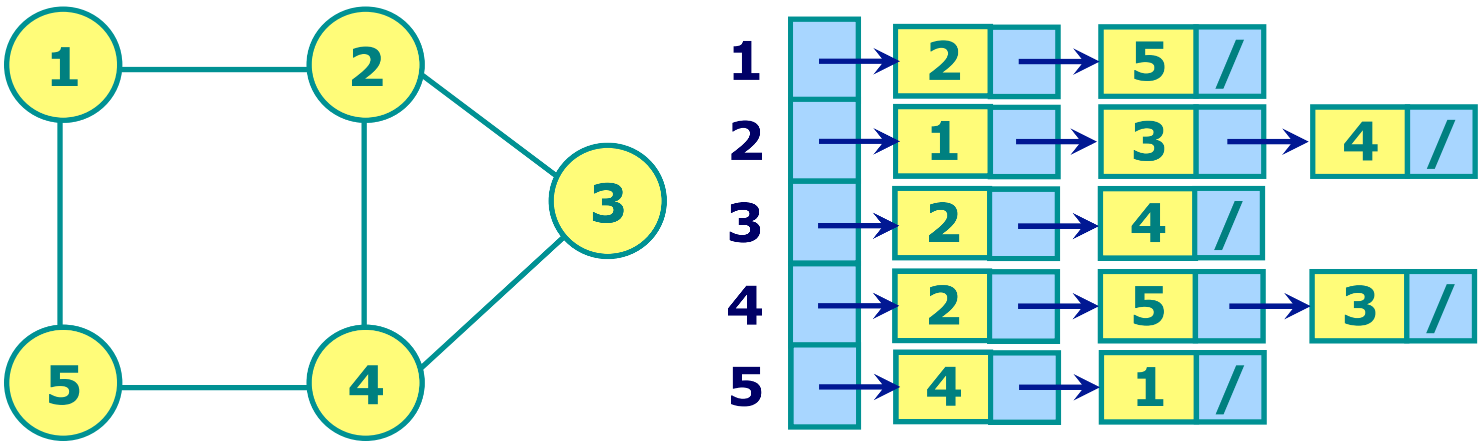
Data Structures and Algorithm

Xiaoqing Zheng
zhengxq@fudan.edu.cn

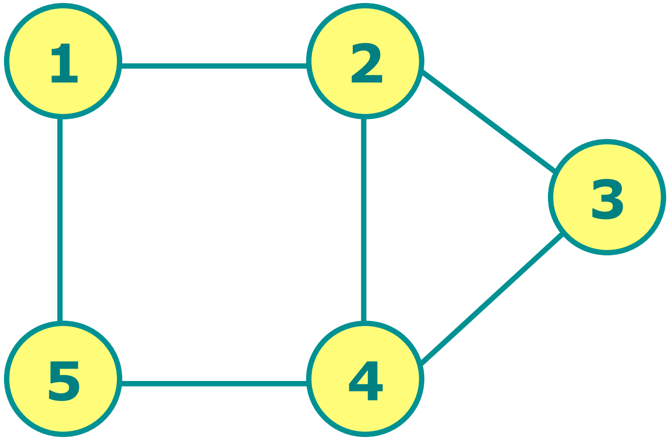
Representations of undirected graph



Adjacency-list representation of graph

$$G = (V, E)$$

Representations of undirected graph

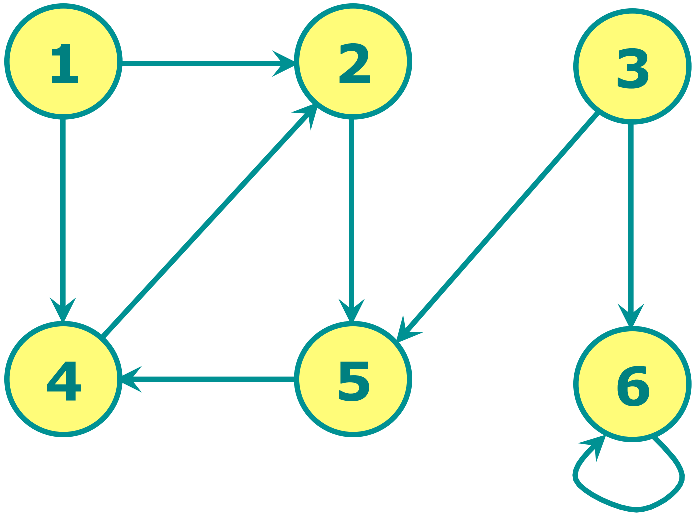


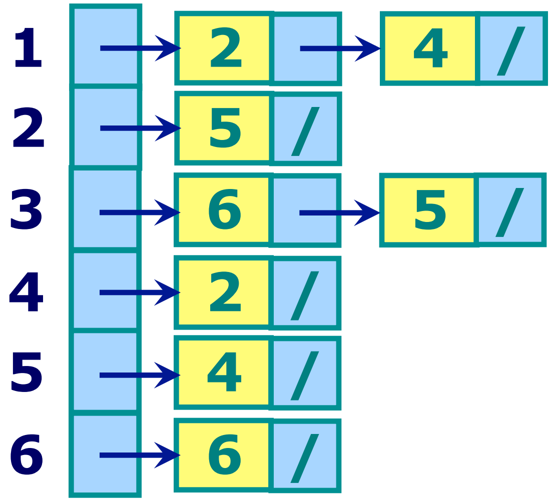
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	0
3	0	1	0	1	0
4	0	1	1	0	1
5	1	0	0	1	0

Adjacency-matrix representation of graph

