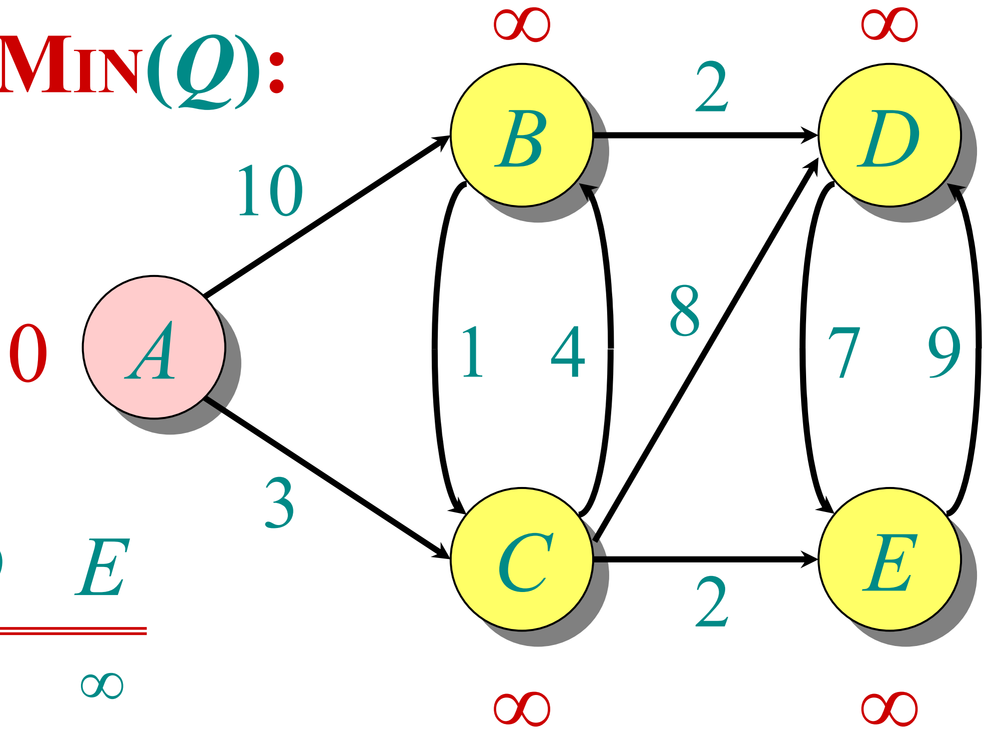
Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {}



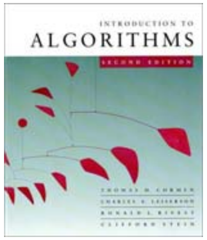
Example of Dijkstra’s algorithm

“A” ← **EXTRACT-MIN**(*Q*):



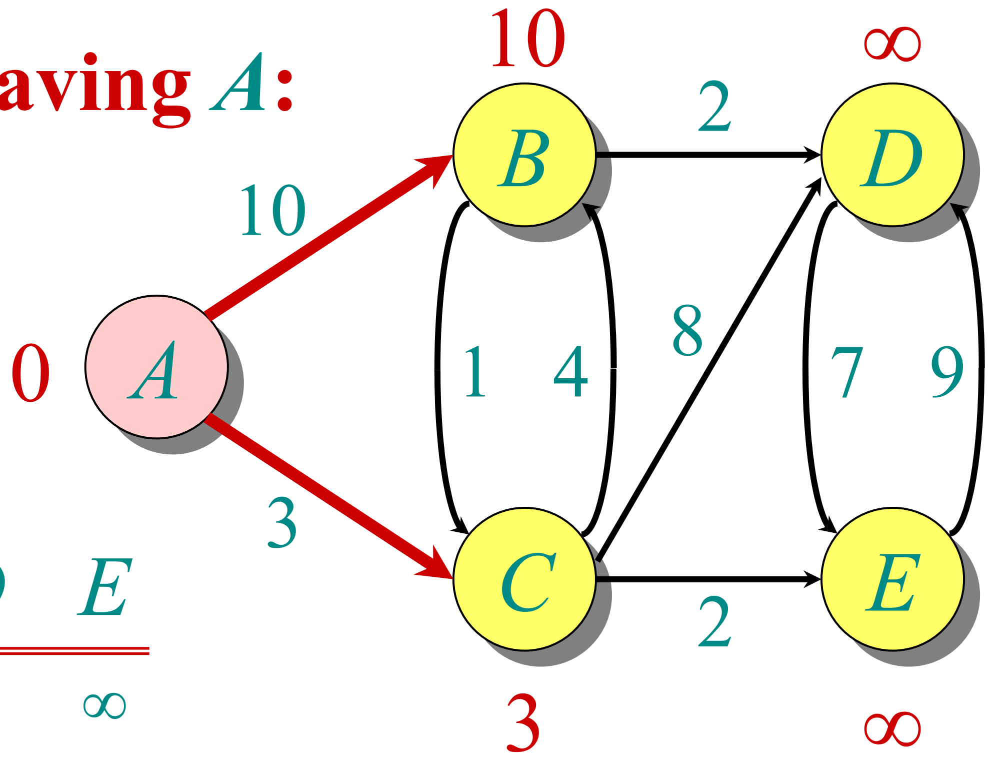
Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞

S: { *A* }



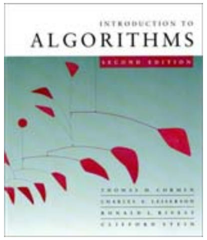
Example of Dijkstra's algorithm

Relax all edges leaving *A*:



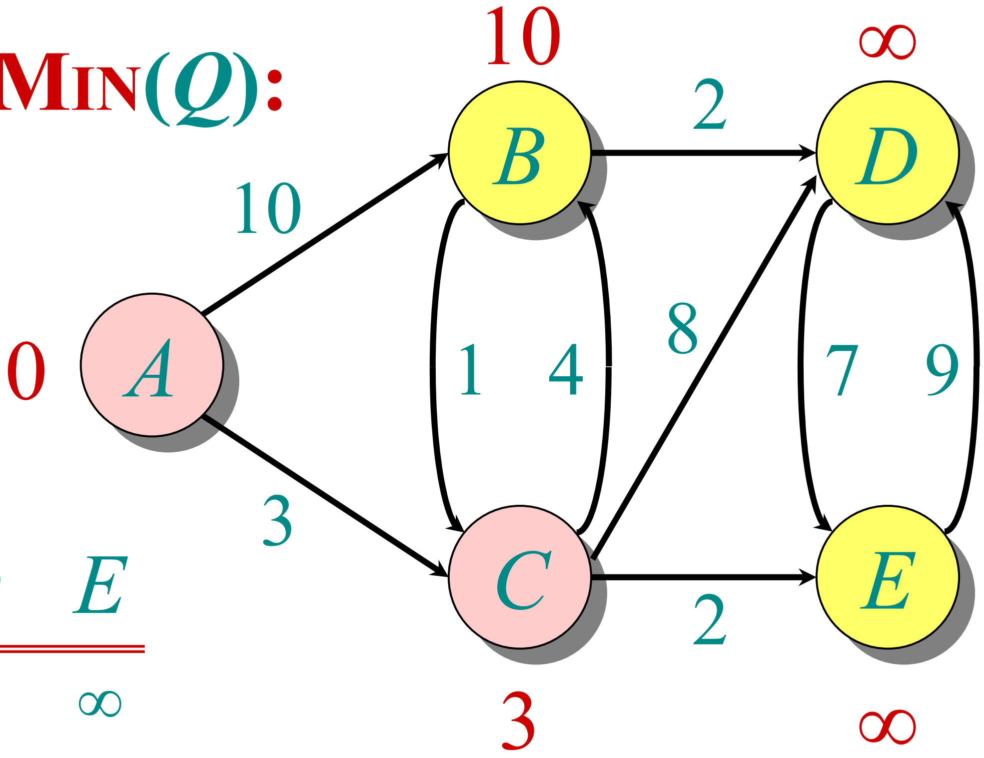
Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { *A* }



Example of Dijkstra’s algorithm

“*C*” ← EXTRACT-MIN(*Q*):

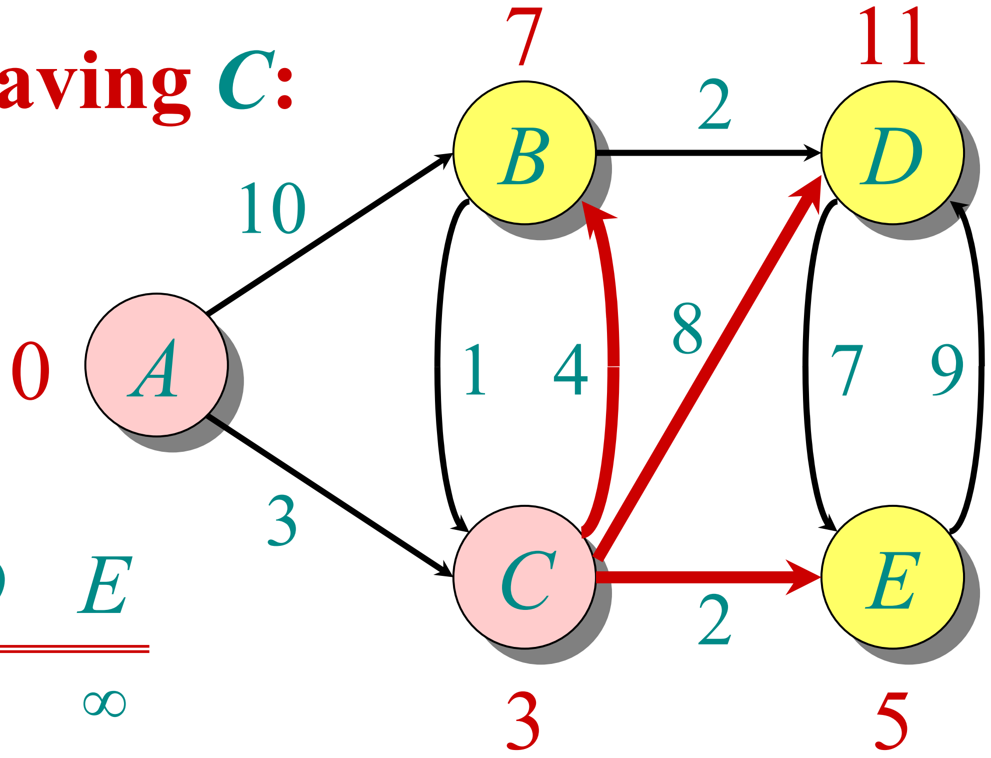


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { *A*, *C* }

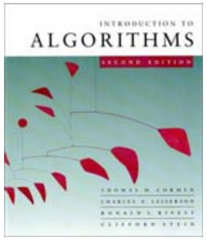
Example of Dijkstra's algorithm

Relax all edges leaving *C*:



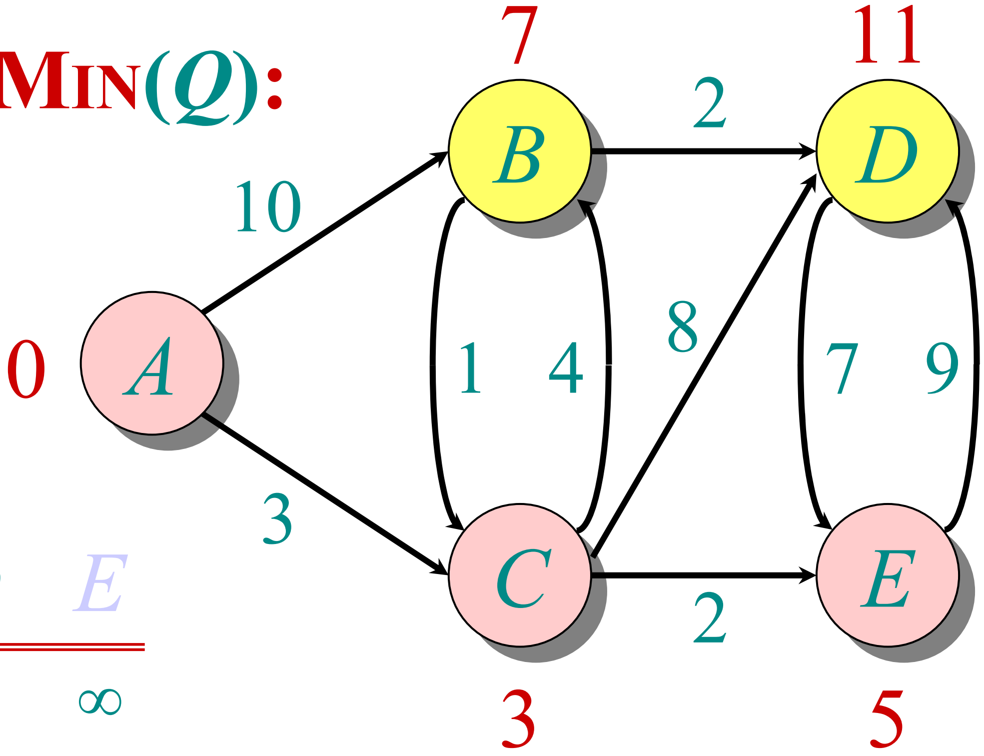
Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { *A*, *C* }



Example of Dijkstra’s algorithm

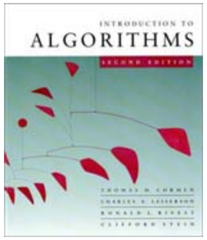
“*E*” ← EXTRACT-MIN(*Q*):



Q:

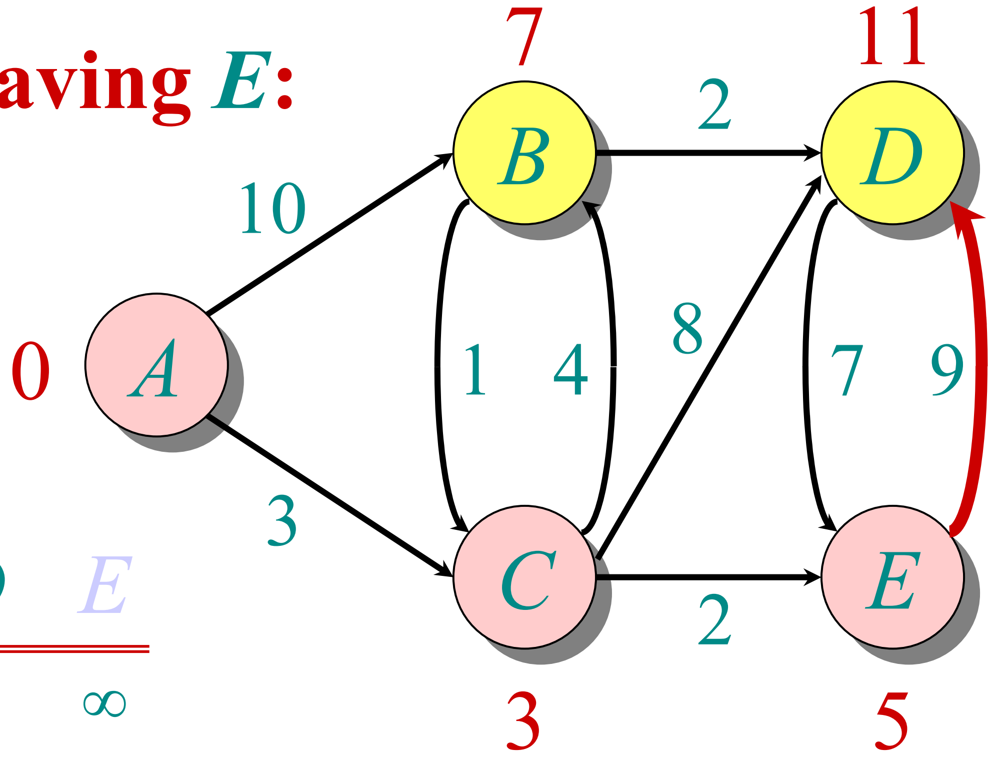
A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5

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



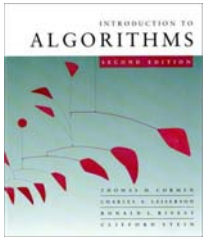
Example of Dijkstra's algorithm

Relax all edges leaving *E*:



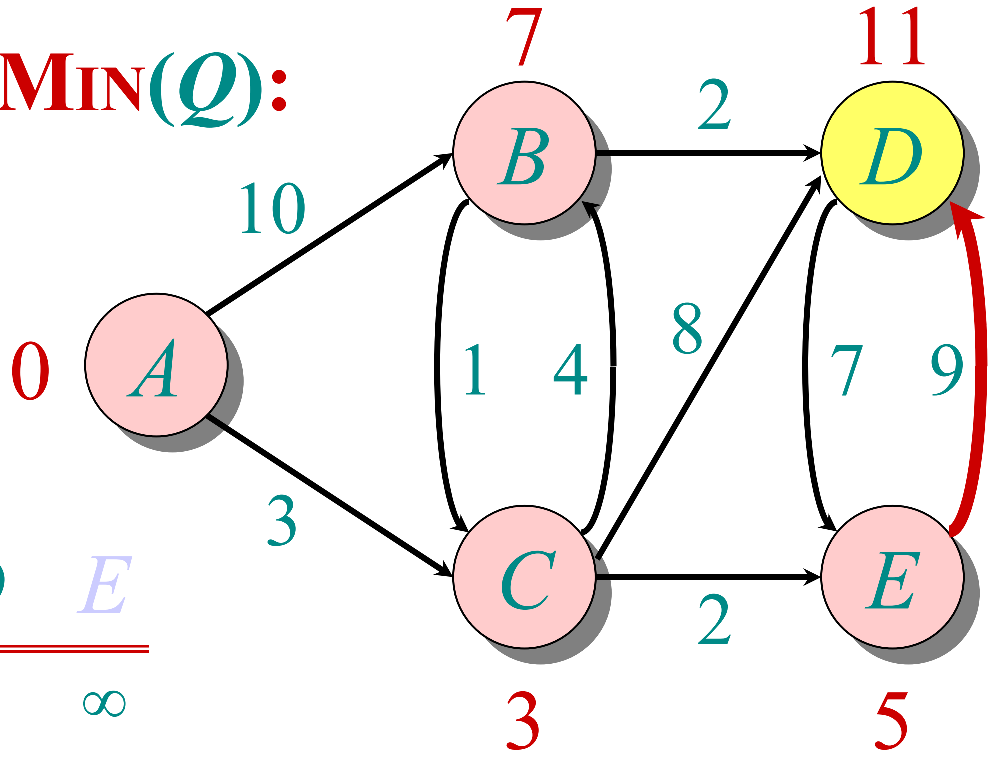
Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

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



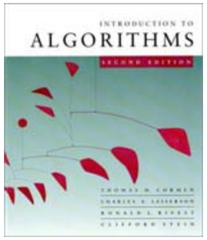
Example of Dijkstra’s algorithm

“*B*” ← **EXTRACT-MIN**(*Q*):



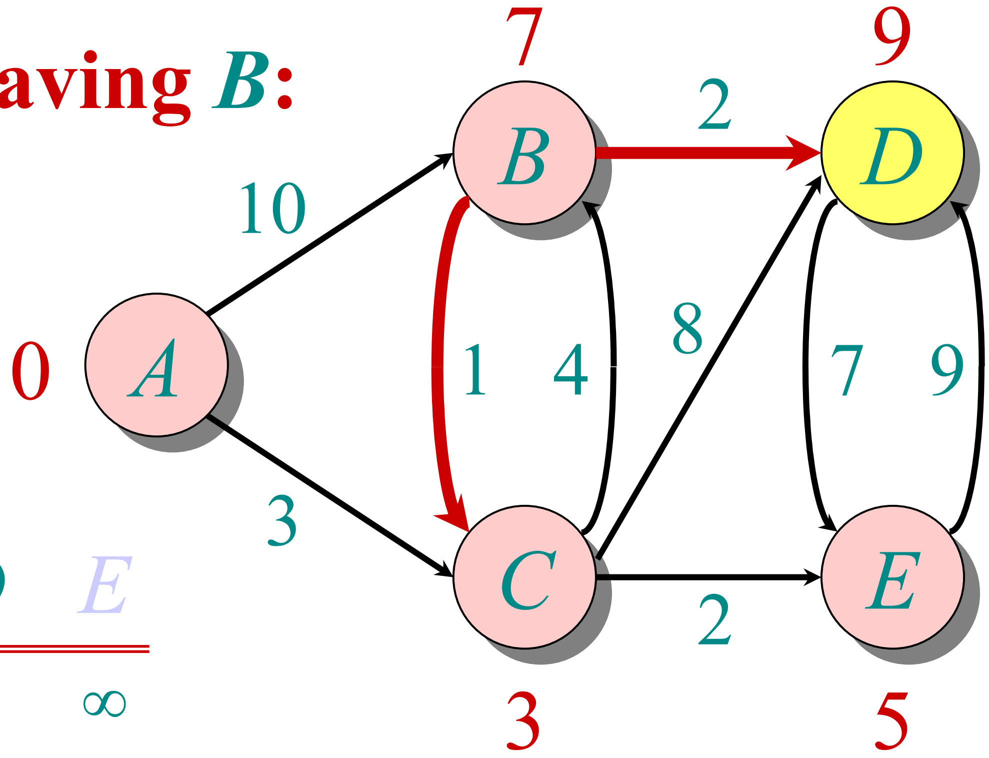
Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

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



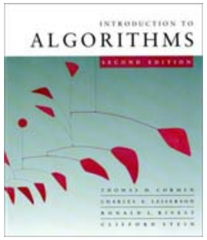
Example of Dijkstra's algorithm

Relax all edges leaving *B*:



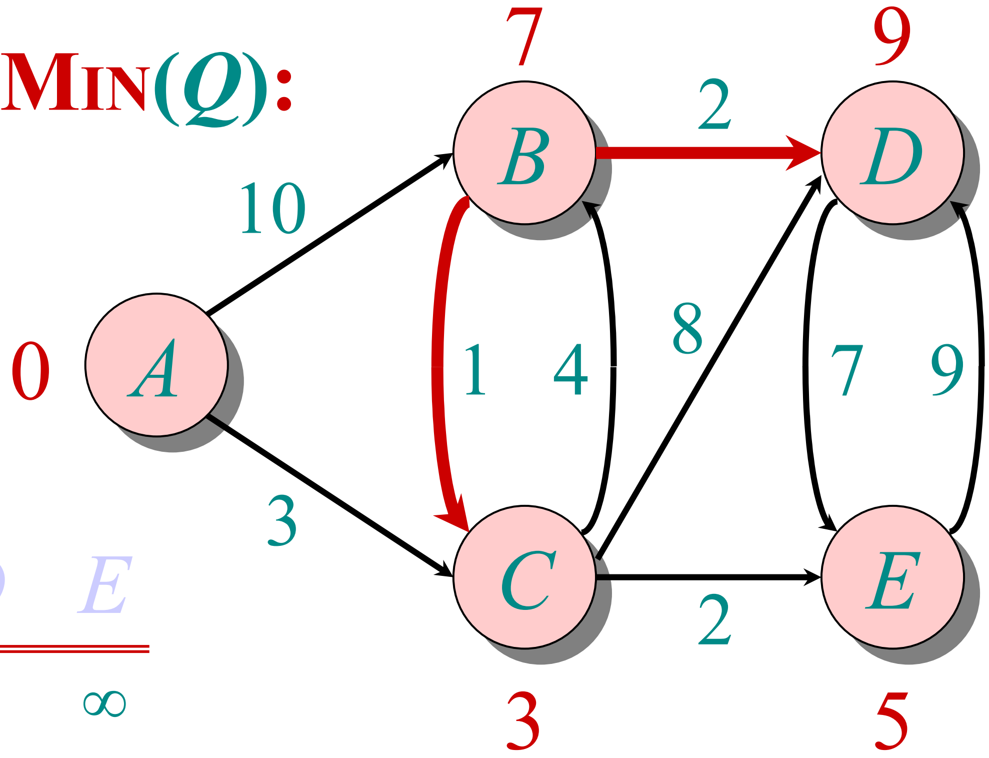
Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

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



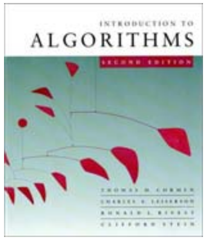
Example of Dijkstra's algorithm

"D" ← **EXTRACT-MIN(*Q*):**



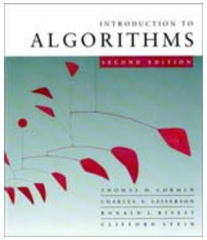
Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

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



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.