$$G = (V, E)$$

Representations of directed graph

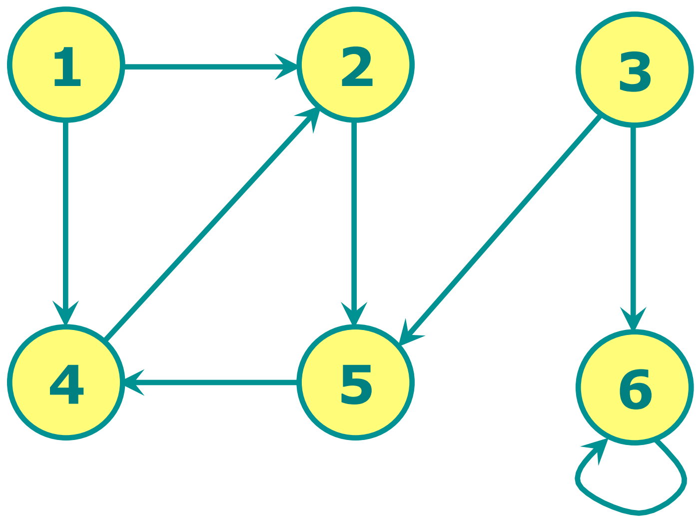




Adjacency-list representation of graph

$G = (V, E)$

Representations of directed graph



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Adjacency-matrix representation of graph

$G = (V, E)$

Graphs

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

- a set V of ***vertices*** (singular: ***vertex***),
- a set $E \in V \times V$ 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 $|E| \geq |V| - 1$.