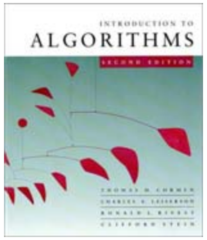
Correctness — Part I

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,

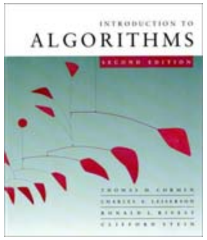
$$\begin{aligned} d[v] &< \delta(s, v) && \text{supposition} \\ &\leq \delta(s, u) + \delta(u, v) && \text{triangle inequality} \\ &\leq \delta(s,u) + w(u, v) && \text{sh. path} \leq \text{specific path} \\ &\leq d[u] + w(u, v) && v \text{ is first violation} \end{aligned}$$

Contradiction. □



Correctness — Part II

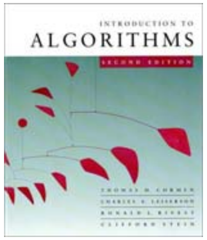
Lemma. Let u be v's predecessor on a shortest path from s to v. Then, if $d[u] = \delta(s, u)$ and edge (u, v) is relaxed, we have $d[v] = \delta(s, v)$ after the relaxation.



Correctness — Part II

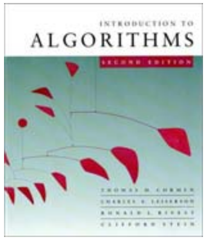
Lemma. Let u be v's predecessor on a shortest path from s to v. Then, if $d[u] = \delta(s, u)$ and edge (u, v) is relaxed, we have $d[v] = \delta(s, v)$ after the relaxation.

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

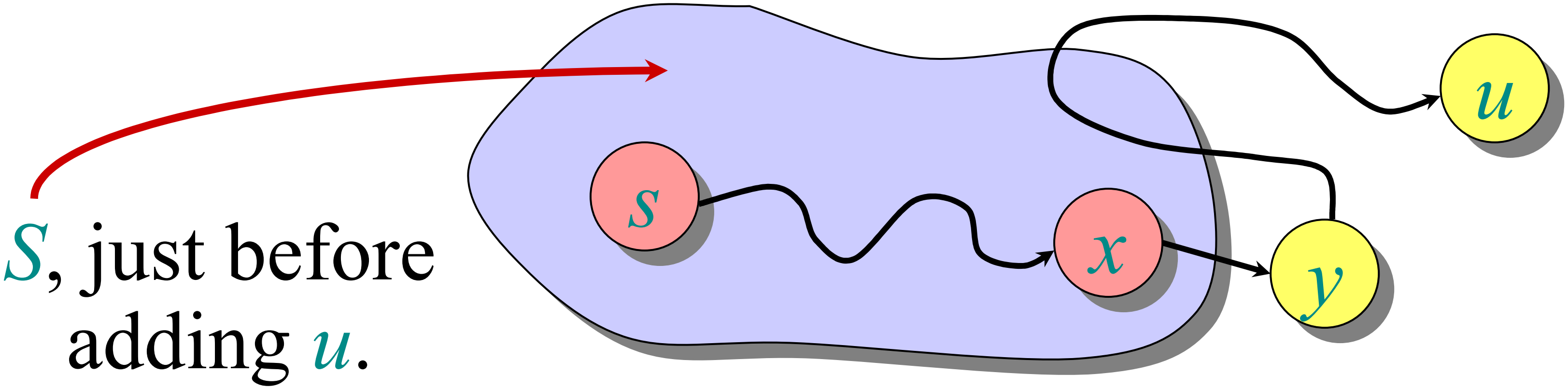
Theorem. Dijkstra's algorithm terminates with $d[v] = \delta(s, v)$ for all $v \in V$.



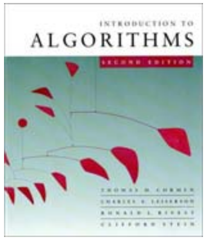
Correctness — Part III

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

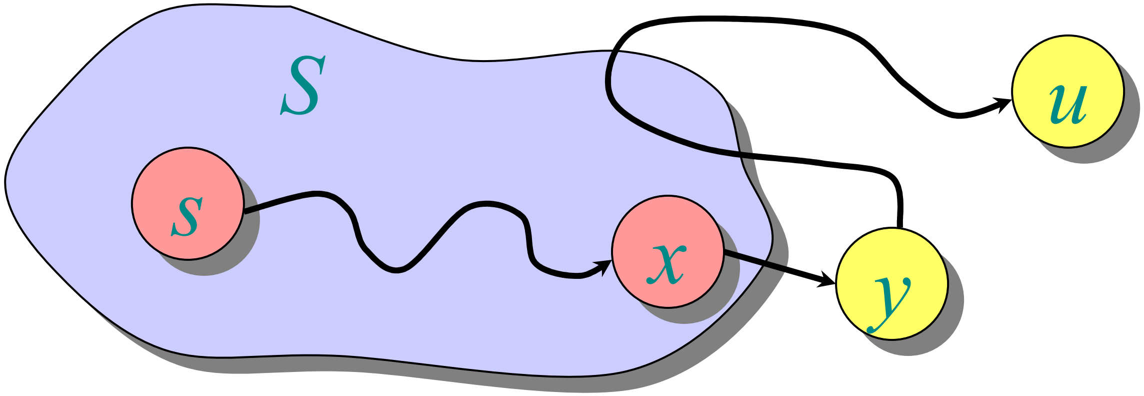
Proof. It suffices to show that $d[v] = \delta(s, v)$ for every $v \in V$ when v is added to S. Suppose u is the first vertex added to S for which $d[u] > \delta(s, u)$. Let y be the first vertex in $V - S$ along a shortest path from s to u, and let x be its predecessor:



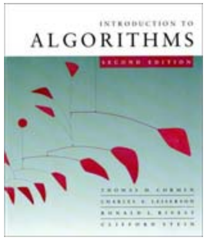
S, just before adding u.



Correctness — Part III (continued)

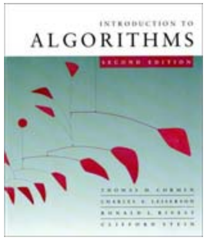


Since u is the first vertex violating the claimed invariant, we have $d[x] = \delta(s, x)$. When x was added to S, the edge (x, y) was relaxed, which implies that $d[y] = \delta(s, y) \le \delta(s, u) < d[u]$. But, $d[u] \le d[y]$ by our choice of u. Contradiction. ∎



Analysis of Dijkstra

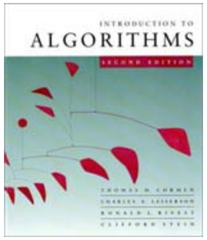
```
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN($Q$)
       $S \leftarrow S \cup \{u\}$
       for each $v \in Adj[u]$
           do if $d[v] > d[u] + w(u, v)$
                 then $d[v] \leftarrow d[u] + w(u, v)$
```



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

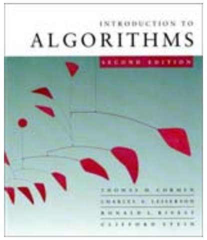


Analysis of Dijkstra

$|V|$ times

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



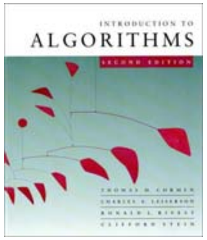
Analysis of Dijkstra

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

$|V|$ times

$degree(u)$ times

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



Analysis of Dijkstra

$|V|$ times

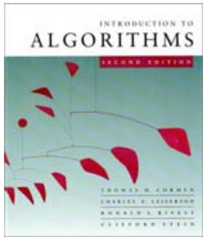
$degree(u)$ times

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

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

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

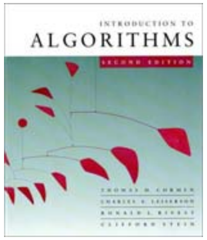
Note: Same formula as in the analysis of Prim's minimum spanning tree algorithm.



Analysis of Dijkstra (continued)

Time = $\Theta(V) \cdot T_{\textsc{Extract-Min}} + \Theta(E) \cdot T_{\textsc{Decrease-Key}}$

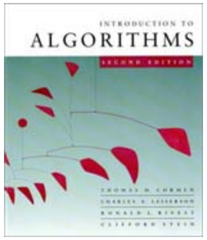
Q	$T_{\textsc{Extract-Min}}$	$T_{\textsc{Decrease-Key}}$	Total



Analysis of Dijkstra (continued)

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

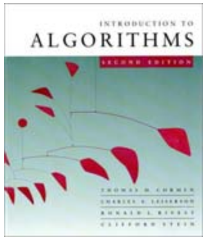
Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$



Analysis of Dijkstra (continued)

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

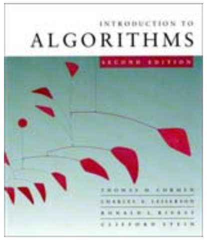
Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$



Analysis of Dijkstra (continued)

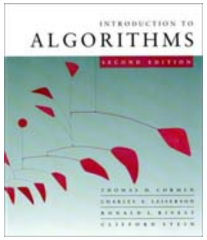
Time = $\Theta(V) \cdot T_{\textsc{Extract-Min}} + \Theta(E) \cdot T_{\textsc{Decrease-Key}}$

Q	$T_{\textsc{Extract-Min}}$	$T_{\textsc{Decrease-Key}}$	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



Unweighted graphs

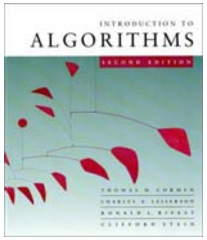
Suppose that $w(u, v) = 1$ for all $(u, v) \in E$.
Can Dijkstra's algorithm be improved?



Unweighted graphs

Suppose that $w(u, v) = 1$ for all $(u, v) \in E$.
Can Dijkstra's algorithm be improved?

- Use a simple FIFO queue instead of a priority queue.



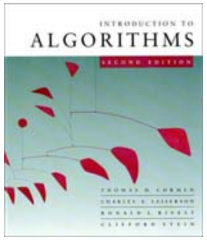
Unweighted graphs

Suppose that $w(u, v) = 1$ for all $(u, v) \in E$. Can Dijkstra's algorithm be improved?

- Use a simple FIFO queue instead of a priority queue.

Breadth-first search

```
while $Q \neq \emptyset$
    do $u \leftarrow$ DEQUEUE$(Q)$
        for each $v \in Adj[u]$
            do if $d[v] = \infty$
                then $d[v] \leftarrow d[u] + 1$
                    ENQUEUE$(Q, v)$
```



Unweighted graphs

Suppose that $w(u, v) = 1$ for all $(u, v) \in E$. Can Dijkstra's algorithm be improved?

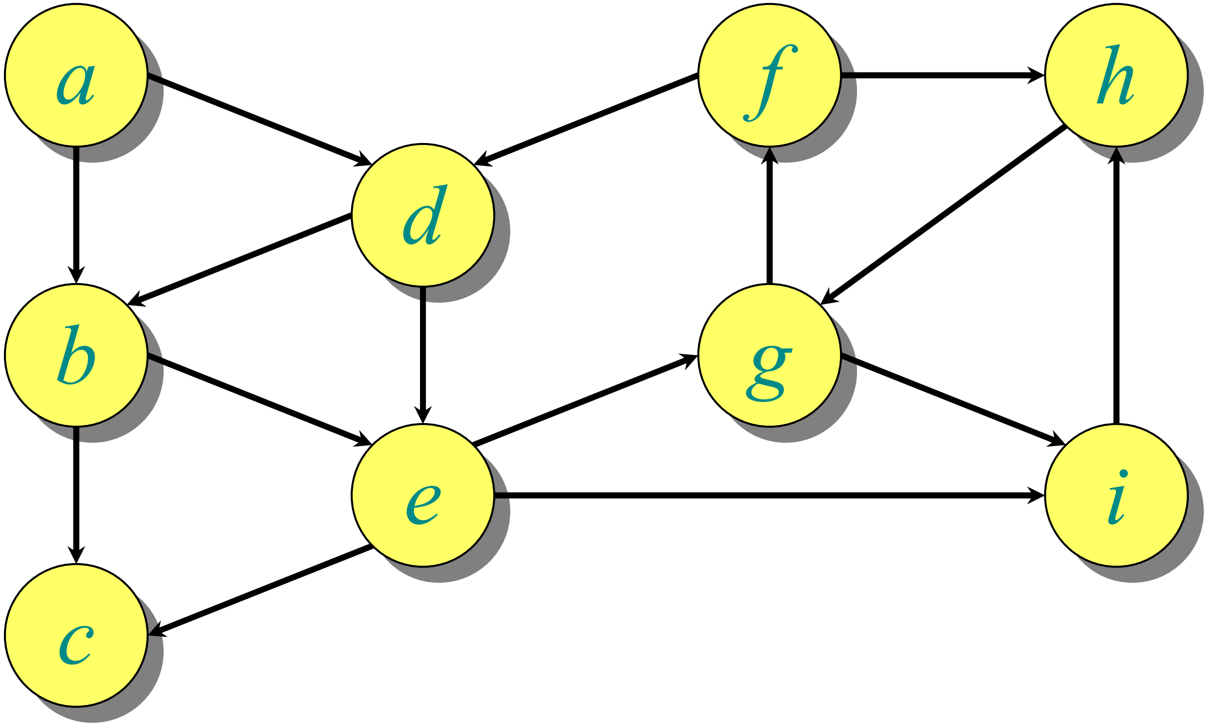
- Use a simple FIFO queue instead of a priority queue.

Breadth-first search

```
while $Q \neq \emptyset$
    do $u \leftarrow$ DEQUEUE($Q$)
        for each $v \in Adj[u]$
            do if $d[v] = \infty$
                then $d[v] \leftarrow d[u] + 1$
                     ENQUEUE($Q, v$)
```

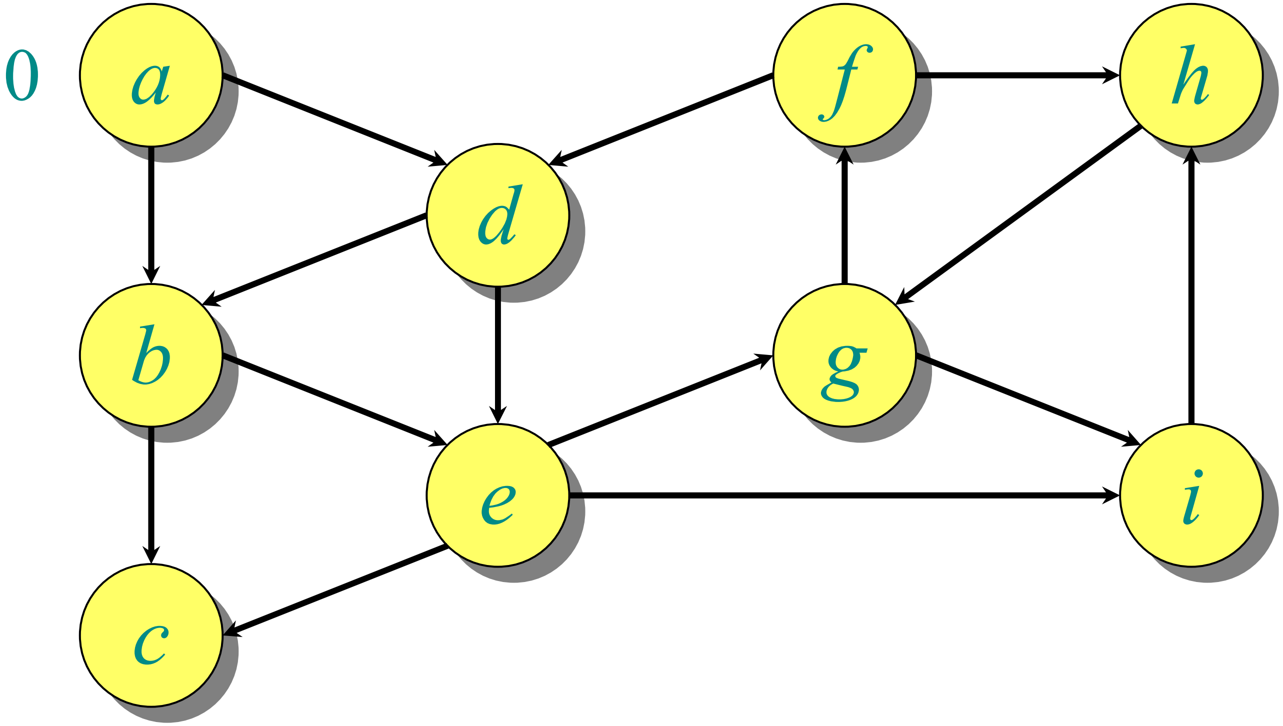
Analysis: Time $= O(V + E)$.

Example of breadth-first search



Q:

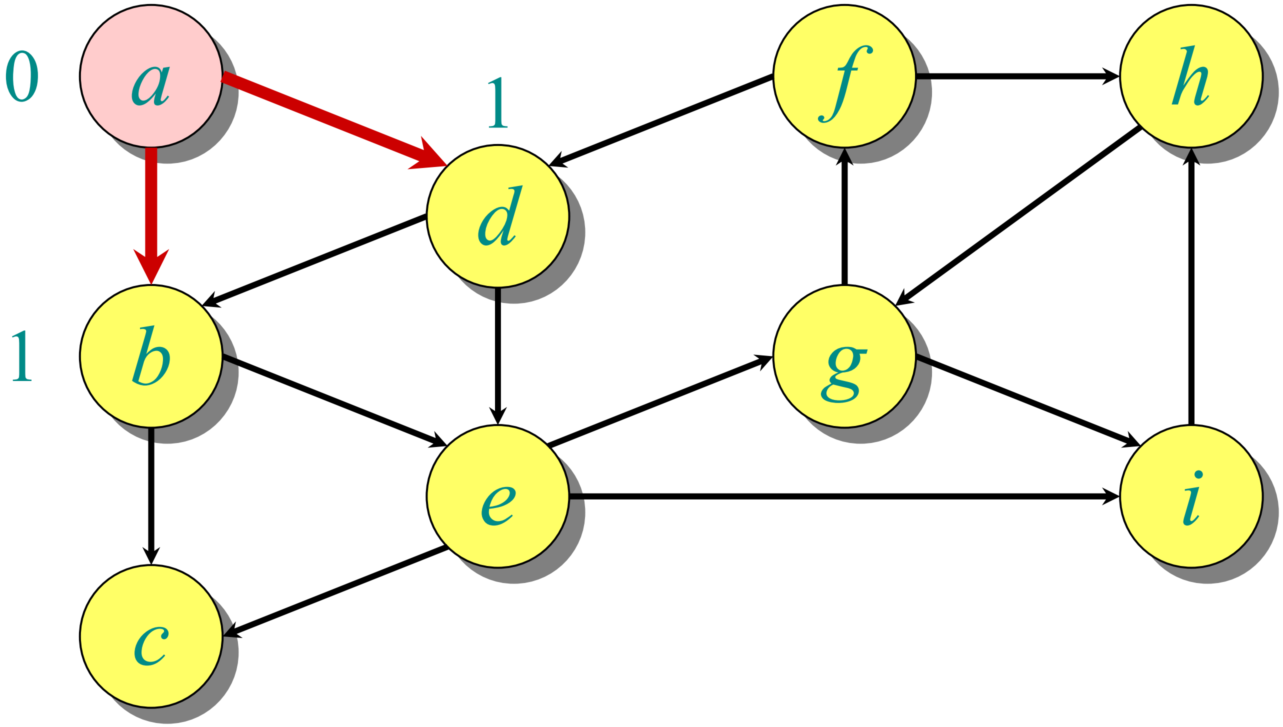
Example of breadth-first search



0

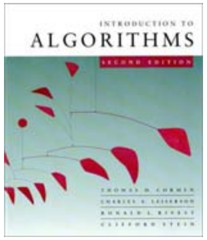
Q: *a*

Example of breadth-first search

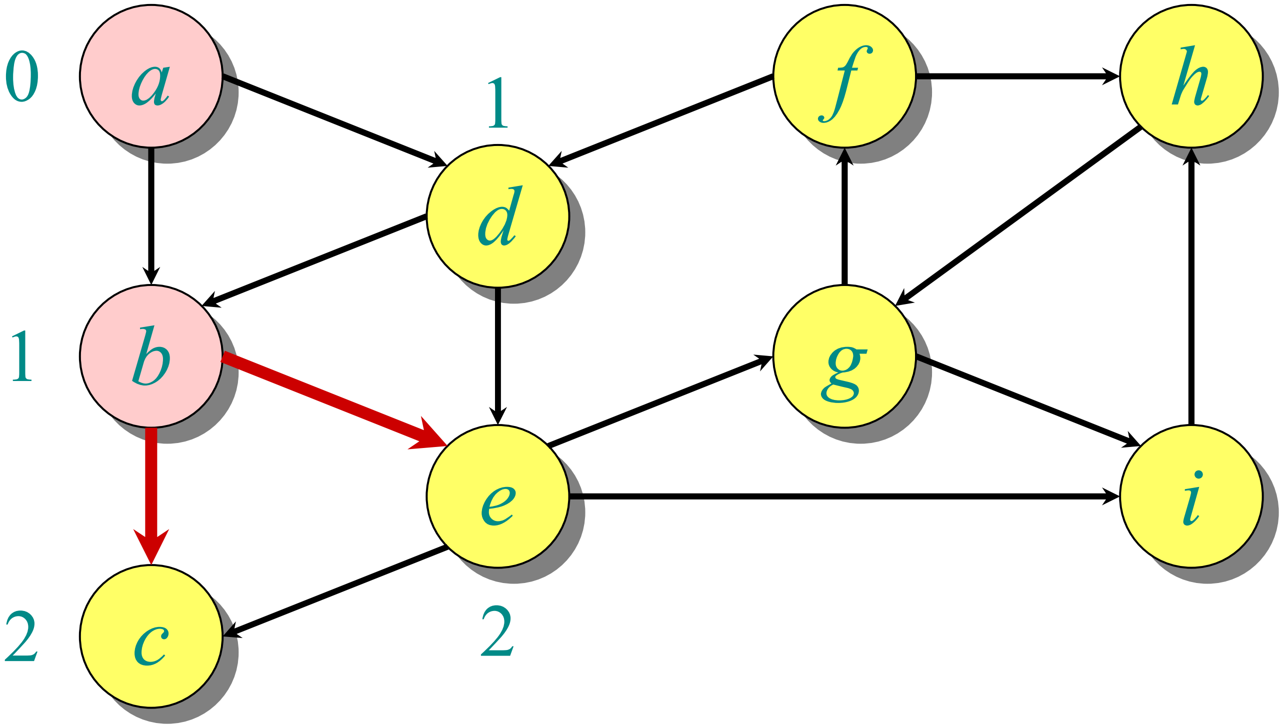


1 1

Q: *a* *b* *d*

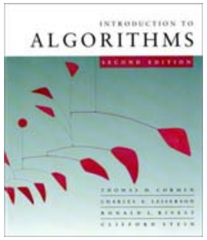


Example of breadth-first search

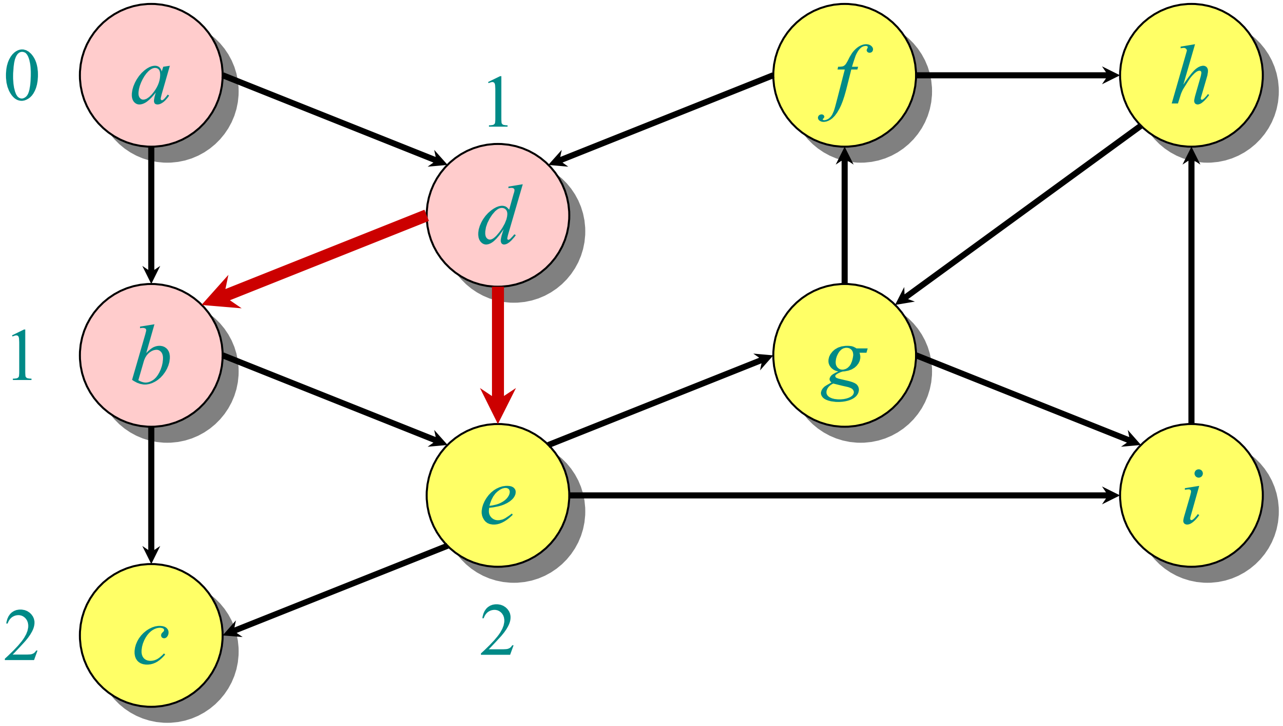


1 2 2

Q: *a* *b* *d* *c* *e*



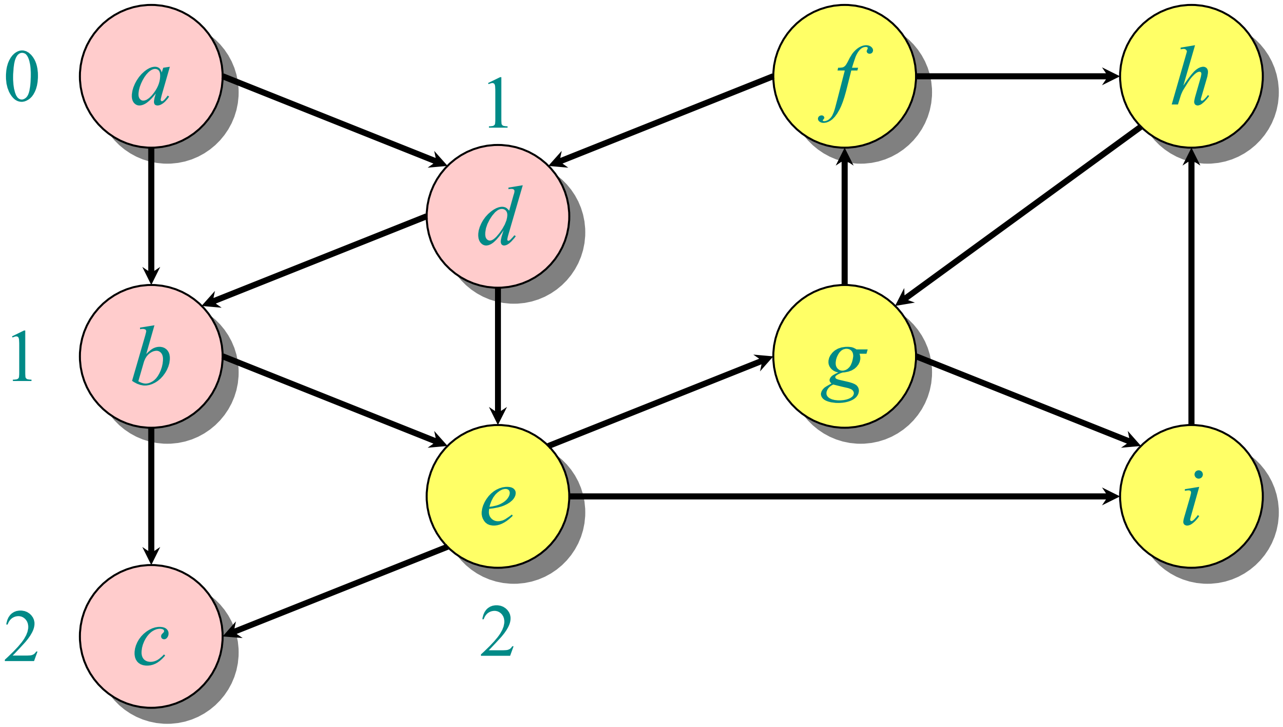
Example of breadth-first search



2 2

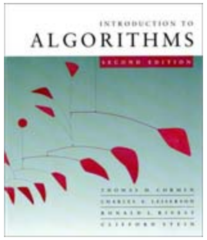
Q: *a* *b* *d* *c* *e*

Example of breadth-first search

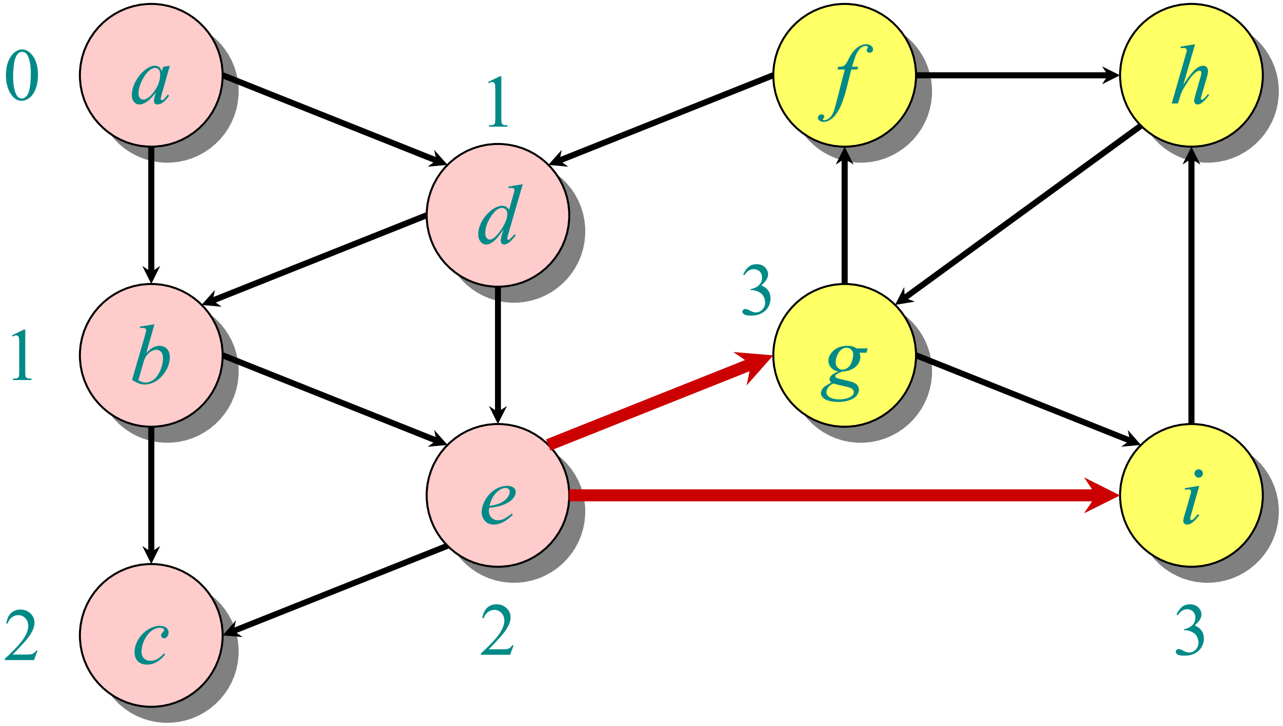


2

Q: *a* *b* *d* *c* *e*

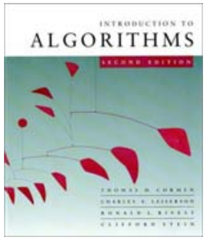


Example of breadth-first search

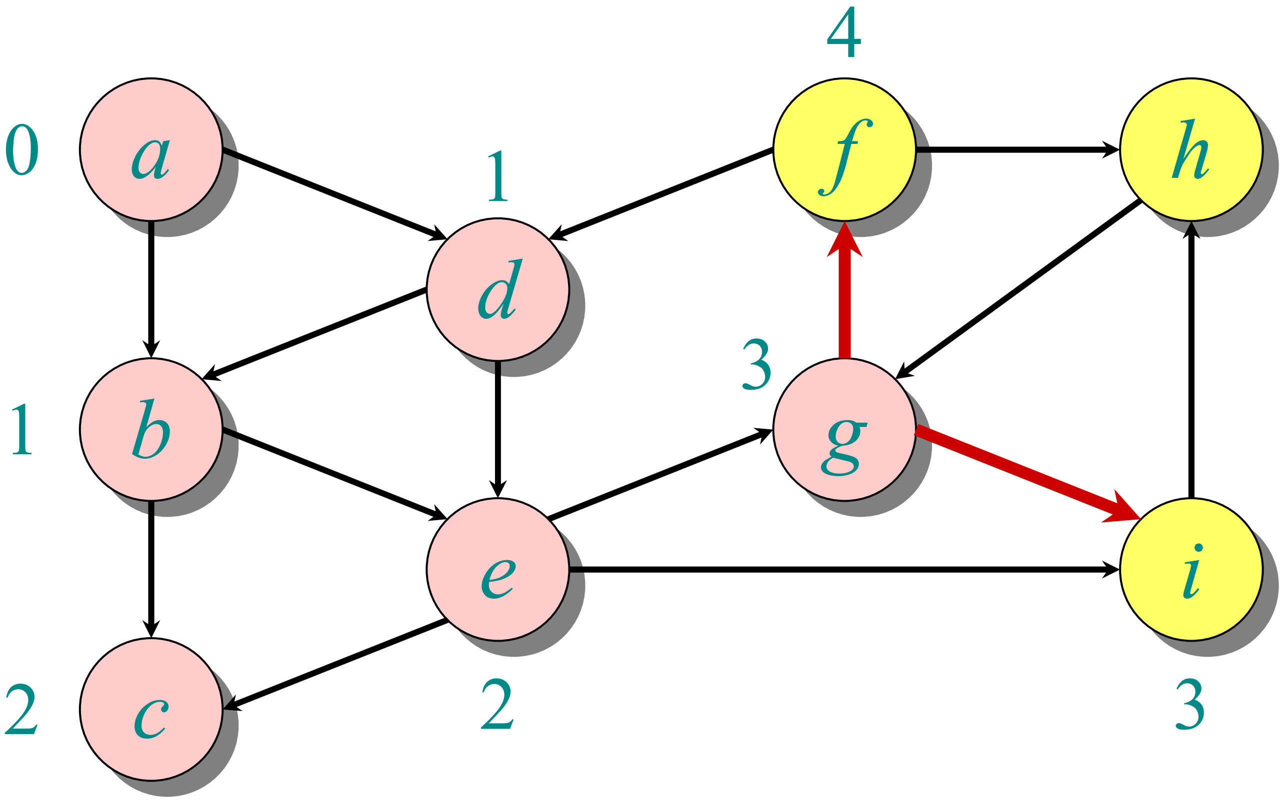


3 3

Q: *a b d c e g i*



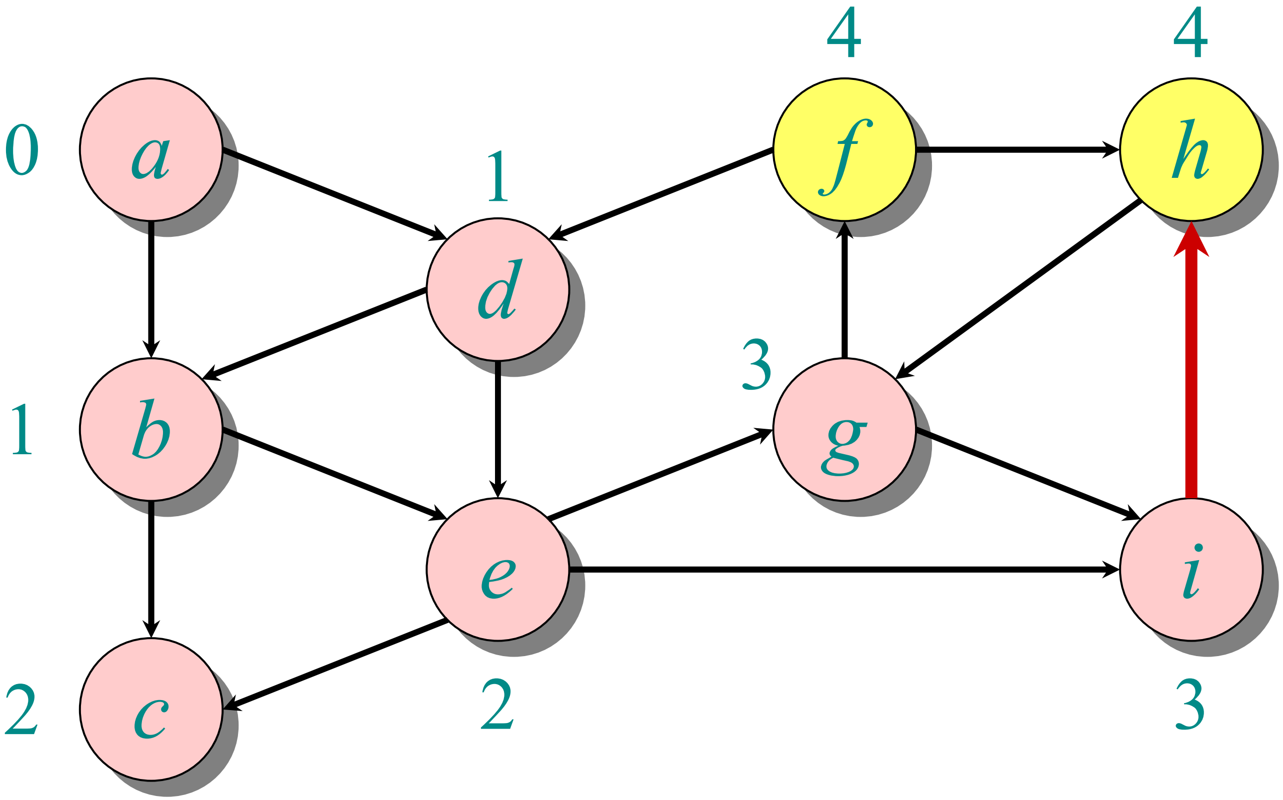
Example of breadth-first search



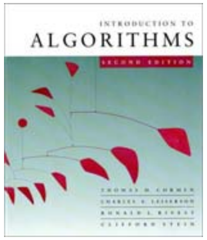
3 4

Q: *a* *b* *d* *c* *e* *g* *i* *f*

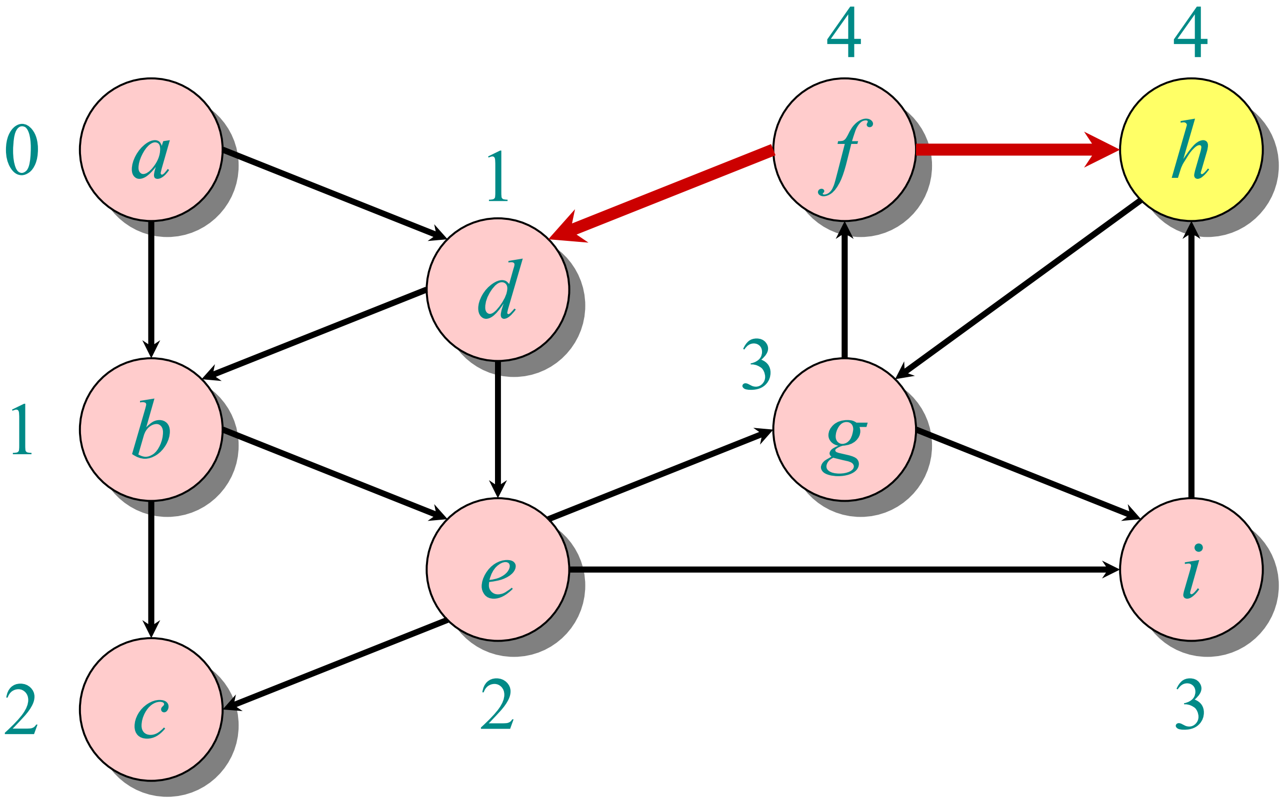
Example of breadth-first search



$\begin{matrix} & & & & & & & & 4 & 4 \\ Q\!: & a & b & d & c & e & g & i & f & h \end{matrix}$



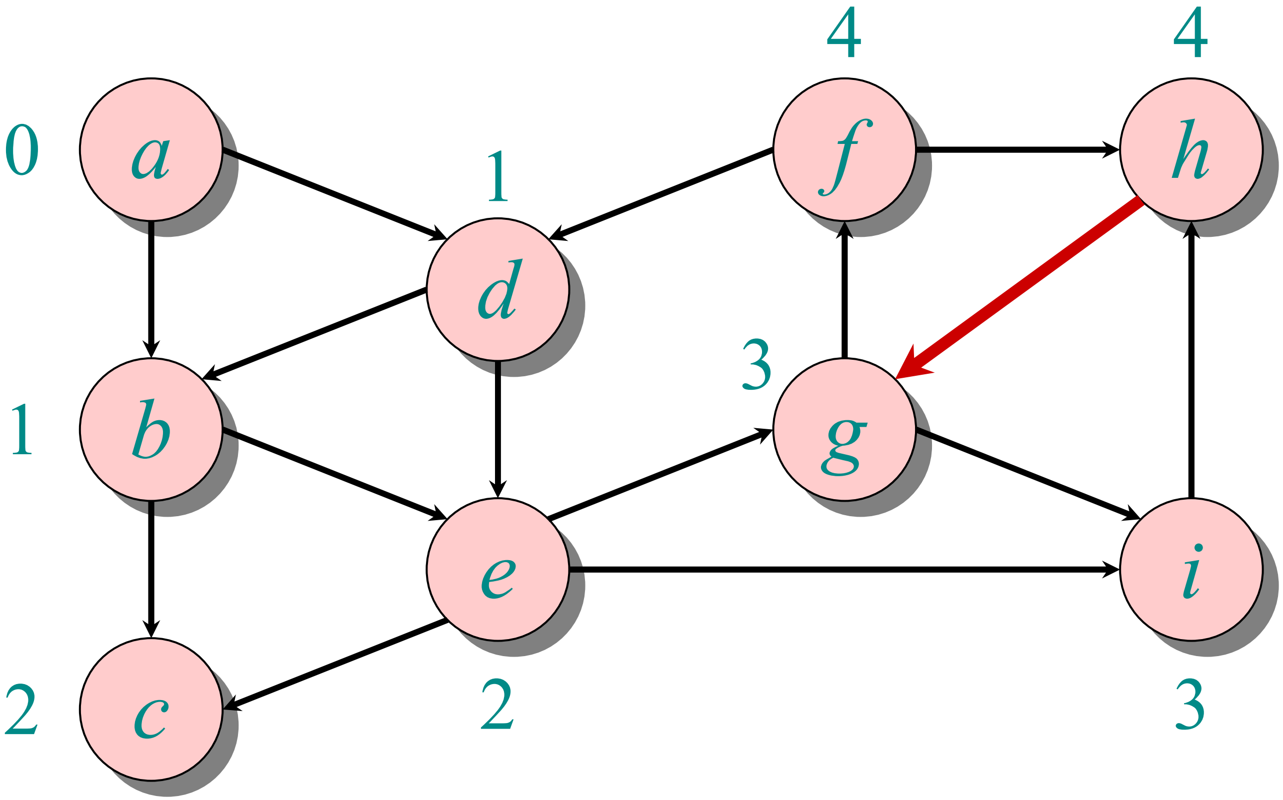
Example of breadth-first search



4

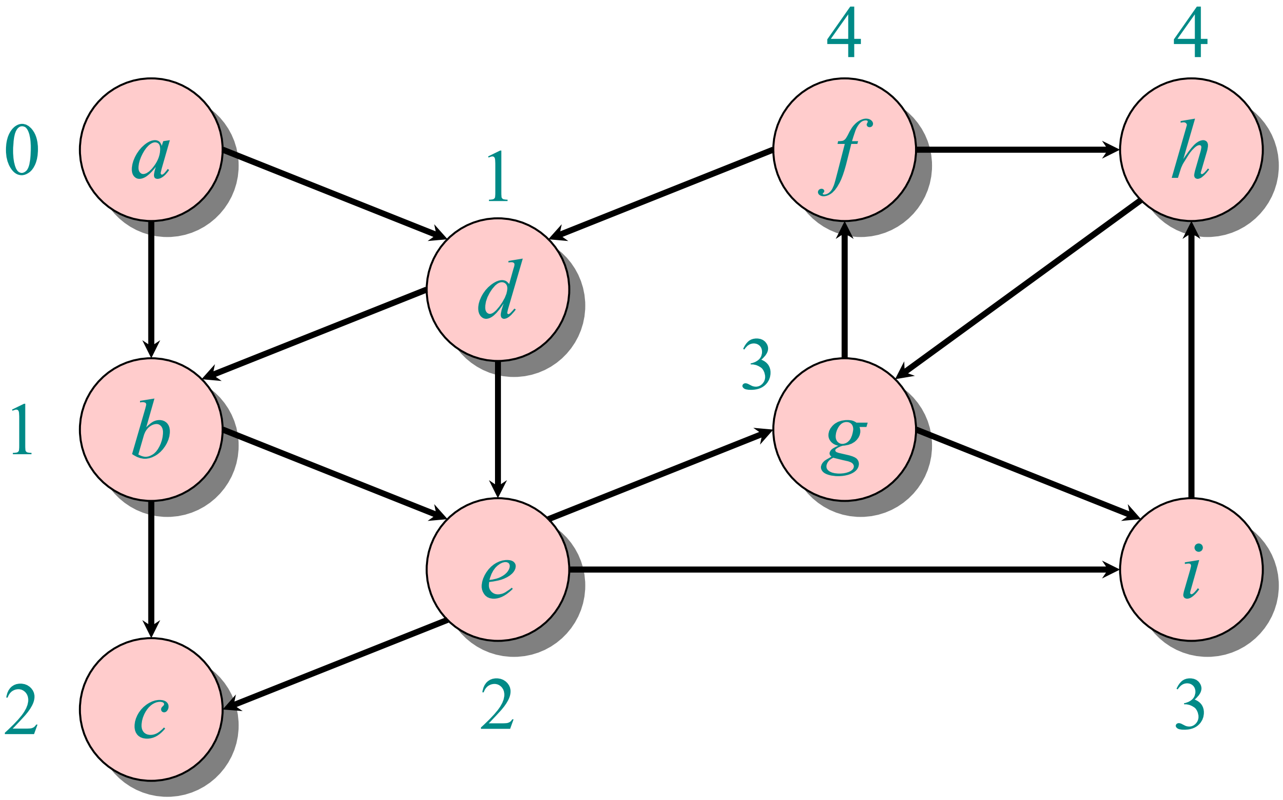
Q: *a b d c e g i f h*

Example of breadth-first search

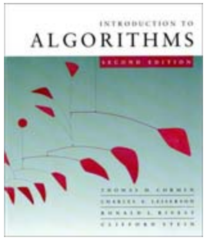


Q: *a b d c e g i f h*

Example of breadth-first search



Q: *a b d c e g i f h*



Correctness of BFS

```
while $Q \neq \emptyset$
    do $u \leftarrow$ DEQUEUE$(Q)$
        for each $v \in Adj[u]$
            do if $d[v] = \infty$
                then $d[v] \leftarrow d[u] + 1$
                    ENQUEUE$(Q, v)$
```

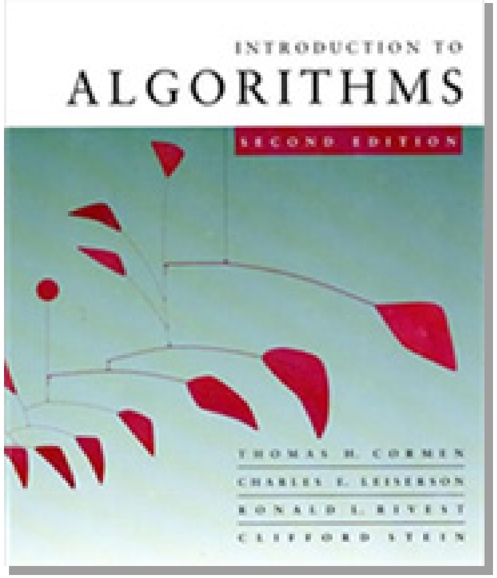
Key idea:

The FIFO Q in breadth-first search mimics the priority queue Q in Dijkstra.

- **Invariant:** v comes after u in Q implies that $d[v] = d[u]$ or $d[v] = d[u] + 1$.

Introduction to Algorithms

6.046J/18.401J

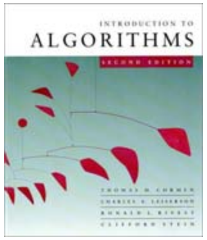


LECTURE 18

Shortest Paths II

- Bellman-Ford algorithm
- Linear programming and difference constraints
- VLSI layout compaction

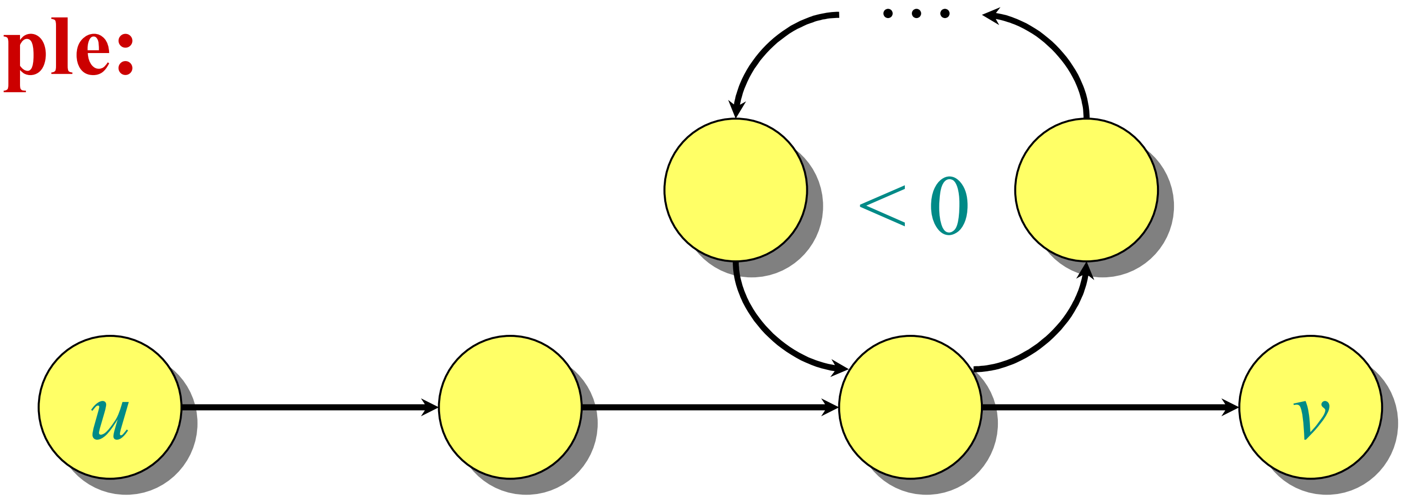
Prof. Erik Demaine

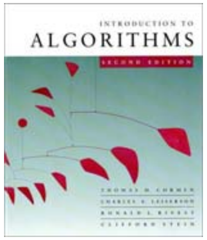


Negative-weight cycles

Recall: If a graph $G = (V, E)$ contains a negative-weight cycle, then some shortest paths may not exist.

Example:

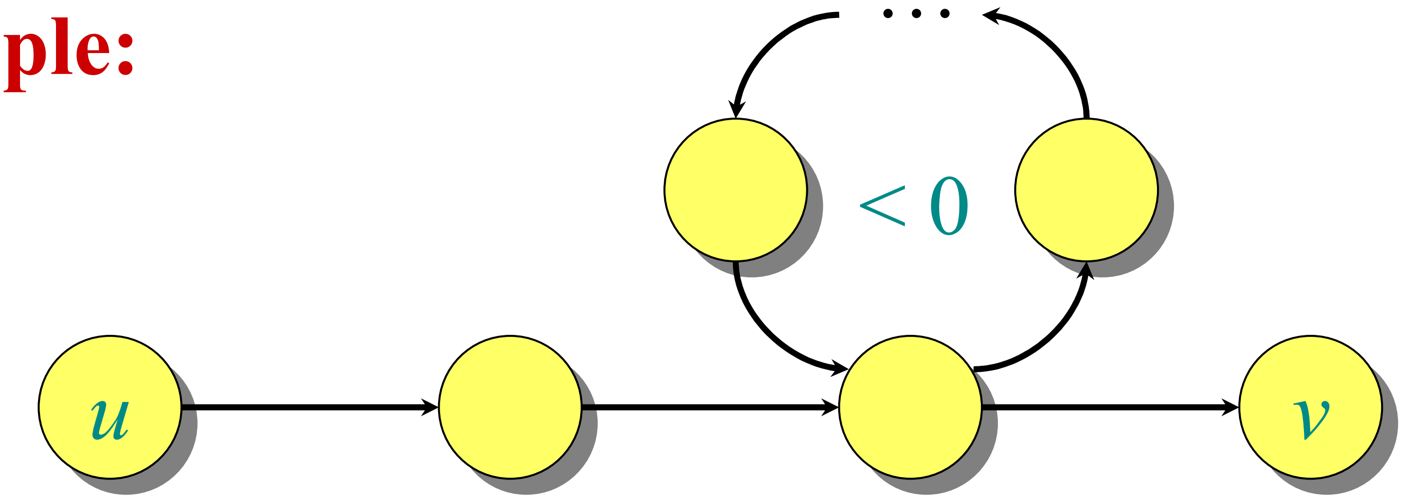




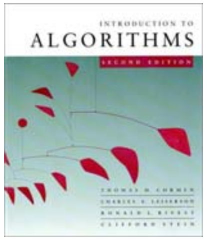
Negative-weight cycles

Recall: If a graph $G = (V, E)$ contains a negative-weight cycle, then some shortest paths may not exist.

Example:



Bellman-Ford algorithm: Finds all shortest-path lengths from a ***source*** $s \in V$ to all $v \in V$ or determines that a negative-weight cycle exists.

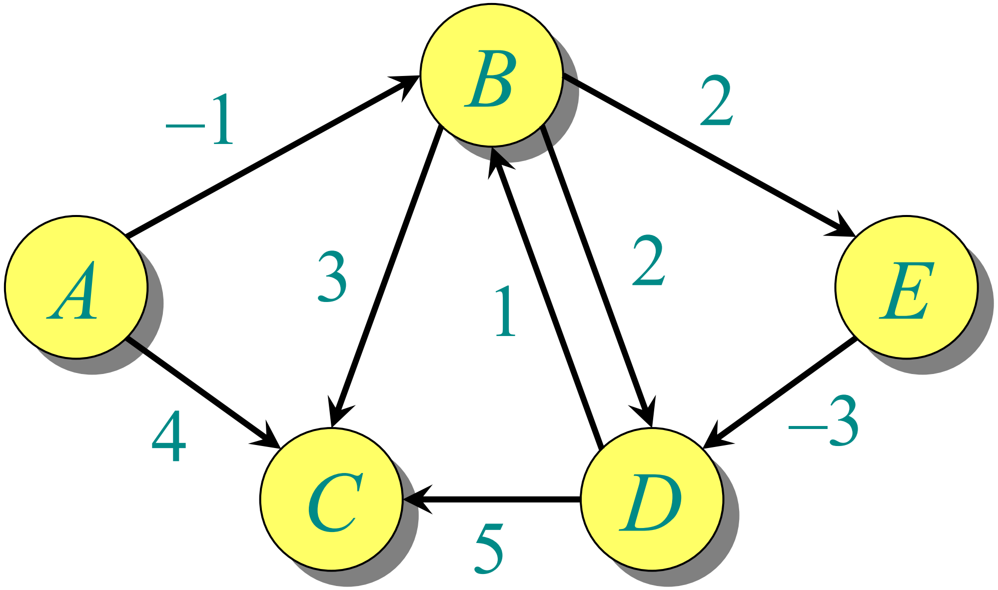


Bellman-Ford algorithm

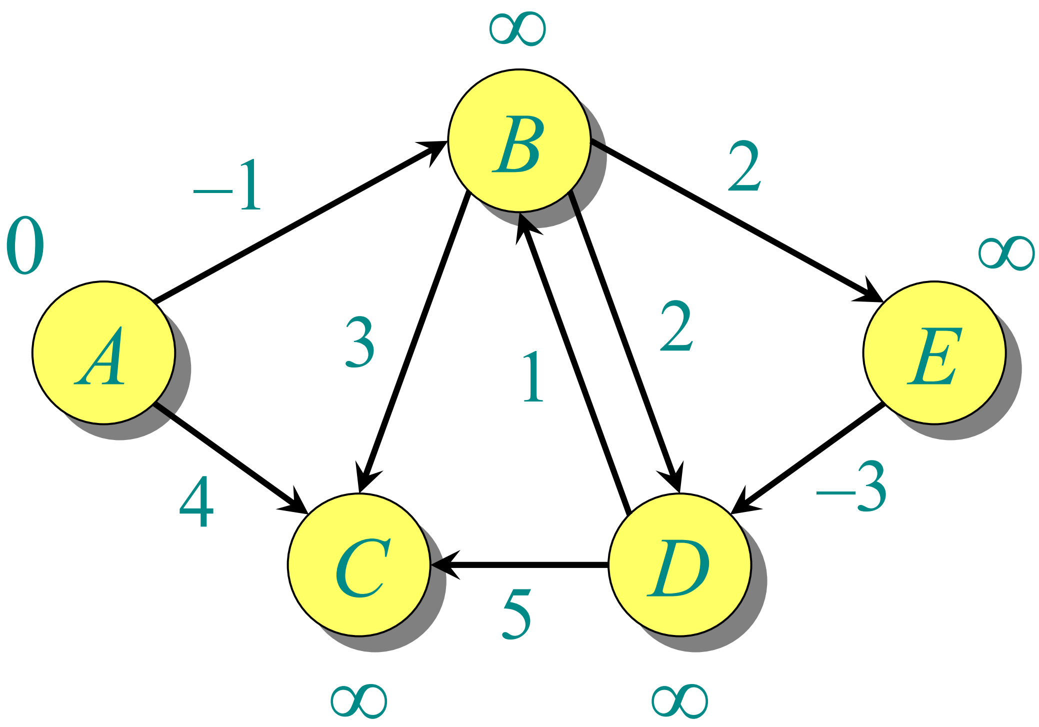
```
$d[s] \leftarrow 0$
for each $v \in V - \{s\}$                     } initialization
    do $d[v] \leftarrow \infty$

for $i \leftarrow 1$ to $|V| - 1$
    do for each edge $(u, v) \in E$
        do if $d[v] > d[u] + w(u, v)$               } relaxation
            then $d[v] \leftarrow d[u] + w(u, v)$   } step

for each edge $(u, v) \in E$
    do if $d[v] > d[u] + w(u, v)$
        then report that a negative-weight cycle exists
```

At the end, $d[v] = \delta(s, v)$, if no negative-weight cycles.
Time = $O(VE)$.

Example of Bellman-Ford

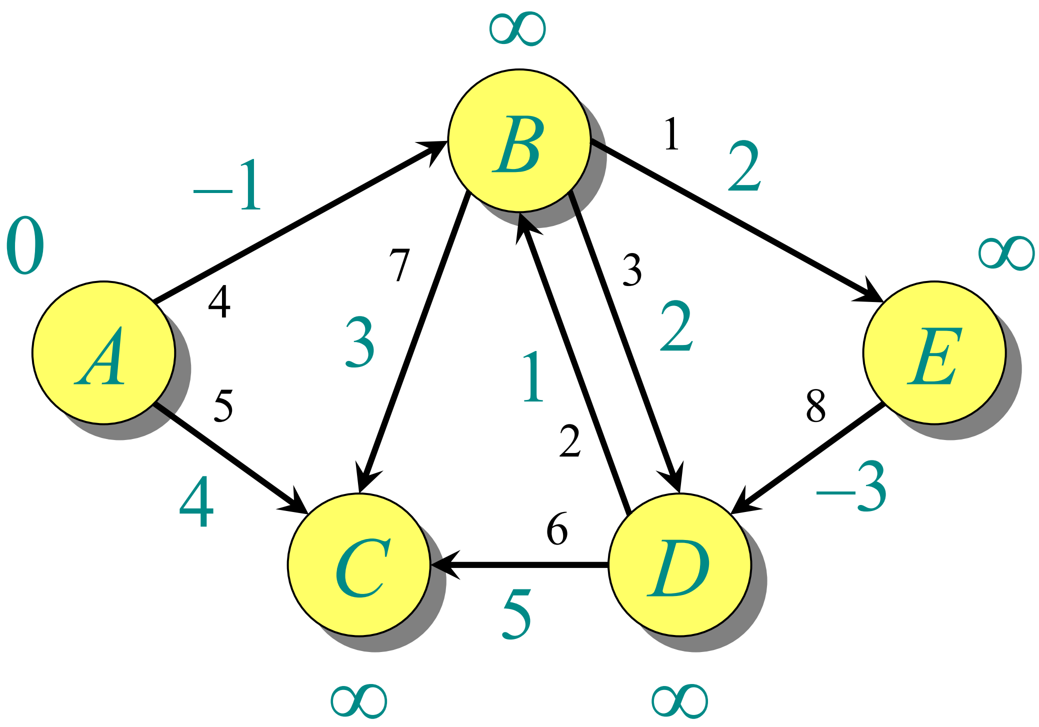


Example of Bellman-Ford

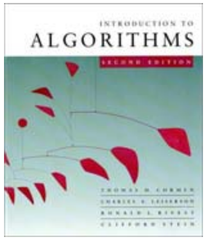


Initialization.

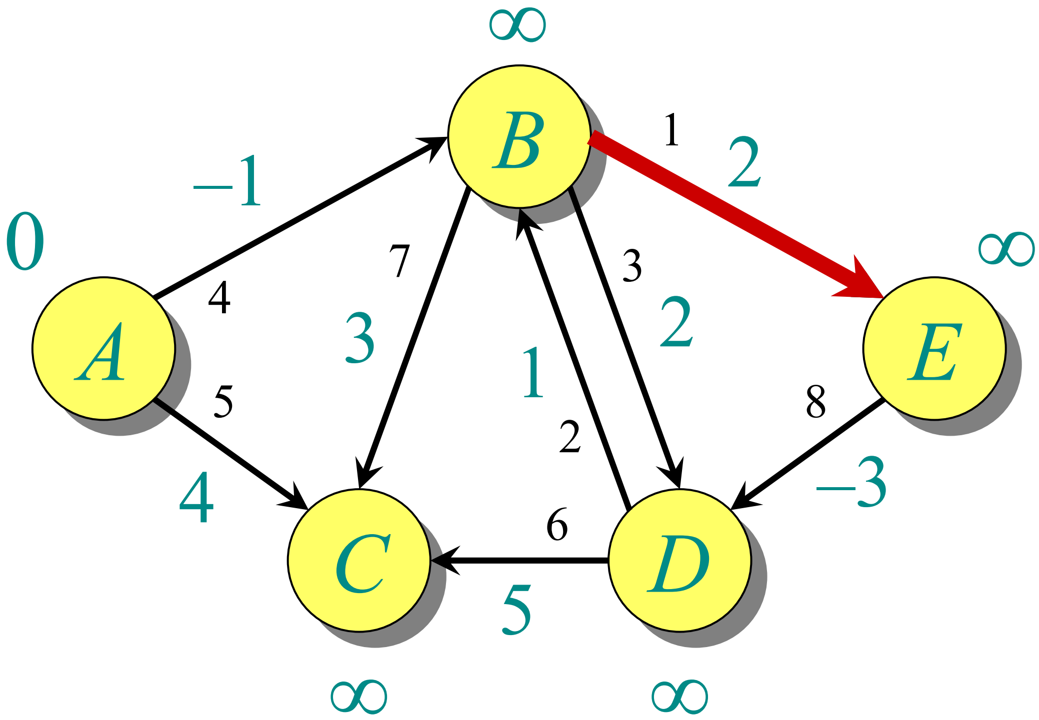
Example of Bellman-Ford

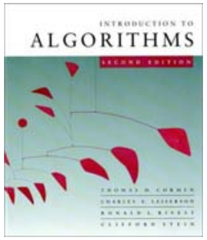


Order of edge relaxation.