Representations of graph

Adjacency-list representation

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

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

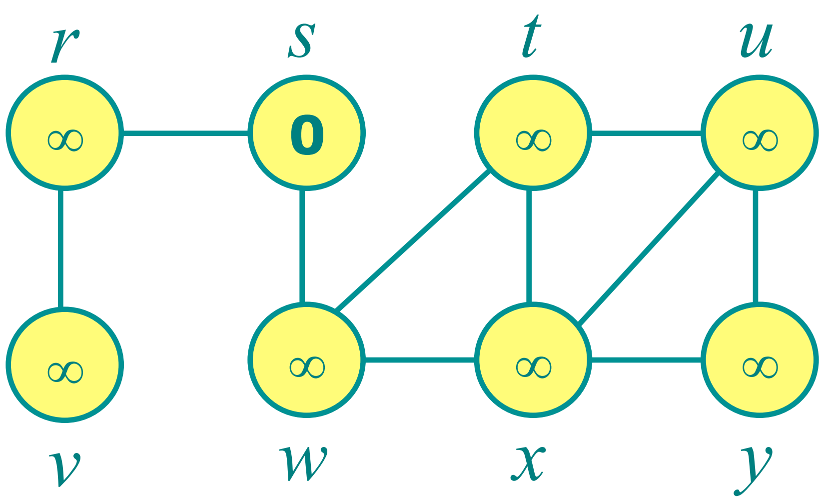
Adjacency-matrix representation

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

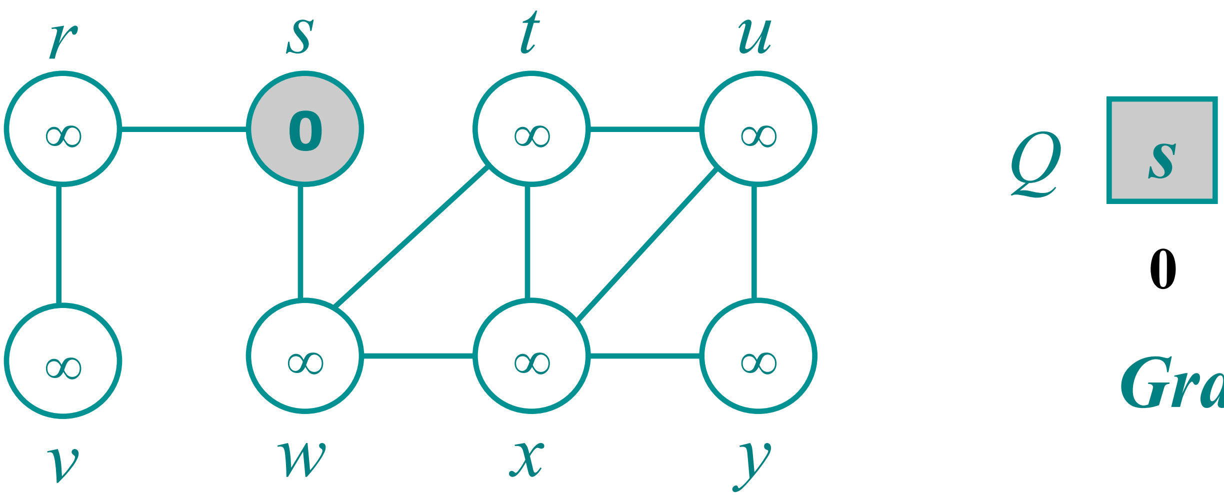
$$A[i, j] = \begin{cases} 1 \text{ if } (i, j) \in E, \\ 0 \text{ if } (i, j) \notin E. \end{cases}$$

Breadth-first search

Given a graph $G = (V, E)$ and a distinguished ***source*** vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s.