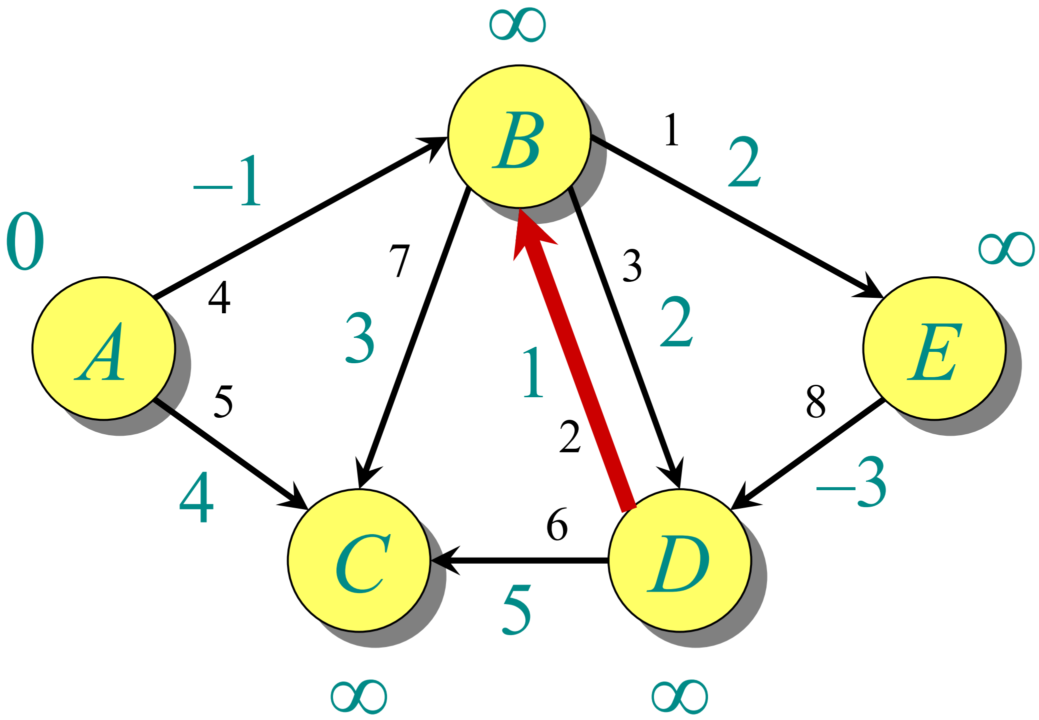


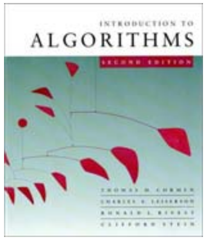
Example of Bellman-Ford



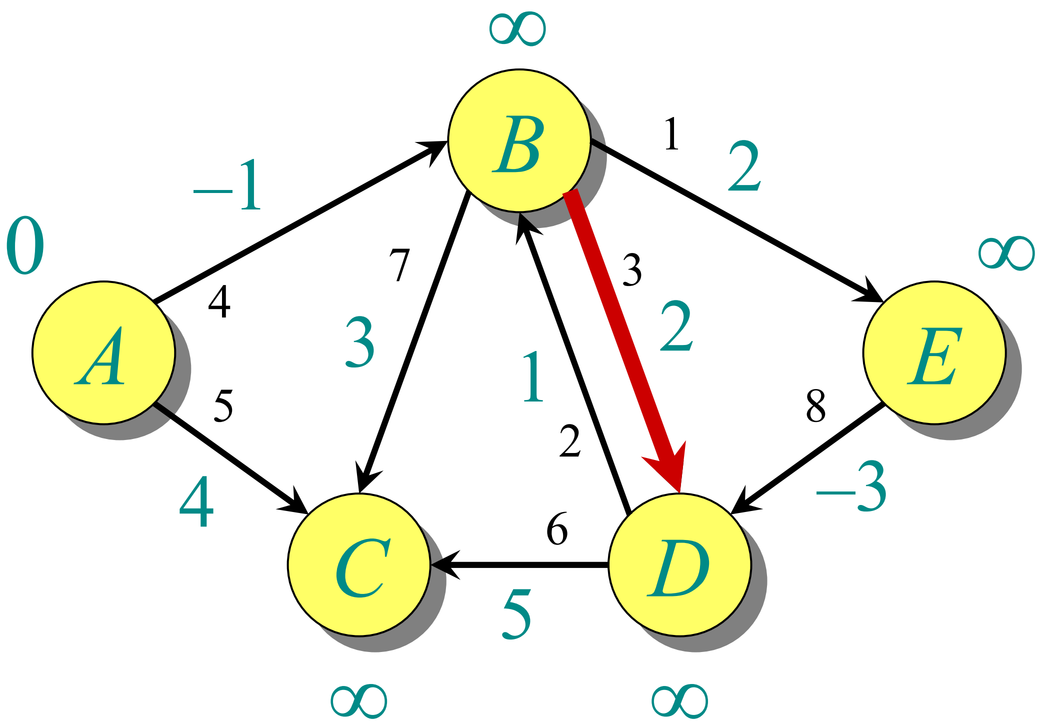


Example of Bellman-Ford

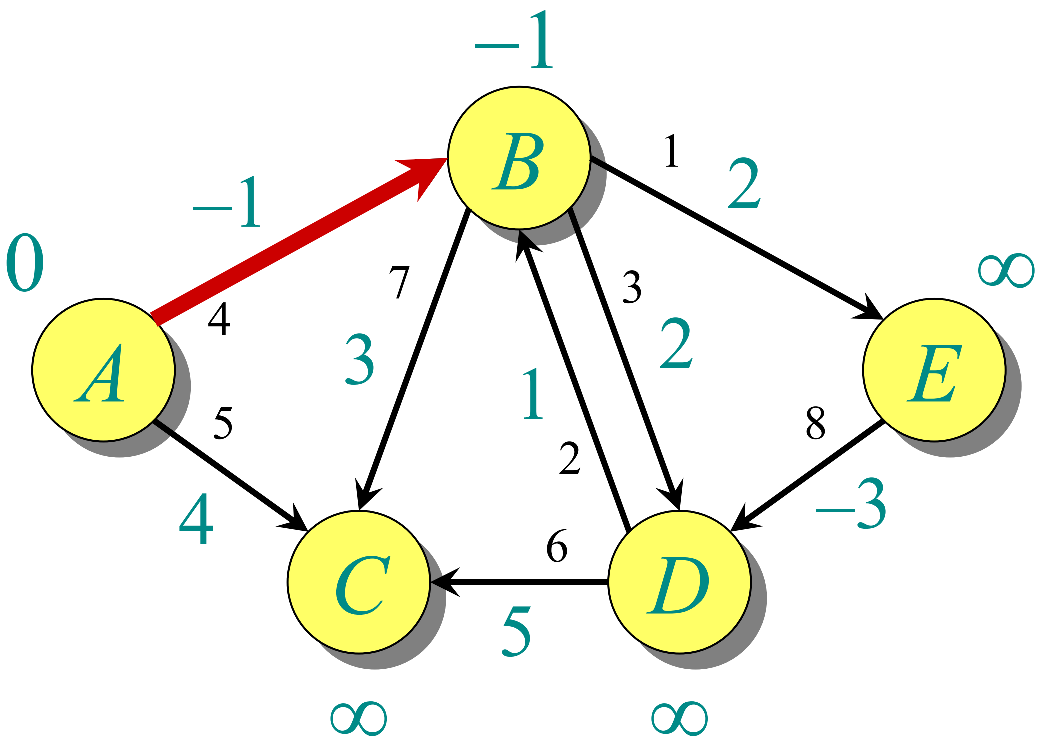


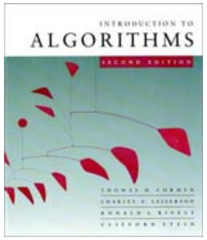


Example of Bellman-Ford

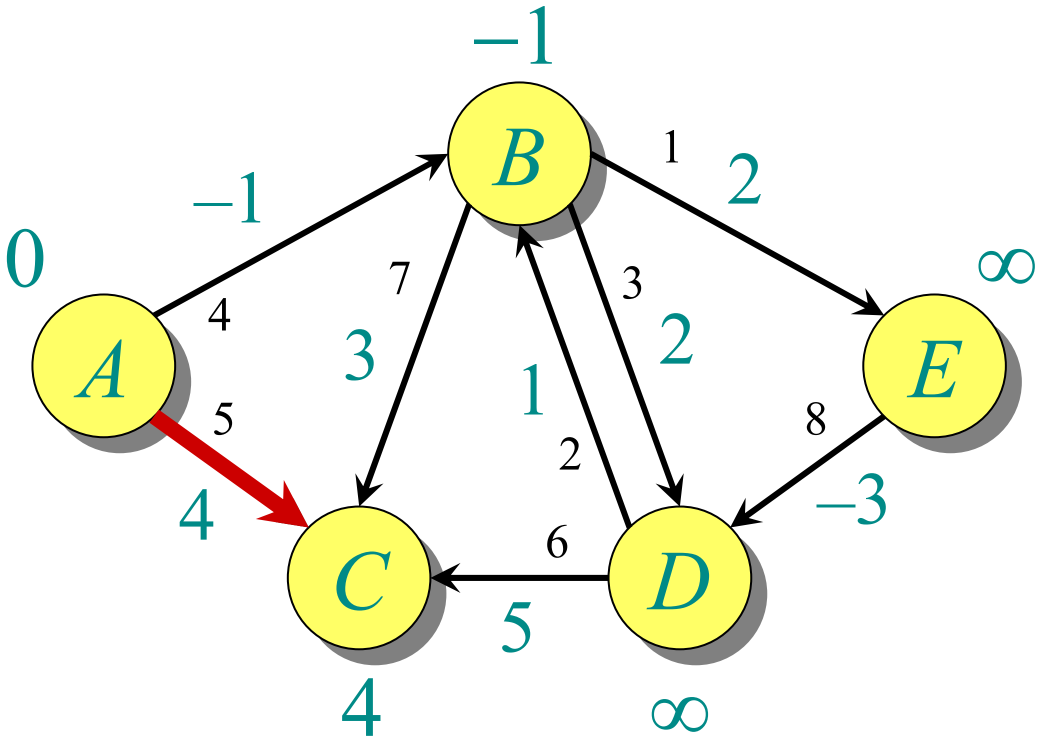


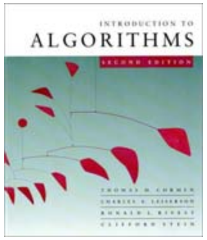
Example of Bellman-Ford



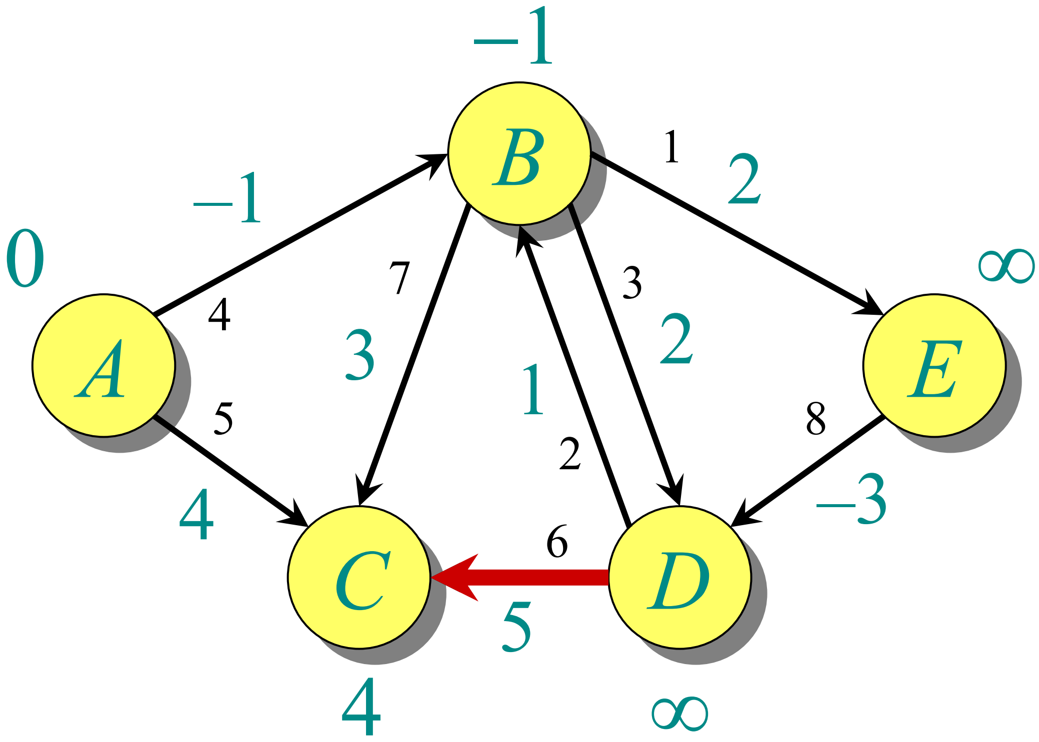


Example of Bellman-Ford

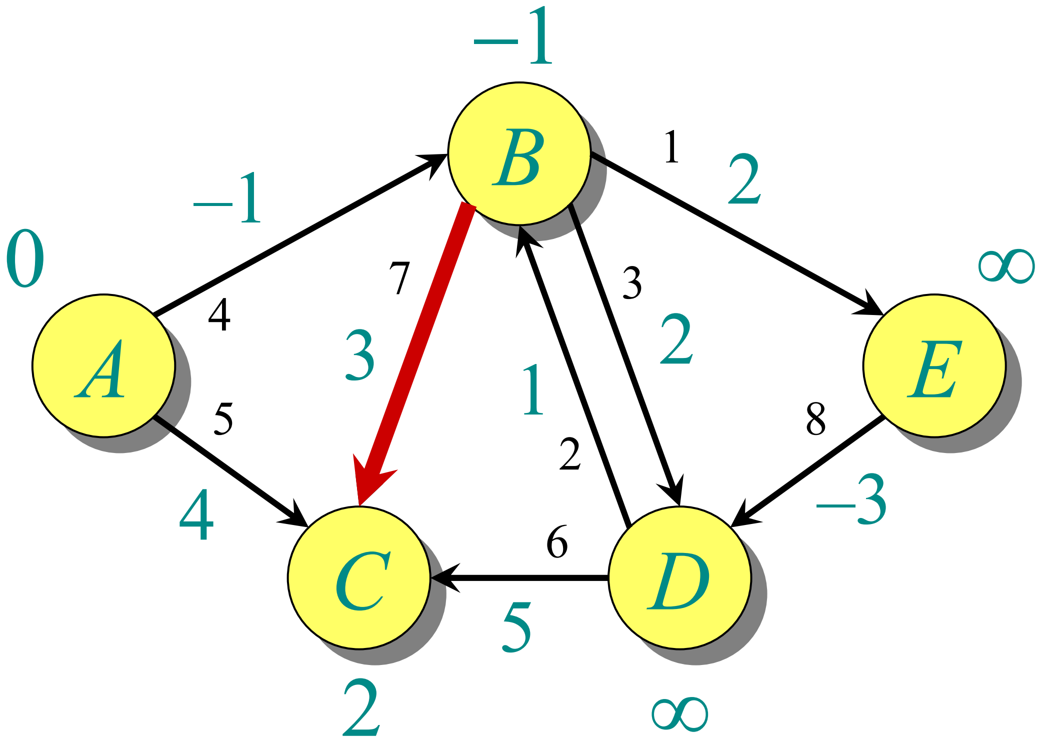


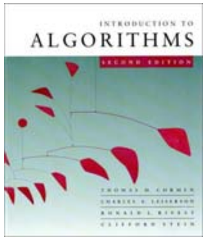


Example of Bellman-Ford

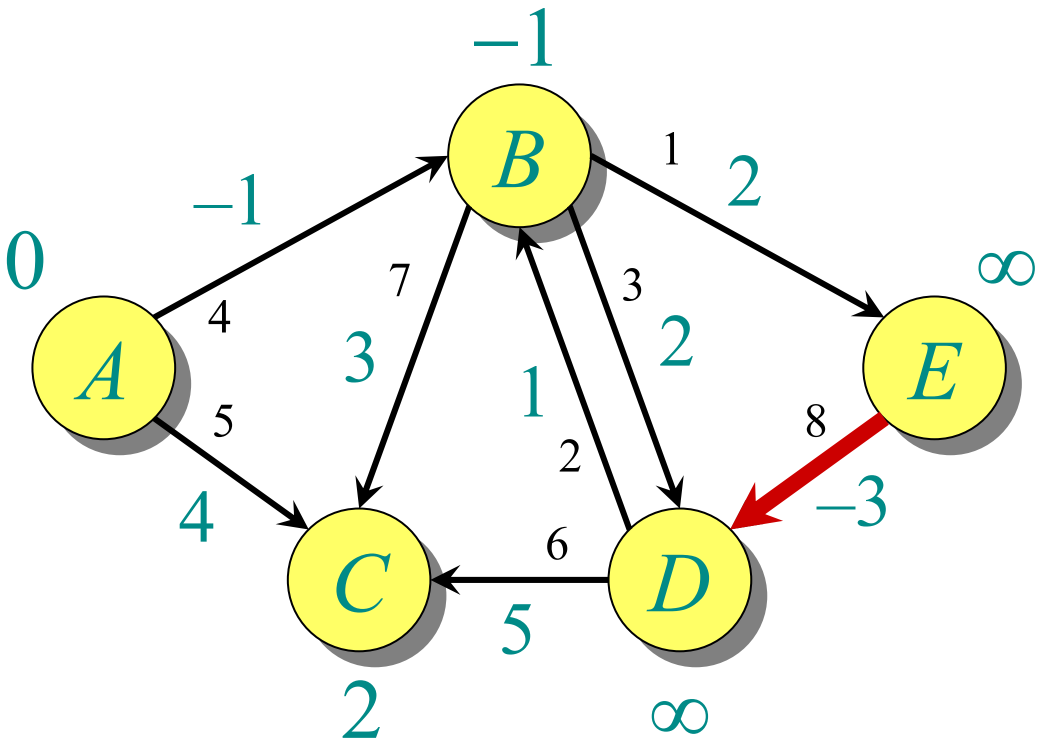


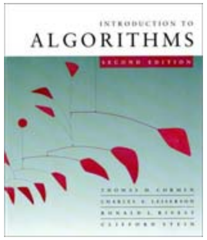
Example of Bellman-Ford



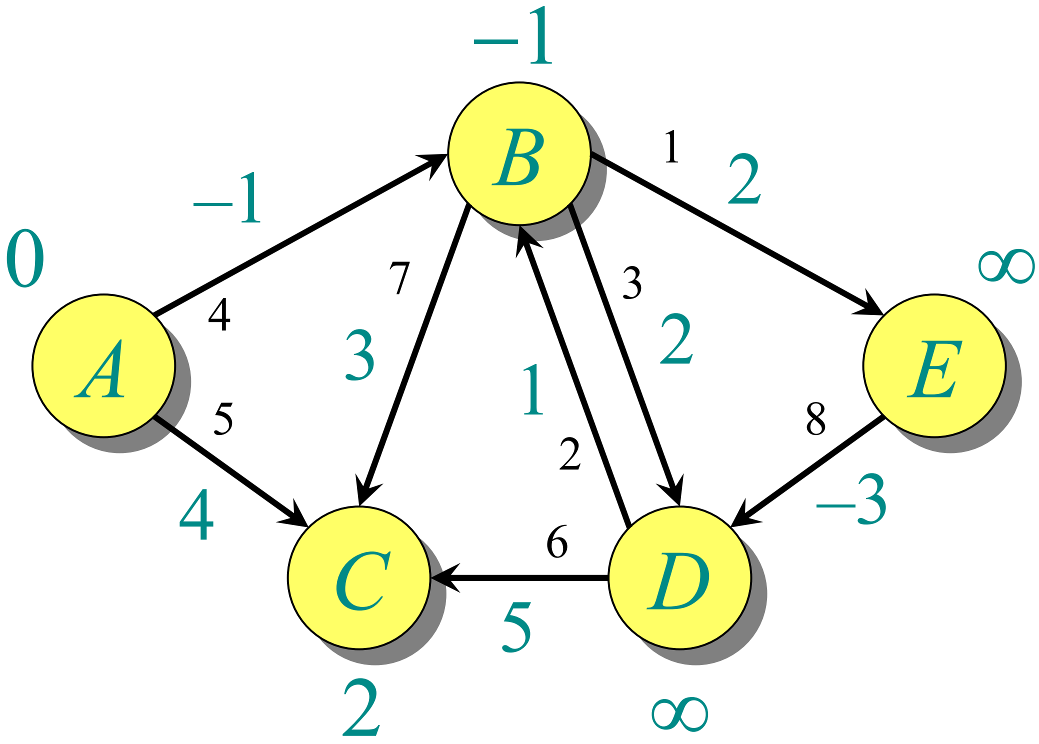


Example of Bellman-Ford

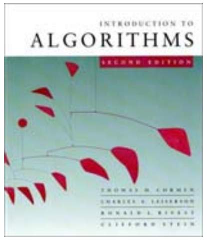




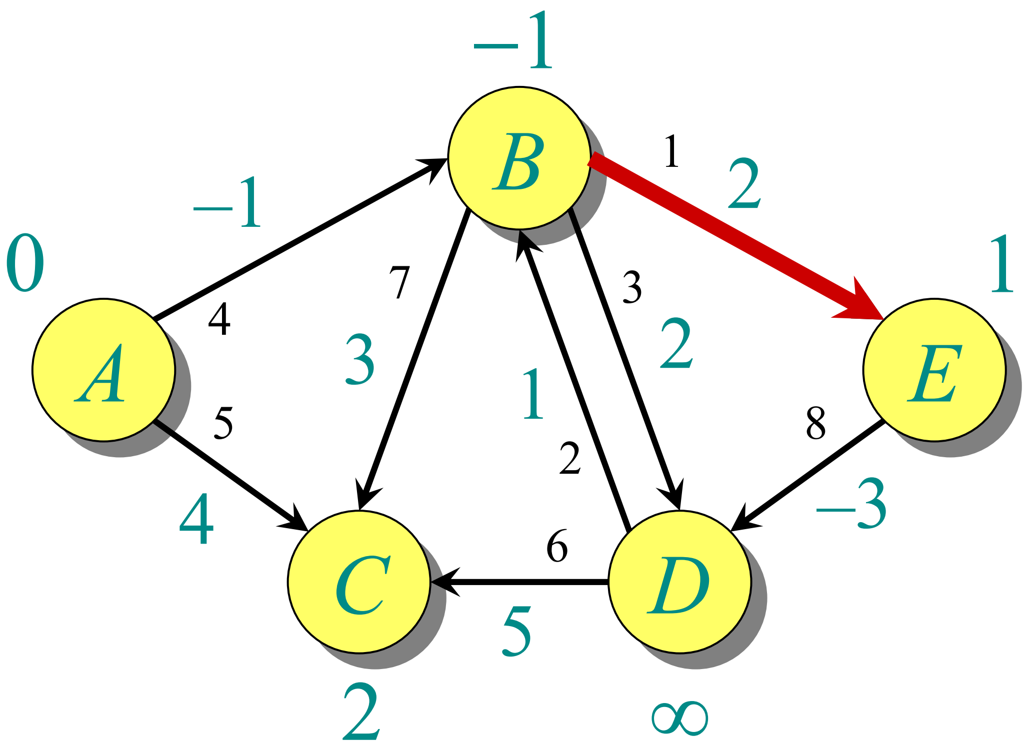
Example of Bellman-Ford



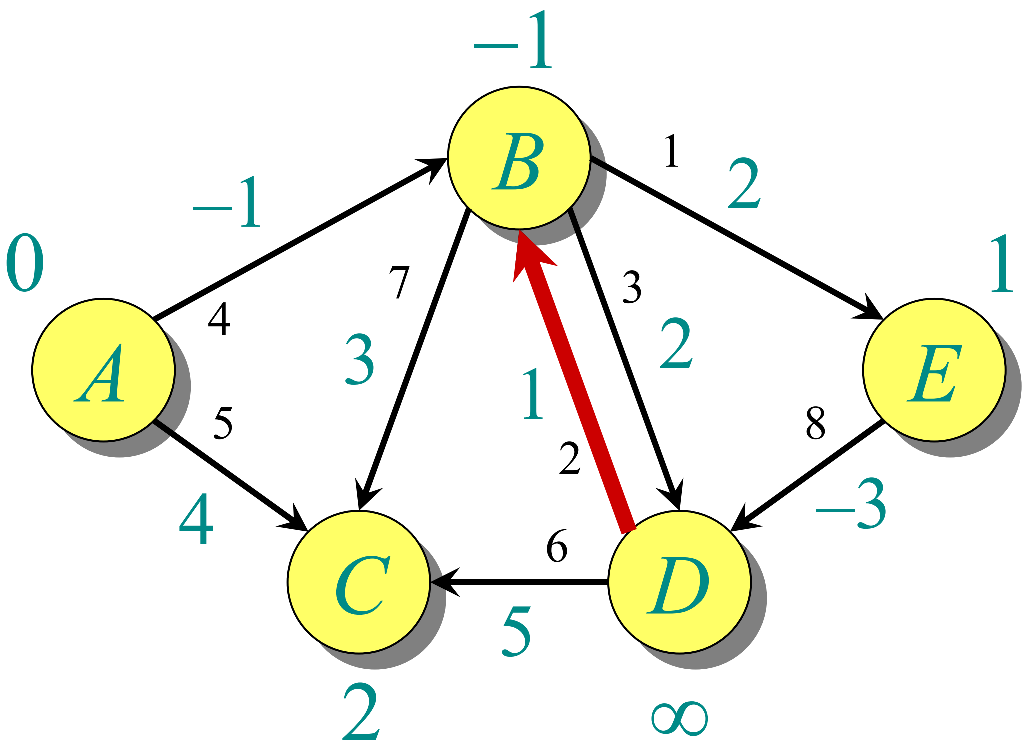
End of pass 1.

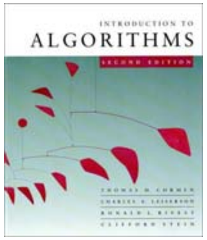


Example of Bellman-Ford

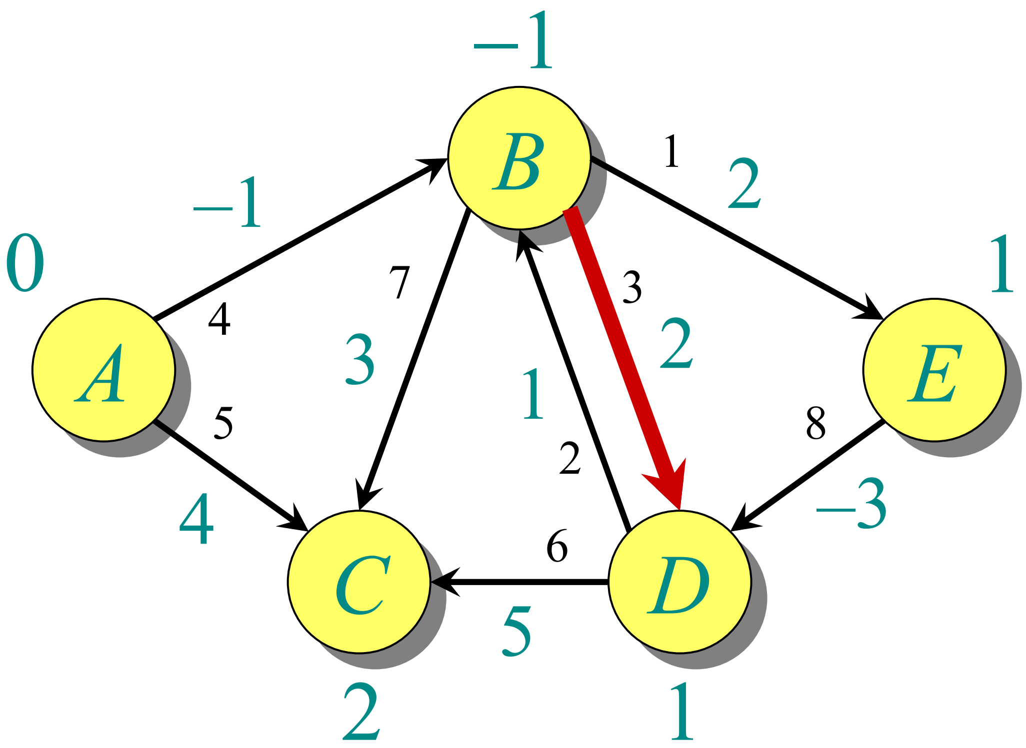


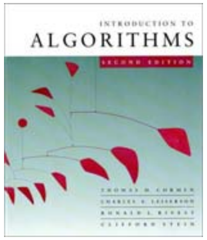
Example of Bellman-Ford



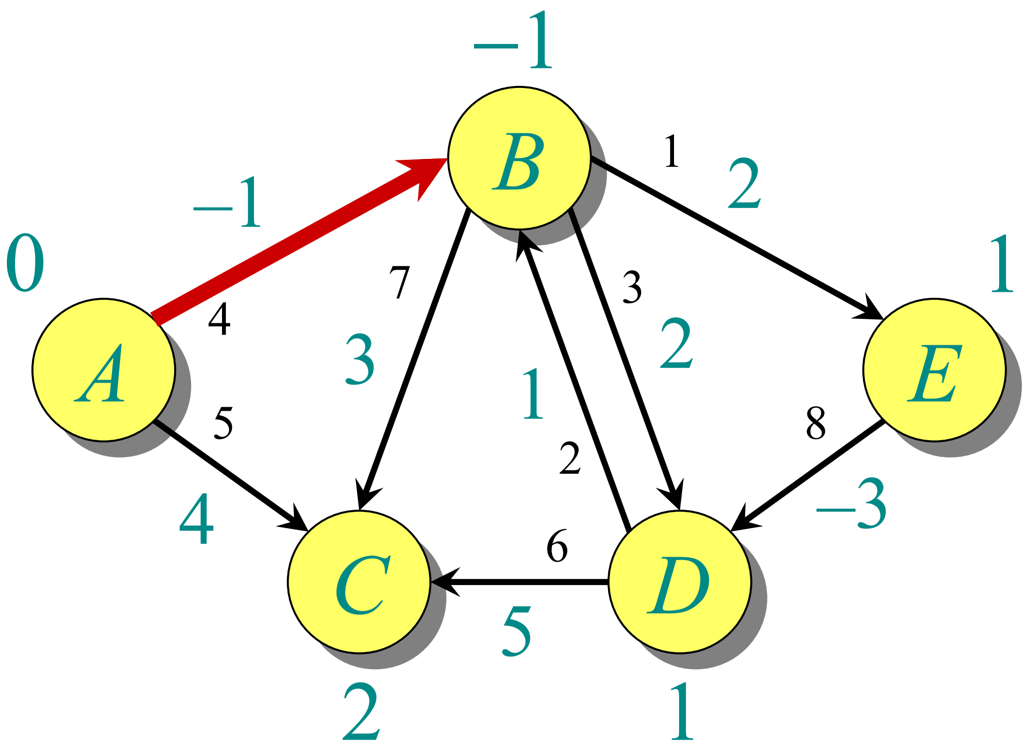


Example of Bellman-Ford

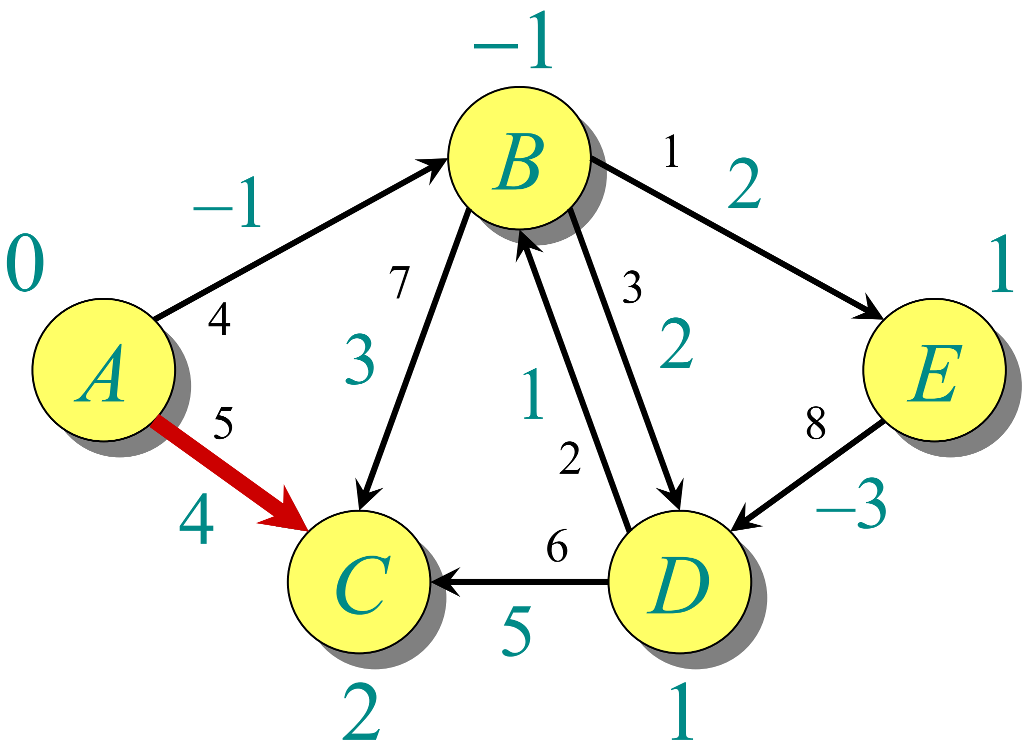


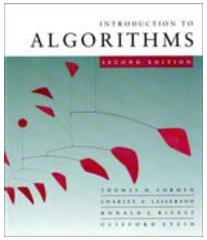


Example of Bellman-Ford

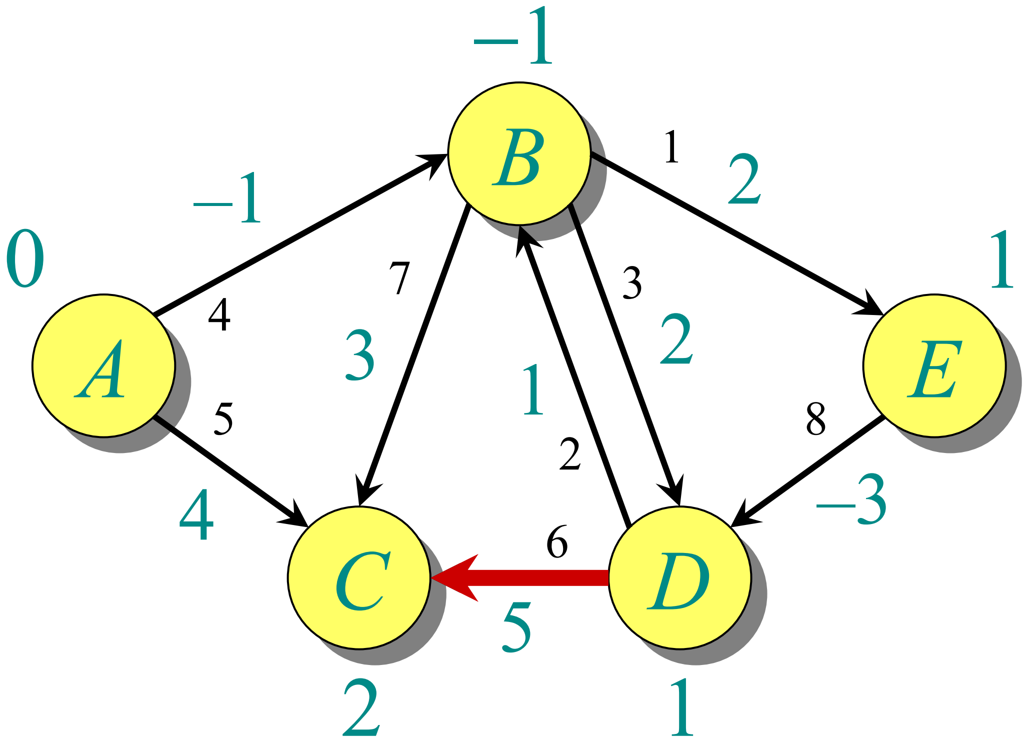


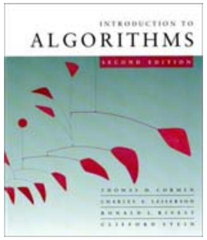
Example of Bellman-Ford



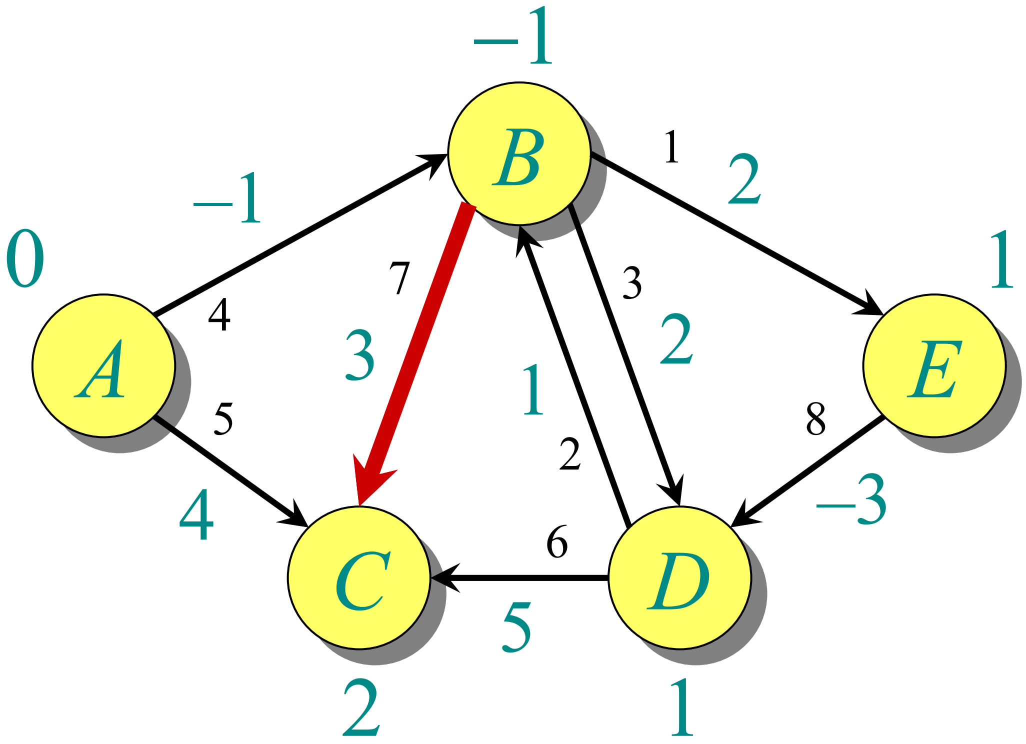


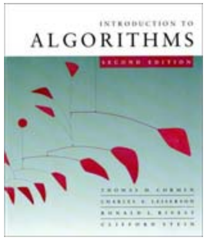
Example of Bellman-Ford



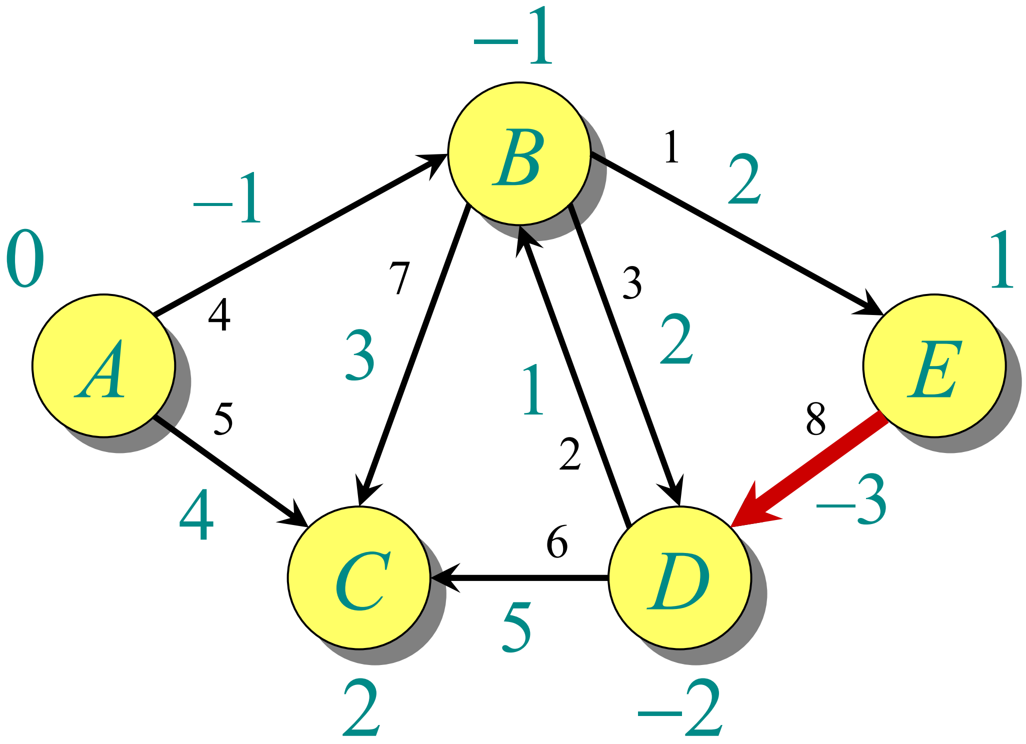


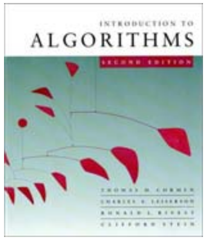
Example of Bellman-Ford



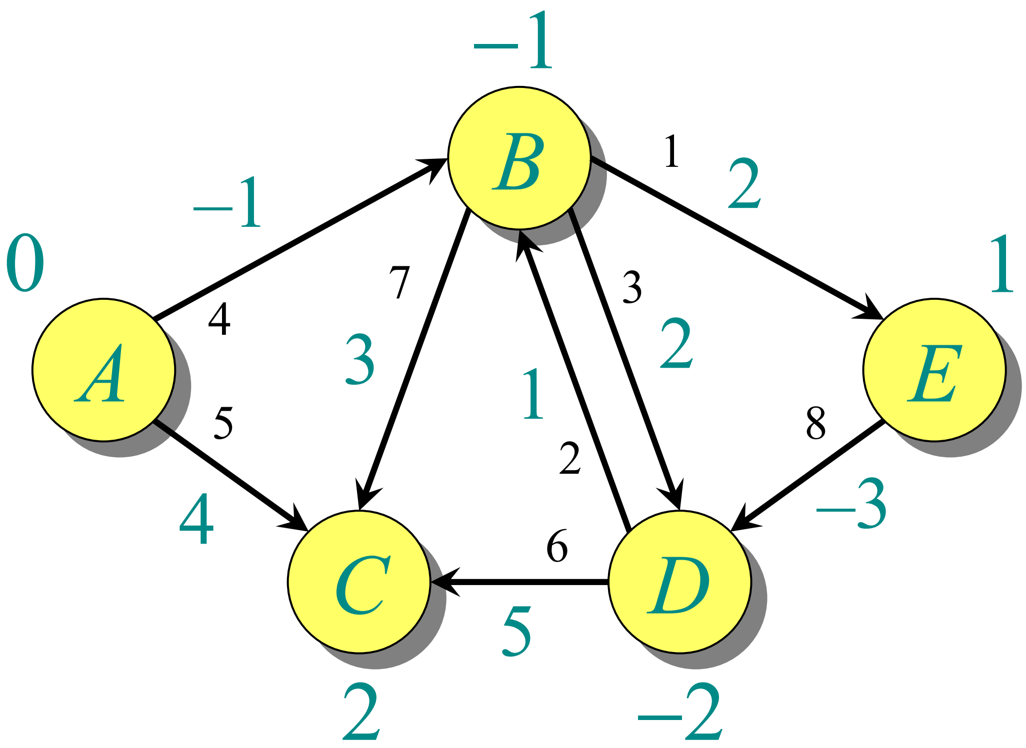


Example of Bellman-Ford

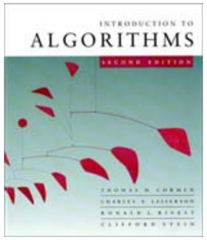




Example of Bellman-Ford

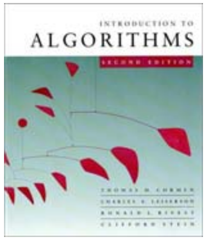


End of pass 2 (and 3 and 4).



Correctness

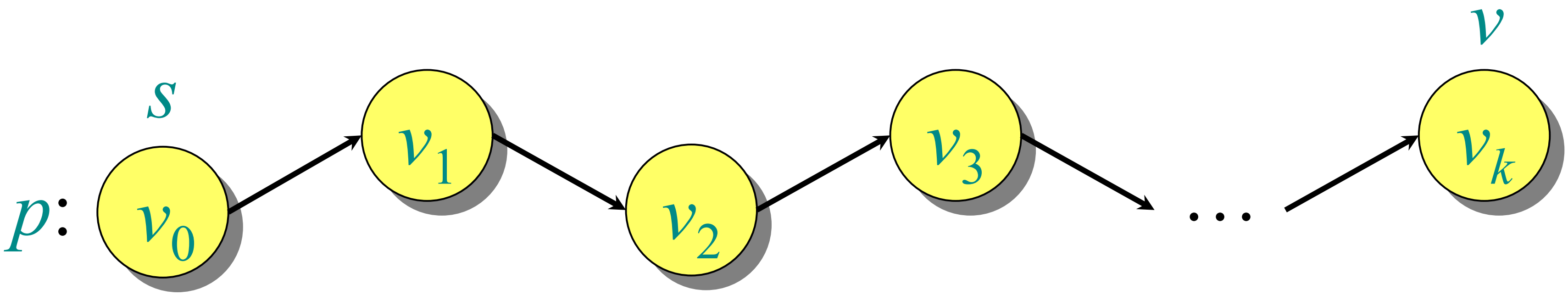
Theorem. If $G = (V, E)$ contains no negative-weight cycles, then after the Bellman-Ford algorithm executes, $d[v] = \delta(s, v)$ for all $v \in V$.



Correctness

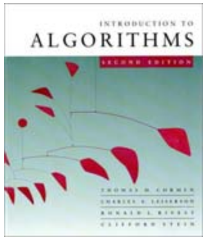
Theorem. If $G = (V, E)$ contains no negative-weight cycles, then after the Bellman-Ford algorithm executes, $d[v] = \delta(s, v)$ for all $v \in V$.

Proof. Let $v \in V$ be any vertex, and consider a shortest path p from s to v with the minimum number of edges.

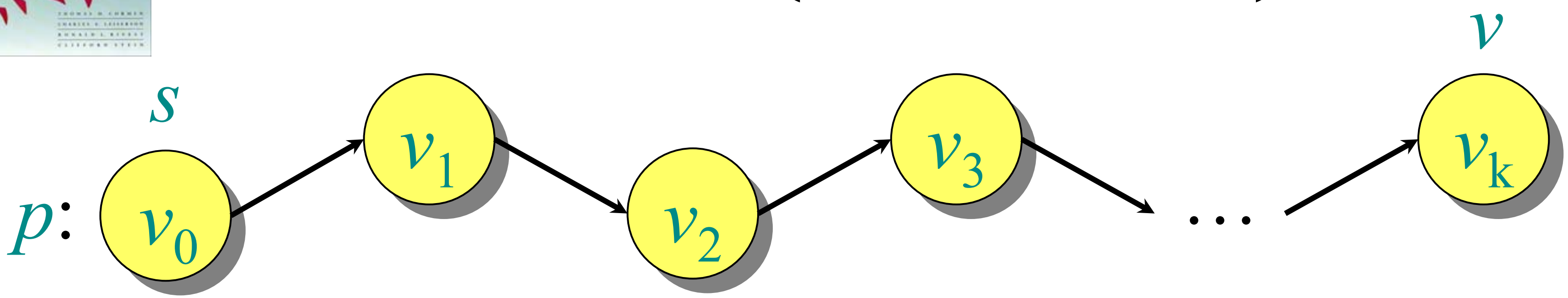


Since p is a shortest path, we have

$$\delta(s, v_i) = \delta(s, v_{i-1}) + w(v_{i-1}, v_i) \; .$$



Correctness (continued)



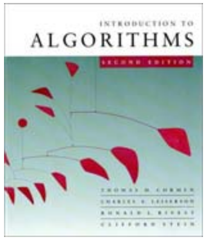
Initially, $d[v_0] = 0 = \delta(s, v_0)$, and $d[v_0]$ is unchanged by subsequent relaxations (because of the lemma from Lecture 14 that $d[v] \geq \delta(s, v)$).