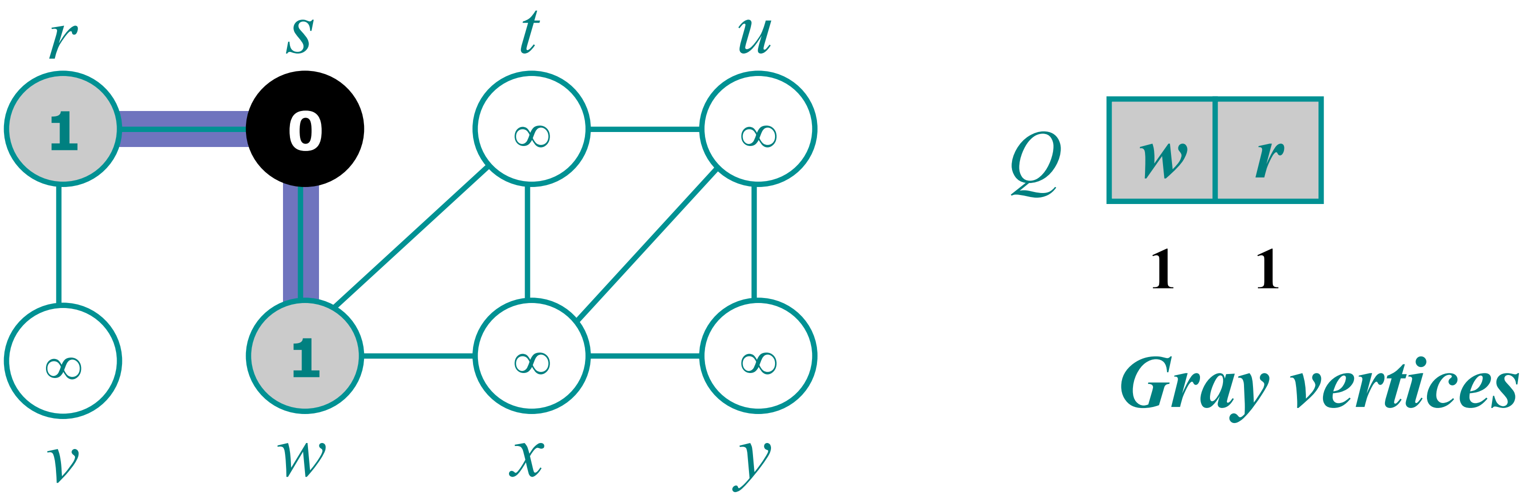


Breadth-first search example

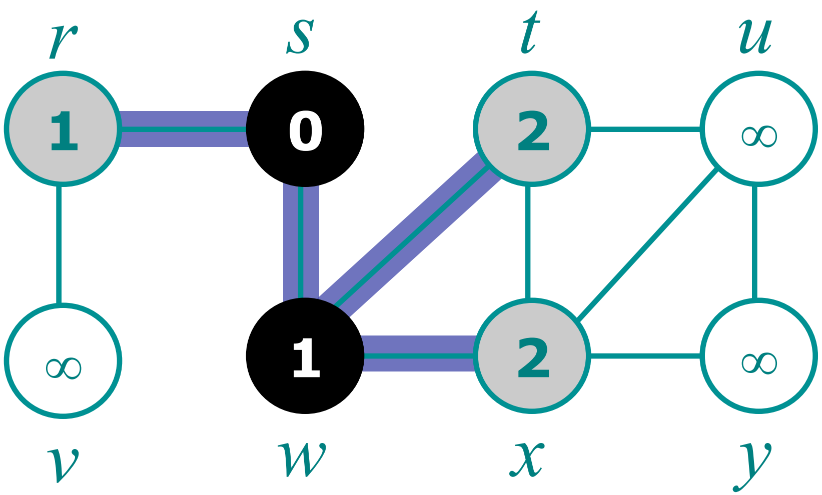


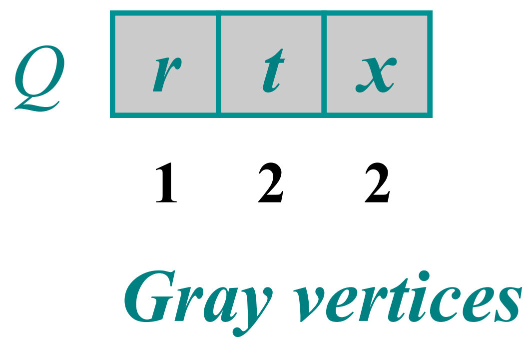
Gray vertices

Breadth-first search example

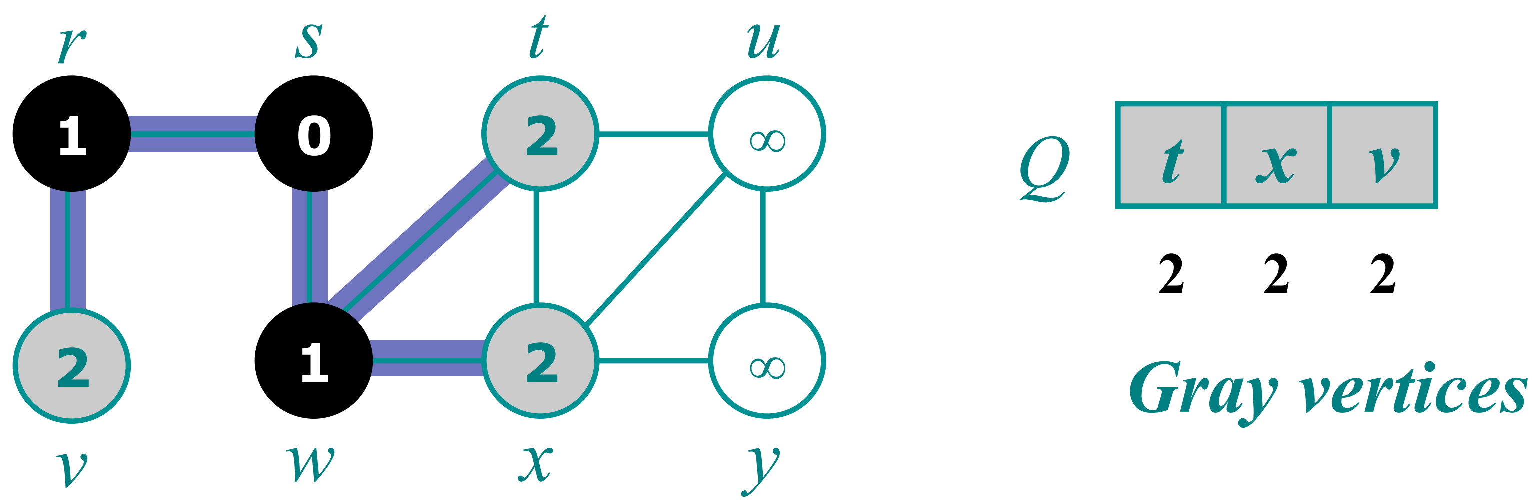


Breadth-first search example

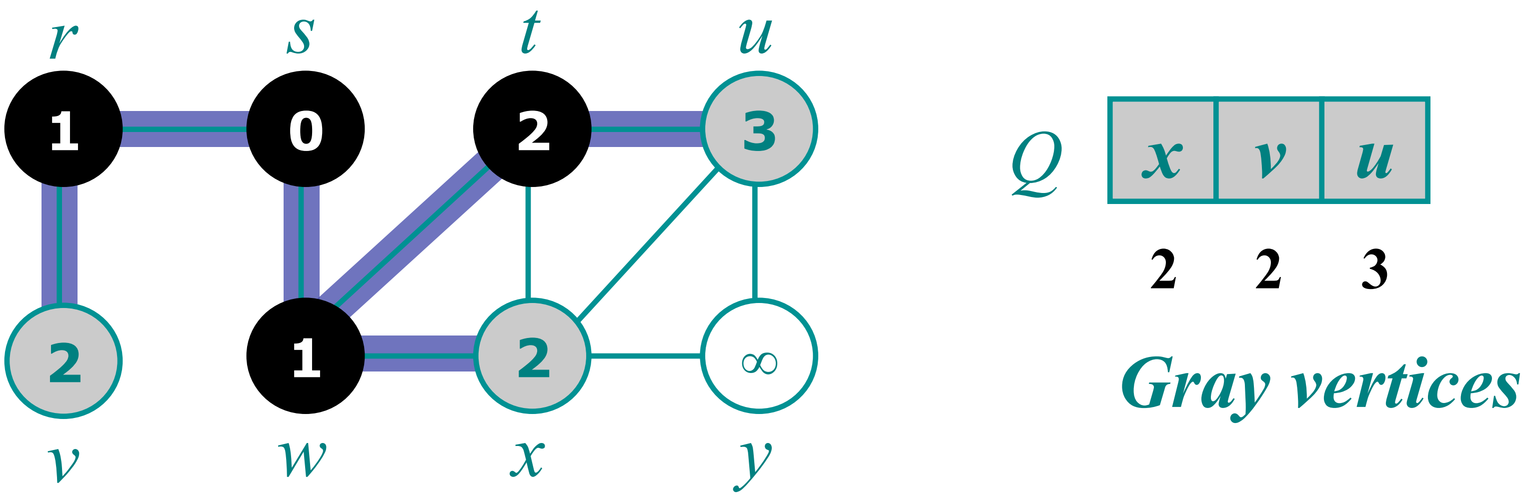