- After 1 pass through E, we have $d[v_1] = \delta(s, v_1)$.
- After 2 passes through E, we have $d[v_2] = \delta(s, v_2)$.

⋮

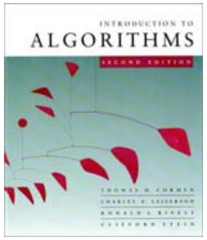
- After k passes through E, we have $d[v_k] = \delta(s, v_k)$.

Since G contains no negative-weight cycles, p is simple. Longest simple path has $\leq |V| - 1$ edges. □



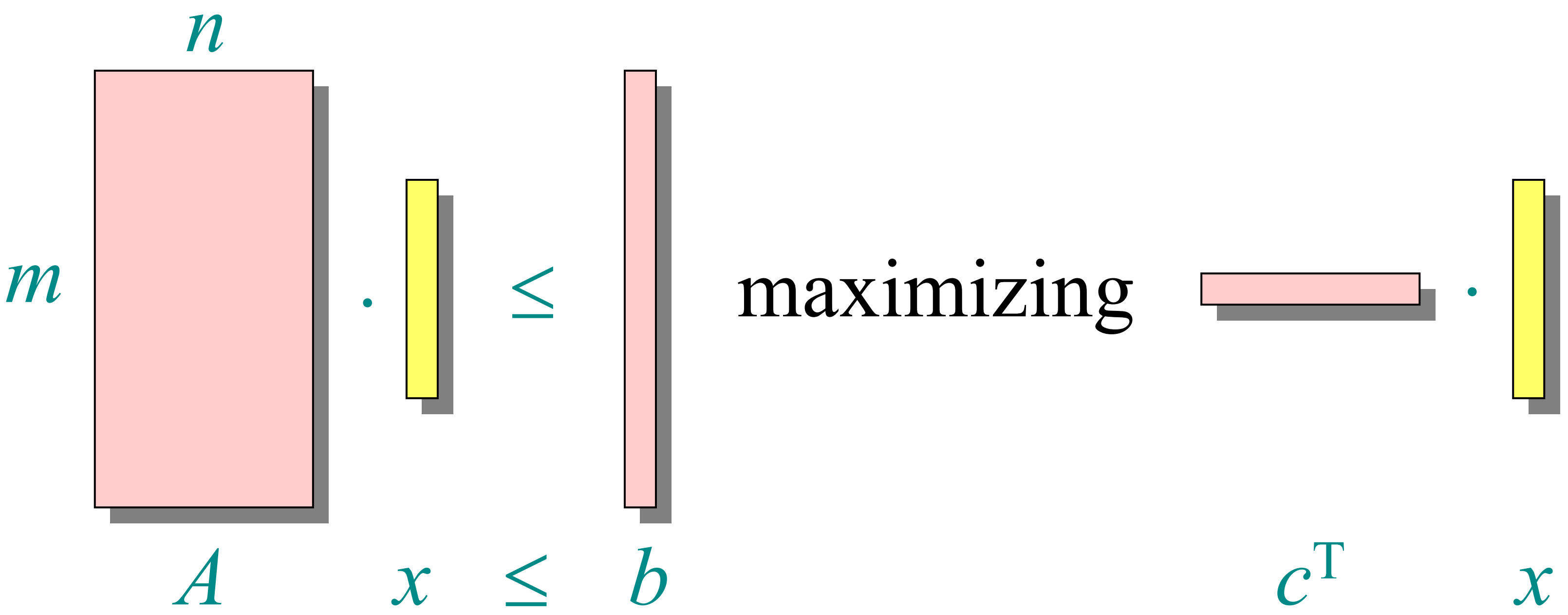
Detection of negative-weight cycles

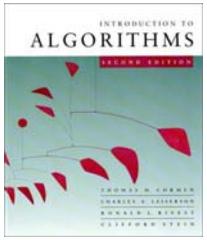
Corollary. If a value $d[v]$ fails to converge after $|V| - 1$ passes, there exists a negative-weight cycle in G reachable from s. □



Linear programming

Let A be an $m \times n$ matrix, b be an m-vector, and c be an n-vector. Find an n-vector x that maximizes $c^{\mathrm{T}}x$ subject to $Ax \le b$, or determine that no such solution exists.

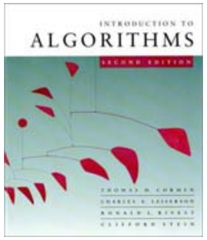




Linear-programming algorithms

Algorithms for the general problem

- Simplex methods — practical, but worst-case exponential time.
- Interior-point methods — polynomial time and competes with simplex.



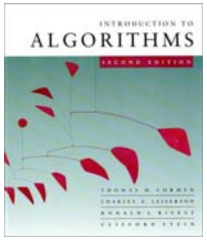
Linear-programming algorithms

Algorithms for the general problem

- Simplex methods — practical, but worst-case exponential time.
- Interior-point methods — polynomial time and competes with simplex.

Feasibility problem: No optimization criterion. Just find x such that $Ax \le b$.

- In general, just as hard as ordinary LP.

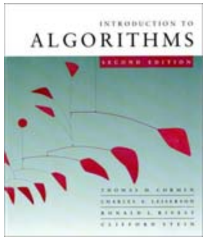


Solving a system of difference constraints

Linear programming where each row of A contains exactly one 1, one –1, and the rest 0's.

Example:

$$\left.\begin{aligned} x_1 - x_2 &\le 3 \\ x_2 - x_3 &\le -2 \\ x_1 - x_3 &\le 2 \end{aligned}\right\} \quad x_j - x_i \le w_{ij}$$



Solving a system of difference constraints

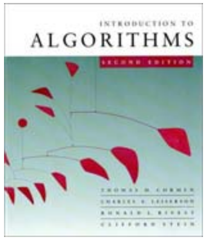
Linear programming where each row of A contains exactly one 1, one –1, and the rest 0's.

Example:

$$\left.\begin{aligned} x_1 - x_2 &\le 3 \\ x_2 - x_3 &\le -2 \\ x_1 - x_3 &\le 2 \end{aligned}\right\} \quad x_j - x_i \le w_{ij}$$

Solution:

$$\begin{aligned} x_1 &= 3 \\ x_2 &= 0 \\ x_3 &= 2 \end{aligned}$$



Solving a system of difference constraints

Linear programming where each row of A contains exactly one 1, one -1, and the rest 0’s.

Example:

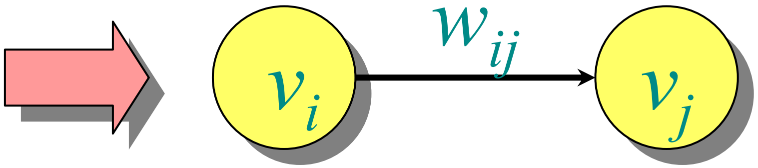
$$\left.\begin{aligned} x_1 - x_2 &\le 3 \\ x_2 - x_3 &\le -2 \\ x_1 - x_3 &\le 2 \end{aligned}\right\} \quad x_j - x_i \le w_{ij}$$

Solution:

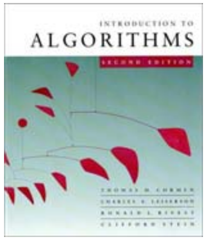
$$\begin{aligned} x_1 &= 3 \\ x_2 &= 0 \\ x_3 &= 2 \end{aligned}$$

Constraint graph:

$x_j - x_i \le w_{ij}$

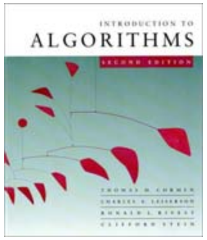


(The “A” matrix has dimensions $|E| \times |V|$.)



Unsatisfiable constraints

Theorem. If the constraint graph contains a negative-weight cycle, then the system of differences is unsatisfiable.

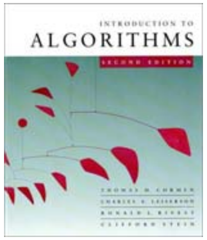


Unsatisfiable constraints

Theorem. If the constraint graph contains a negative-weight cycle, then the system of differences is unsatisfiable.

Proof. Suppose that the negative-weight cycle is $v_1 \to v_2 \to \cdots \to v_k \to v_1$. Then, we have

$$\begin{aligned} x_2 - x_1 &\le w_{12} \\ x_3 - x_2 &\le w_{23} \\ &\vdots \\ x_k - x_{k-1} &\le w_{k-1,\,k} \\ x_1 - x_k &\le w_{k1} \end{aligned}$$



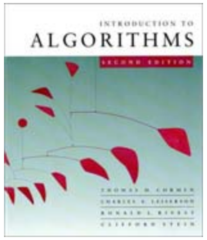
Unsatisfiable constraints

Theorem. If the constraint graph contains a negative-weight cycle, then the system of differences is unsatisfiable.

Proof. Suppose that the negative-weight cycle is $v_1 \to v_2 \to \cdots \to v_k \to v_1$. Then, we have

$$\begin{array}{rcl} x_2 - x_1 & \le & w_{12} \\ x_3 - x_2 & \le & w_{23} \\ & \vdots & \\ x_k - x_{k-1} & \le & w_{k-1,\,k} \\ x_1 - x_k & \le & w_{k1} \\ \hline 0 & \le & \text{weight of cycle} \\ & < & 0 \end{array}$$

Therefore, no values for the x_i can satisfy the constraints. □



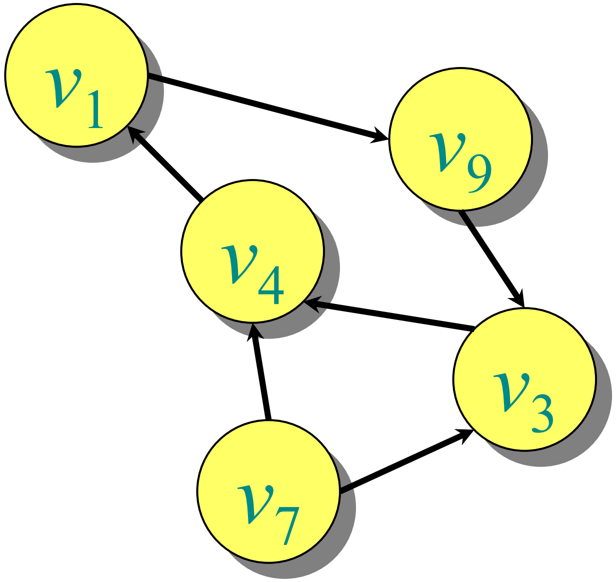
Satisfying the constraints

Theorem. Suppose no negative-weight cycle exists in the constraint graph. Then, the constraints are satisfiable.

Satisfying the constraints

Theorem. Suppose no negative-weight cycle exists in the constraint graph. Then, the constraints are satisfiable.

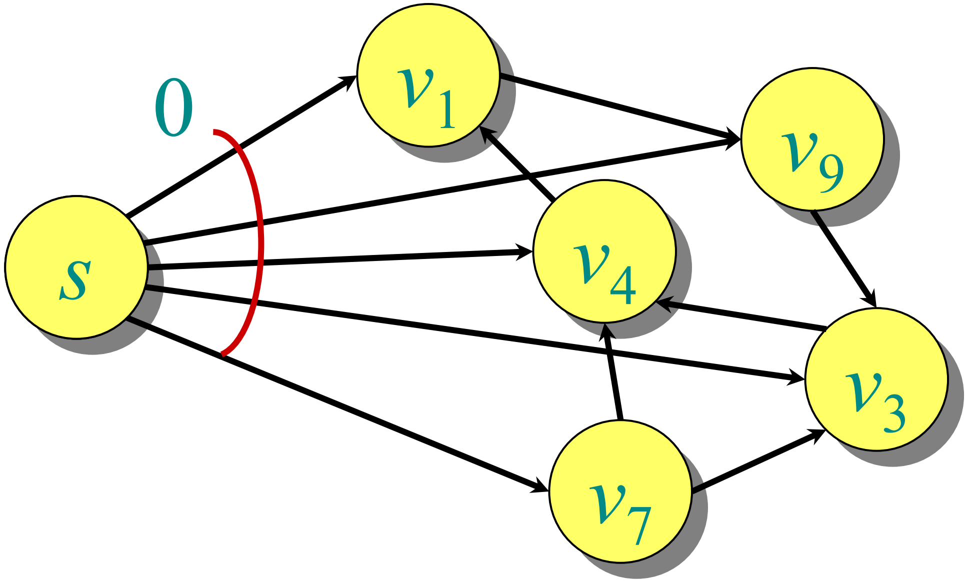
Proof. Add a new vertex s to V with a 0-weight edge to each vertex $v_i \in V$.



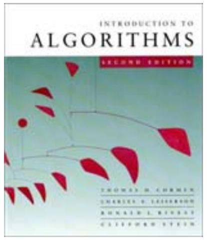
Satisfying the constraints

Theorem. Suppose no negative-weight cycle exists in the constraint graph. Then, the constraints are satisfiable.

Proof. Add a new vertex s to V with a 0-weight edge to each vertex $v_i \in V$.



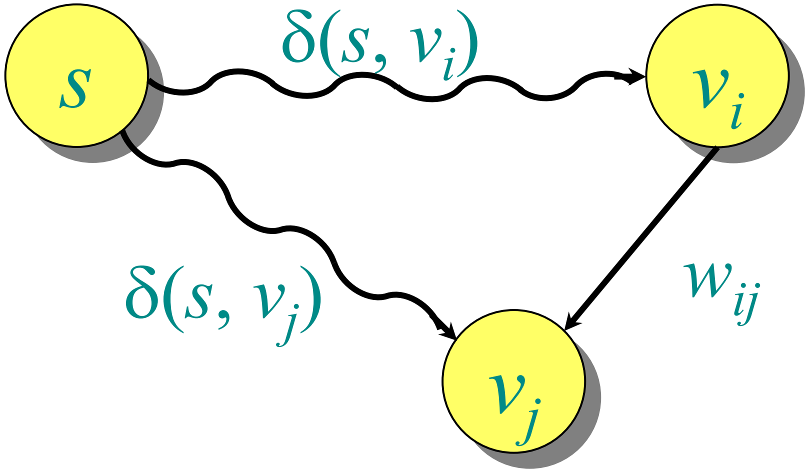
Note:
No negative-weight cycles introduced ⇒ shortest paths exist.



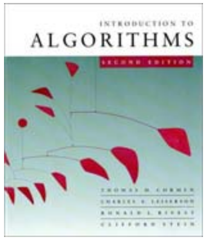
Proof (continued)

Claim: The assignment $x_i = \delta(s, v_i)$ solves the constraints.

Consider any constraint $x_j - x_i \le w_{ij}$, and consider the shortest paths from s to v_j and v_i:



The triangle inequality gives us $\delta(s,v_j) \le \delta(s, v_i) + w_{ij}$. Since $x_i = \delta(s, v_i)$ and $x_j = \delta(s, v_j)$, the constraint $x_j - x_i \le w_{ij}$ is satisfied. □



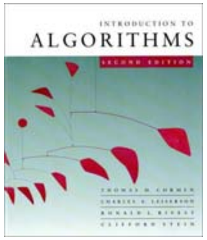
Bellman-Ford and linear programming

Corollary. The Bellman-Ford algorithm can solve a system of m difference constraints on n variables in $O(mn)$ time. □

Single-source shortest paths is a simple LP problem.

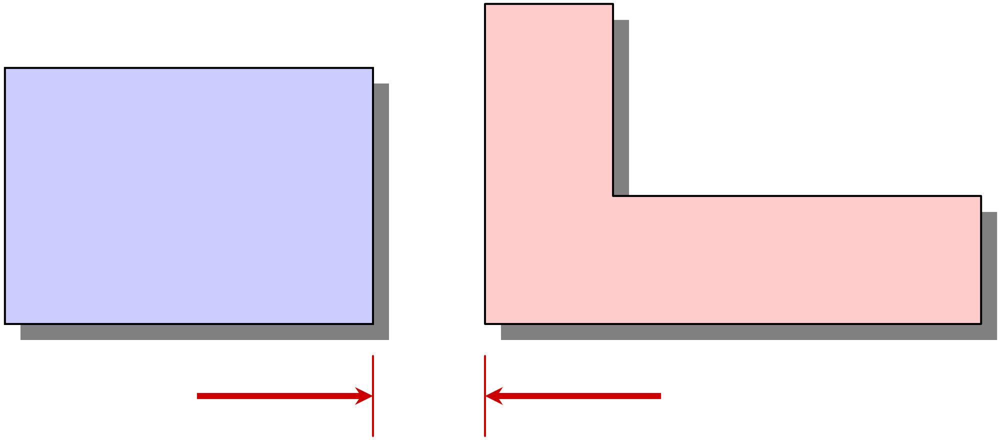
In fact, Bellman-Ford maximizes $x_1 + x_2 + \cdots + x_n$ subject to the constraints $x_j - x_i \le w_{ij}$ and $x_i \le 0$ (exercise).

Bellman-Ford also minimizes $\max_i\{x_i\} - \min_i\{x_i\}$ (exercise).



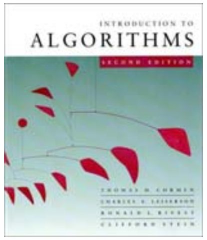
Application to VLSI layout compaction

Integrated -circuit features:

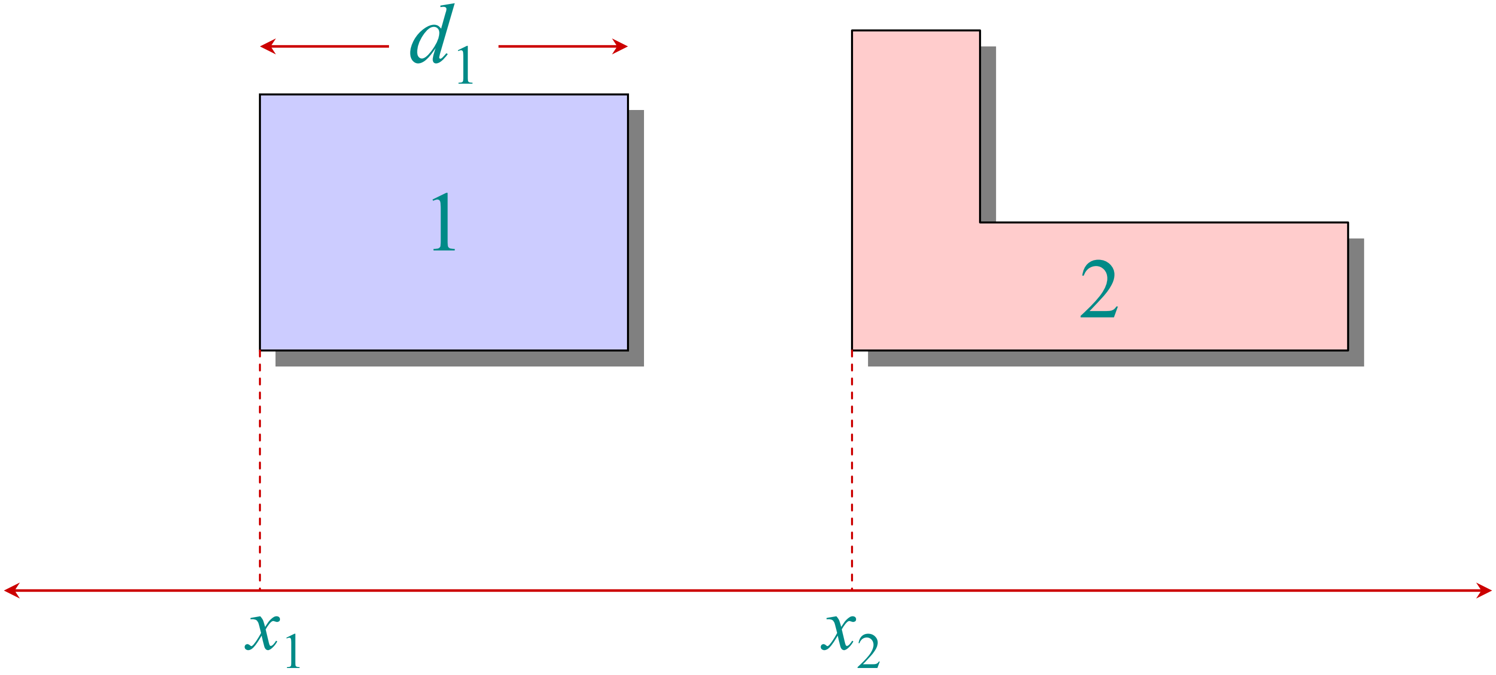


minimum separation λ

Problem: Compact (in one dimension) the space between the features of a VLSI layout without bringing any features too close together.



VLSI layout compaction

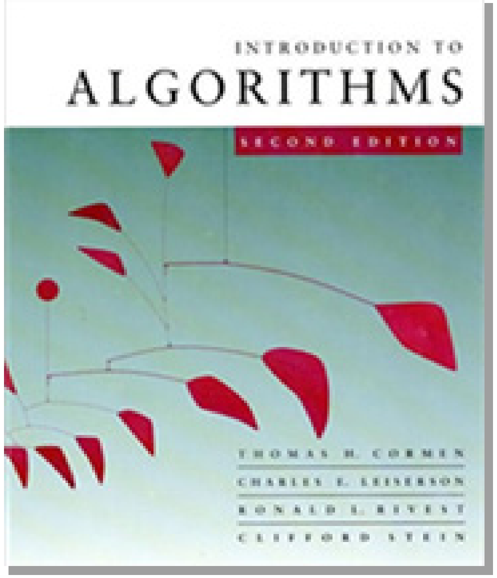


Constraint: $x_2 - x_1 \geq d_1 + \lambda$

Bellman-Ford minimizes $\max_i\{x_i\} - \min_i\{x_i\}$, which compacts the layout in the x-dimension.

Introduction to Algorithms

6.046J/18.401J

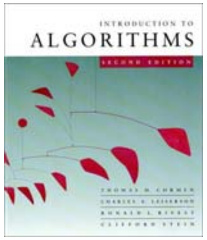


LECTURE 19

Shortest Paths III

- All-pairs shortest paths
- Matrix-multiplication algorithm
- Floyd-Warshall algorithm
- Johnson’s algorithm

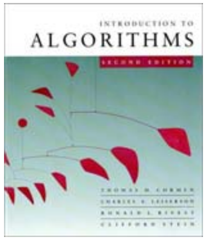
Prof. Charles E. Leiserson



Shortest paths

Single-source shortest paths

- Nonnegative edge weights
  - Dijkstra's algorithm: $O(E + V \lg V)$
- General
  - Bellman-Ford algorithm: $O(VE)$
- DAG
  - One pass of Bellman-Ford: $O(V + E)$



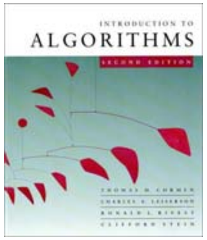
Shortest paths

Single-source shortest paths

- Nonnegative edge weights
  - Dijkstra's algorithm: $O(E + V \lg V)$
- General
  - Bellman-Ford: $O(VE)$
- DAG
  - One pass of Bellman-Ford: $O(V + E)$

All-pairs shortest paths

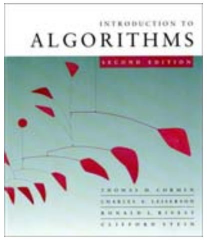
- Nonnegative edge weights
  - Dijkstra's algorithm $|V|$ times: $O(VE + V^2 \lg V)$
- General
  - Three algorithms today.