Breadth-first search example

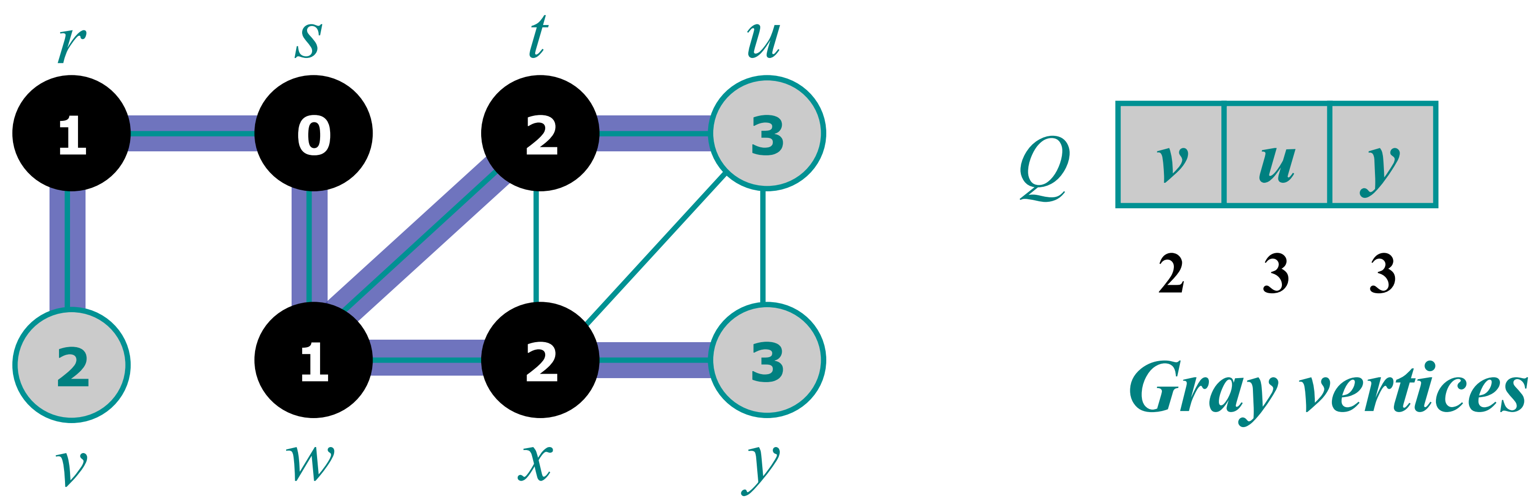


Breadth-first search example

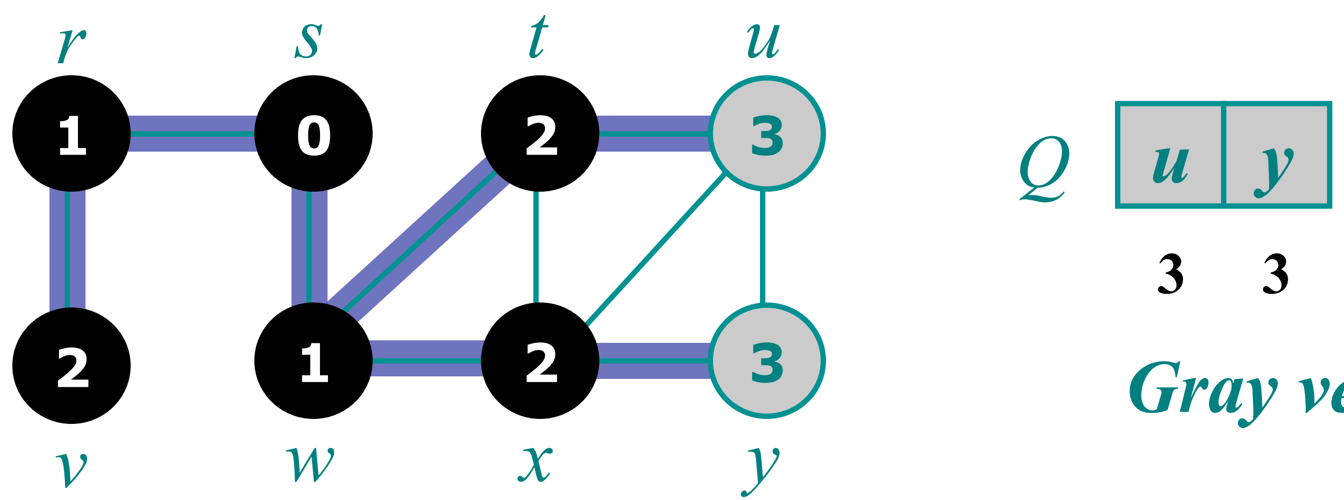


Why not x?

Breadth-first search example

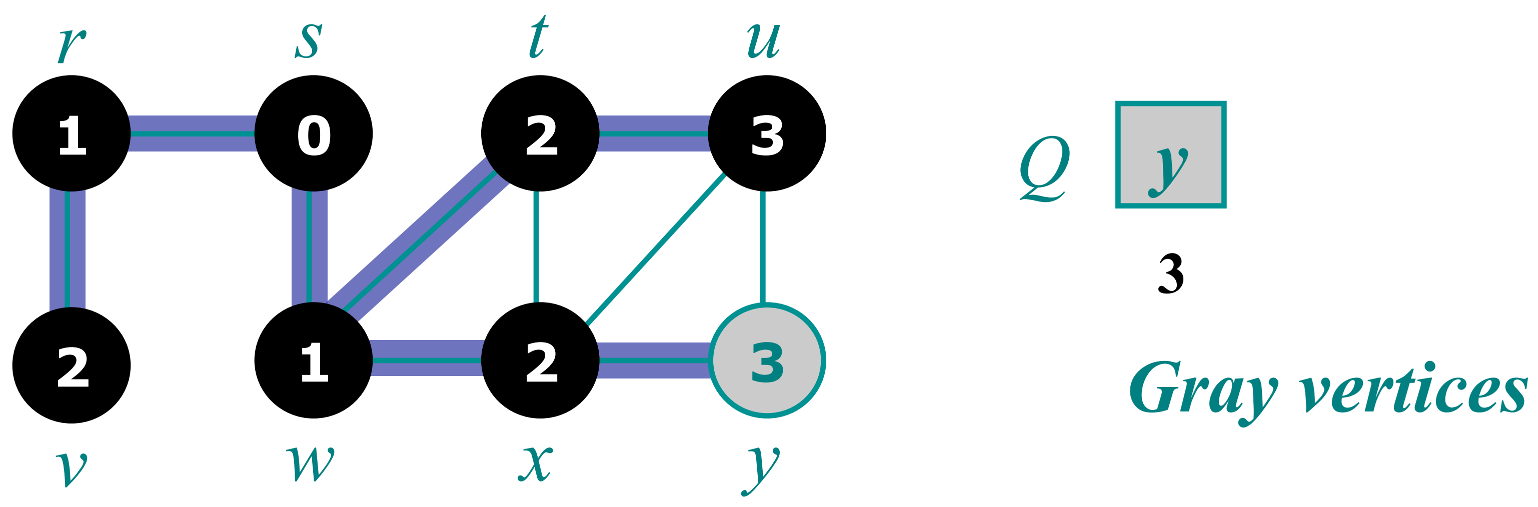


Breadth-first search example

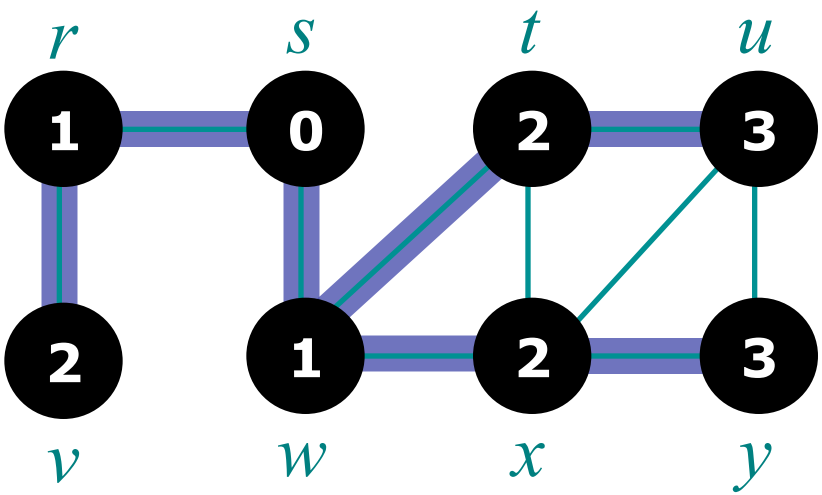


Gray vertices

Breadth-first search example



Breadth-first search example



$Q \quad \varnothing$

Gray vertices

Breadth-first search algorithm

BFS(G, s)