All-pairs shortest paths

Input: Digraph $G = (V, E)$, where $V = \{1, 2, \dots, n\}$, with edge-weight function $w : E \to \mathbb{R}$.

Output: $n \times n$ matrix of shortest-path lengths $\delta(i, j)$ for all $i, j \in V$.



All-pairs shortest paths

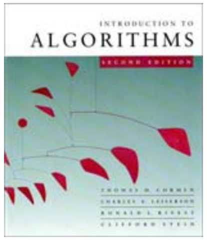
Input: Digraph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, with edge-weight function $w : E \to \mathbb{R}$.

Output: $n \times n$ matrix of shortest-path lengths $\delta(i, j)$ for all $i, j \in V$.

IDEA:

- Run Bellman-Ford once from each vertex.
- Time $= \mathrm{O}(V^2E)$.
- Dense graph (n^2 edges) $\Rightarrow$ $\Theta(n^4)$ time in the worst case.

Good first try!



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

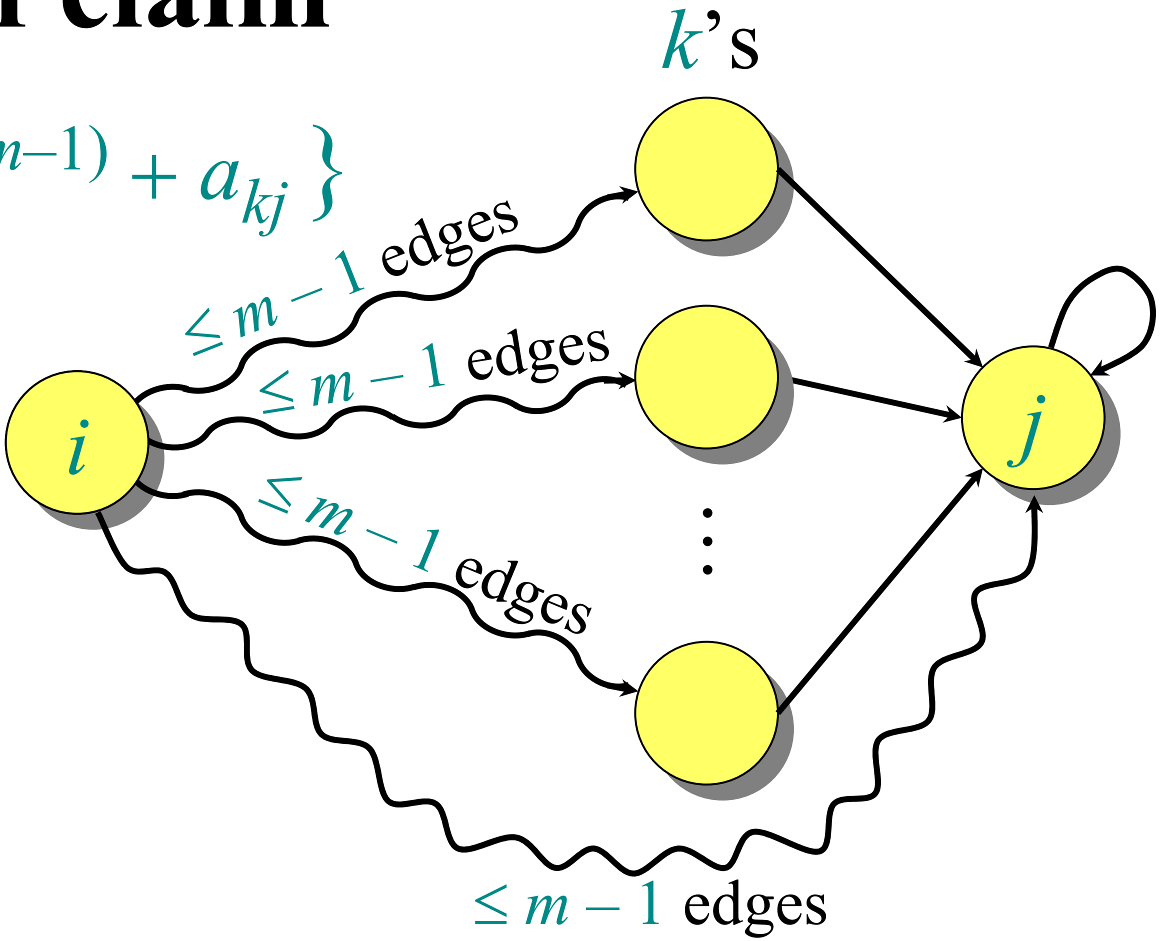
$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j, \\ \infty & \text{if } i \neq j; \end{cases}$$

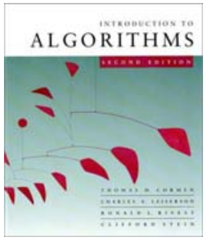
and for $m = 1, 2, \ldots, n - 1$,

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

Proof of claim

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



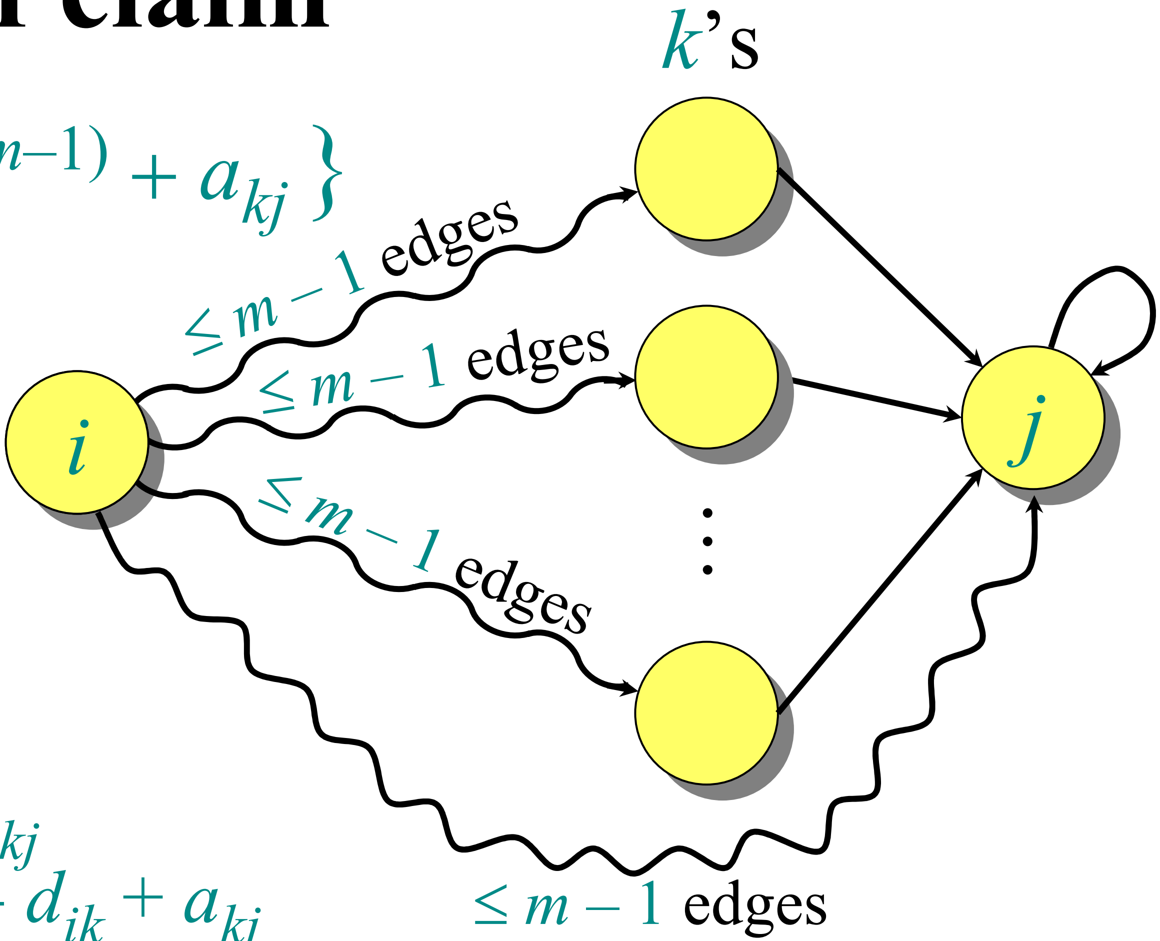


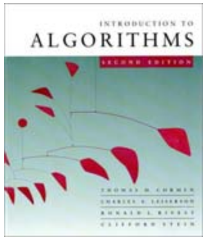
Proof of claim

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

Relaxation!

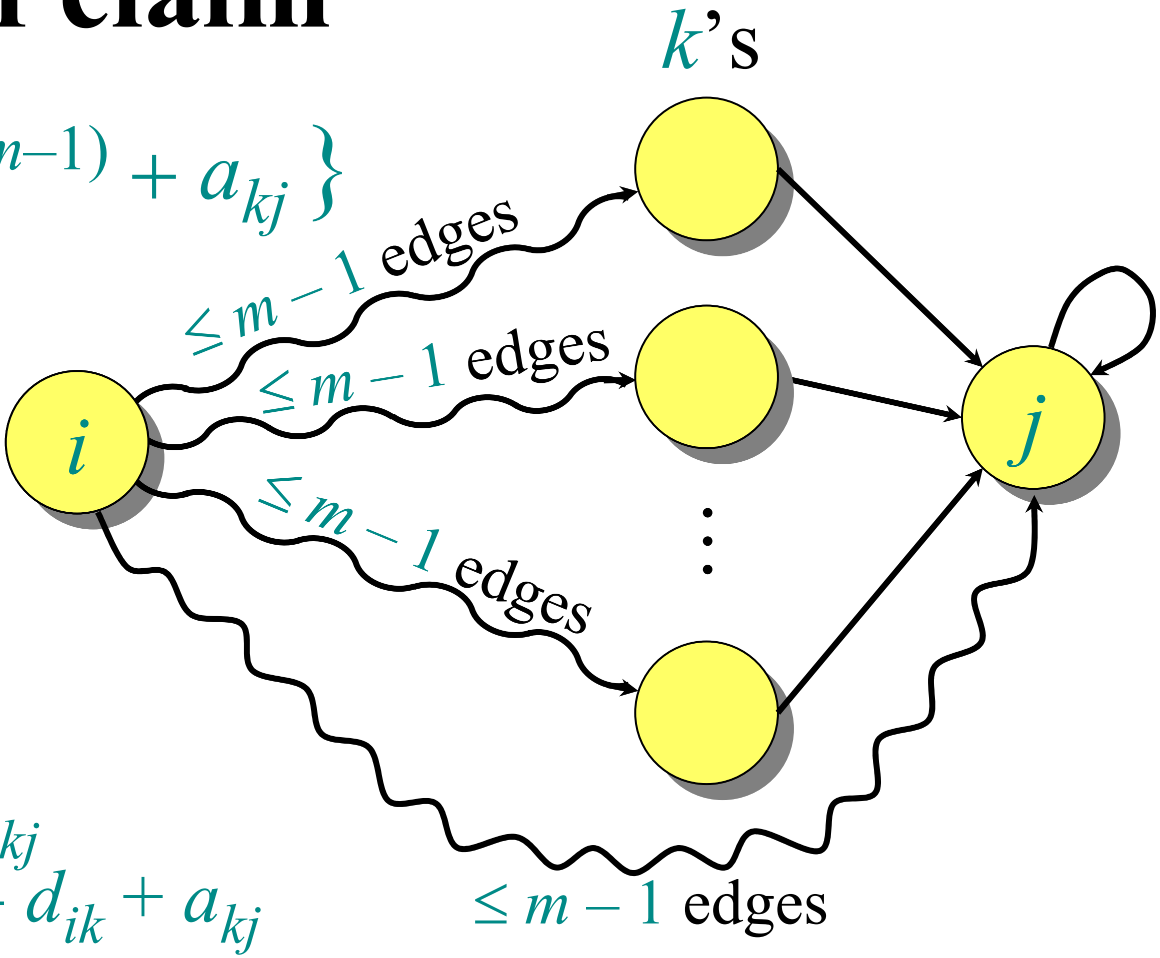
```
for $k \leftarrow 1$ to $n$
    do if $d_{ij} > d_{ik} + a_{kj}$
        then $d_{ij} \leftarrow d_{ik} + a_{kj}$
```





Proof of claim

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

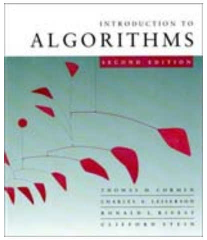


Relaxation!

```
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

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

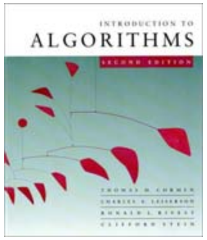


Matrix multiplication

Compute $C = A \cdot B$, where C, A, and B are $n \times n$ matrices:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} .$$

Time $= \Theta(n^3)$ using the standard algorithm.



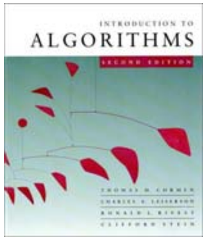
Matrix multiplication

Compute $C = A \cdot B$, where C, A, and B are $n \times n$ matrices:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} .$$

Time $= \Theta(n^3)$ using the standard algorithm.

What if we map “+” → “min” and “·” → “+”?



Matrix multiplication

Compute $C = A \cdot B$, where C, A, and B are $n \times n$ matrices:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \, .$$

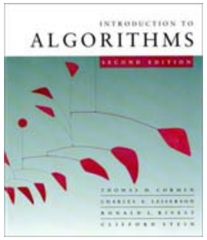
Time $= \Theta(n^3)$ using the standard algorithm.

What if we map "+" → "min" and "·" → "+"?

$$c_{ij} = \min_k \{a_{ik} + b_{kj}\} \, .$$

Thus, $D^{(m)} = D^{(m-1)}$ "×" A.

Identity matrix $= \mathrm{I} = \begin{pmatrix} 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix} = D^0 = (d_{ij}{}^{(0)})$.



Matrix multiplication (continued)

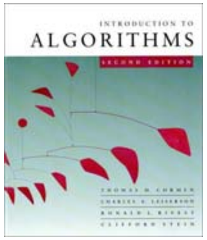
The (min, +) multiplication is ***associative***, and with the real numbers, it forms an algebraic structure called a ***closed semiring***.

Consequently, we can compute

$$\begin{aligned} D^{(1)} &= D^{(0)} \cdot A = A^1 \\ D^{(2)} &= D^{(1)} \cdot A = A^2 \\ &\vdots \qquad\qquad \vdots \\ D^{(n-1)} &= D^{(n-2)} \cdot A = A^{n-1}, \end{aligned}$$

yielding $D^{(n-1)} = (\delta(i, j))$.

Time $= \Theta(n \cdot n^3) = \Theta(n^4)$. No better than n × B-F.



Improved matrix multiplication algorithm

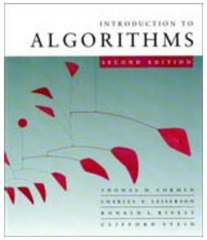
Repeated squaring: $A^{2k} = A^k \times A^k$.

Compute $\underbrace{A^2, A^4, \ldots, A^{2^{\lceil \lg(n-1) \rceil}}}_{O(\lg n) \text{ squarings}}$.

Note: $A^{n-1} = A^n = A^{n+1} = \cdots$.

Time $= \Theta(n^3 \lg n)$.

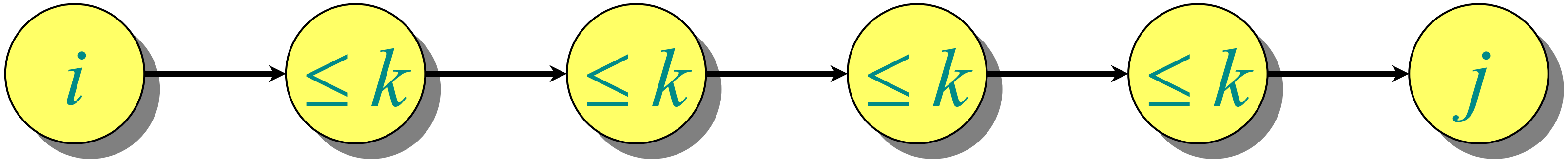
To detect negative-weight cycles, check the diagonal for negative values in $O(n)$ additional time.



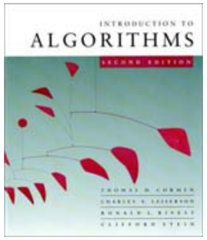
Floyd-Warshall algorithm

Also dynamic programming, but faster!

Define $c_{ij}^{(k)} =$ weight of a shortest path from i to j with intermediate vertices belonging to the set $\{1, 2, \ldots, k\}$.

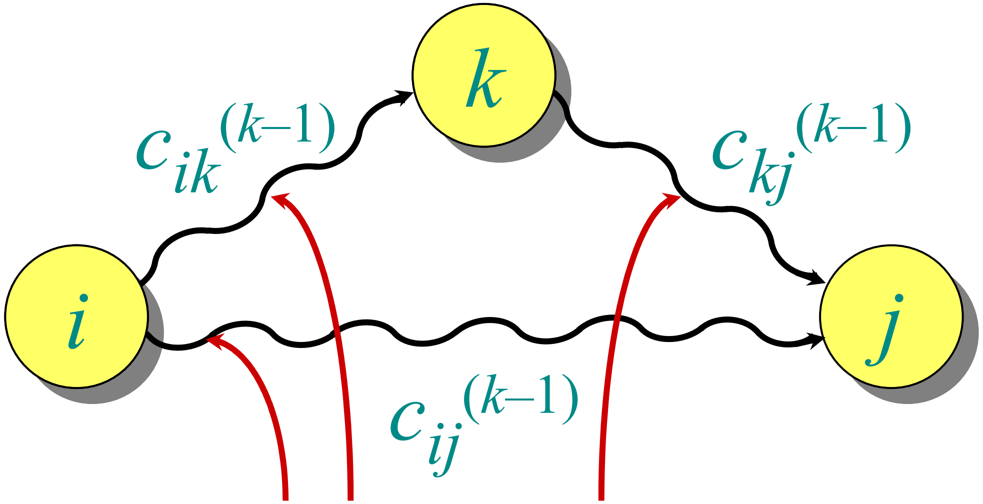


Thus, $\delta(i, j) = c_{ij}^{(n)}$. Also, $c_{ij}^{(0)} = a_{ij}$.

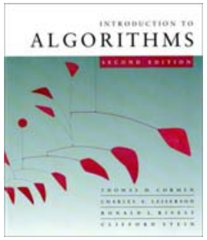


Floyd-Warshall recurrence

$$c_{ij}^{(k)} = \min_k \{c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)}\}$$



intermediate vertices in $\{1, 2, \ldots, k\}$



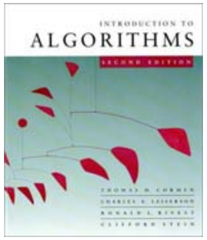
Pseudocode for Floyd-Warshall

```
for $k \leftarrow 1$ to $n$
    do for $i \leftarrow 1$ to $n$
        do for $j \leftarrow 1$ to $n$
            do if $c_{ij} > c_{ik} + c_{kj}$
                then $c_{ij} \leftarrow c_{ik} + c_{kj}$
```

relaxation

Notes:

- Okay to omit superscripts, since extra relaxations can't hurt.
- Runs in $\Theta(n^3)$ time.
- Simple to code.
- Efficient in practice.



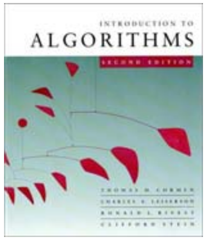
Transitive closure of a directed graph

Compute $t_{ij} = \begin{cases} 1 & \text{if there exists a path from } i \text{ to } j, \\ 0 & \text{otherwise.} \end{cases}$

Idea: Use Floyd-Warshall, but with $(\vee, \wedge)$ instead of $(\min, +)$:

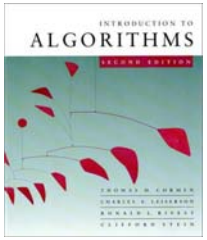
$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}).$$

Time = $\Theta(n^3)$.



Graph reweighting

Theorem. Given a function $h : V \to \mathbb{R}$, ***reweight*** each edge $(u, v) \in E$ by $w_h(u, v) = w(u, v) + h(u) - h(v)$. Then, for any two vertices, all paths between them are reweighted by the same amount.



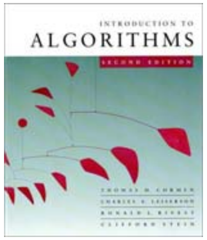
Graph reweighting

Theorem. Given a function $h : V \to \mathbb{R}$, ***reweight*** each edge $(u, v) \in E$ by $w_h(u, v) = w(u, v) + h(u) - h(v)$. Then, for any two vertices, all paths between them are reweighted by the same amount.

Proof. Let $p = v_1 \to v_2 \to \cdots \to v_k$ be a path in G. We have

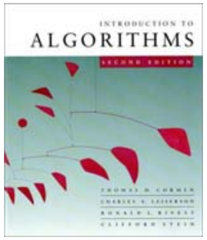
$$\begin{aligned} w_h(p) &= \sum_{i=1}^{k-1} w_h(v_i, v_{i+1}) \\ &= \sum_{i=1}^{k-1} \big(w(v_i, v_{i+1}) + h(v_i) - h(v_{i+1})\big) \\ &= \sum_{i=1}^{k-1} w(v_i, v_{i+1}) + h(v_1) - h(v_k) \\ &= w(p) + h(v_1) - h(v_k). \qquad \square \end{aligned}$$

Same amount!



Shortest paths in reweighted graphs

Corollary. $\delta_h(u, v) = \delta(u, v) + h(u) - h(v)$. □

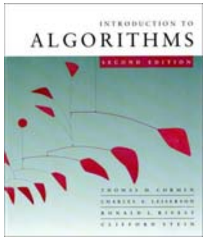


Shortest paths in reweighted graphs

Corollary. $\delta_h(u, v) = \delta(u, v) + h(u) - h(v)$. □

IDEA: Find a function $h : V \to \mathbb{R}$ such that $w_h(u, v) \geq 0$ for all $(u, v) \in E$. Then, run Dijkstra's algorithm from each vertex on the reweighted graph.

NOTE: $w_h(u, v) \geq 0$ iff $h(v) - h(u) \leq w(u, v)$.



Johnson's algorithm

1. Find a function $h : V \to \mathbb{R}$ such that $w_h(u, v) \geq 0$ for all $(u, v) \in E$ by using Bellman-Ford to solve the difference constraints $h(v) - h(u) \leq w(u, v)$, or determine that a negative-weight cycle exists.
   - Time = $O(VE)$.
2. Run Dijkstra's algorithm using w_h from each vertex $u \in V$ to compute $\delta_h(u, v)$ for all $v \in V$.
   - Time = $O(VE + V^2 \lg V)$.
3. For each $(u, v) \in V \times V$, compute
   $$\delta(u, v) = \delta_h(u, v) - h(u) + h(v) .$$
   - Time = $O(V^2)$.

Total time = $O(VE + V^2 \lg V)$.