1. **for** each vertex $u \in V[G] - \{s\}$
2. **do** $color[u] \leftarrow$ WHITE
3. $d[u] \leftarrow \infty$
4. $\pi[u] \leftarrow$ NIL
5. $color[s] \leftarrow$ GRAY
6. $d[s] \leftarrow 0$
7. $\pi[s] \leftarrow$ NIL
8. $Q \leftarrow \varnothing$
9. ENQUEUE(Q, s)

Breadth-first search algorithm

BFS(G, s)

while $Q \neq \emptyset$
    do $u \leftarrow$ DEQUEUE(Q)
        for each $v \in Adj[u]$
            do if $color[v] =$ WHITE
$O(E)$ **then** $color[v] \leftarrow$ GRAY $O(V)$
times $d[v] \leftarrow d[u] + 1$ times
                    $\pi[v] \leftarrow u$
                    ENQUEUE(Q, v)
        $color[u] \leftarrow$ BLACK

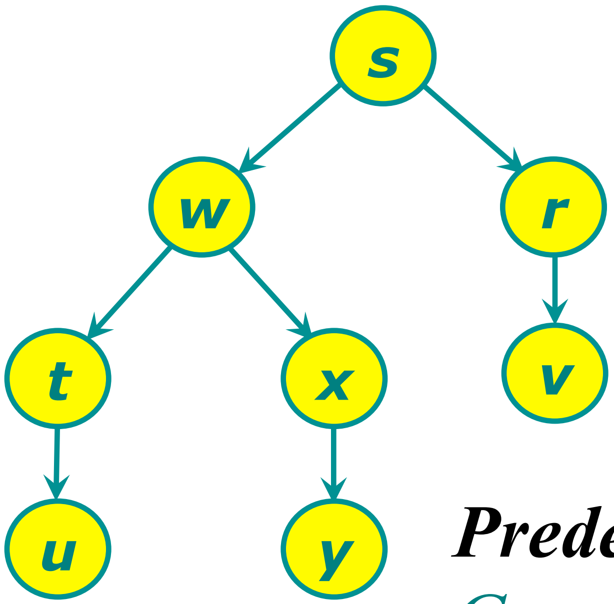
Running time is $O(V + E)$

Shortest paths

PRINT-PATH(G, s, v)

```
if v = s
   then print s
   else if π[v] = NIL
           then print "no path from" s "to" v "exists."
           else  PRINT-PATH(G, s, π[v] )
                 print v
```

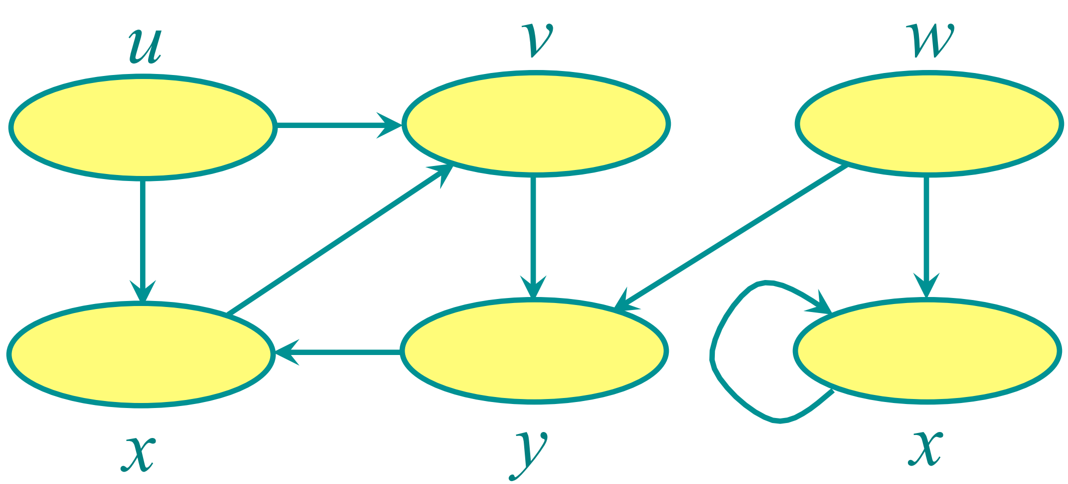
Predecessor subgraph of BFS



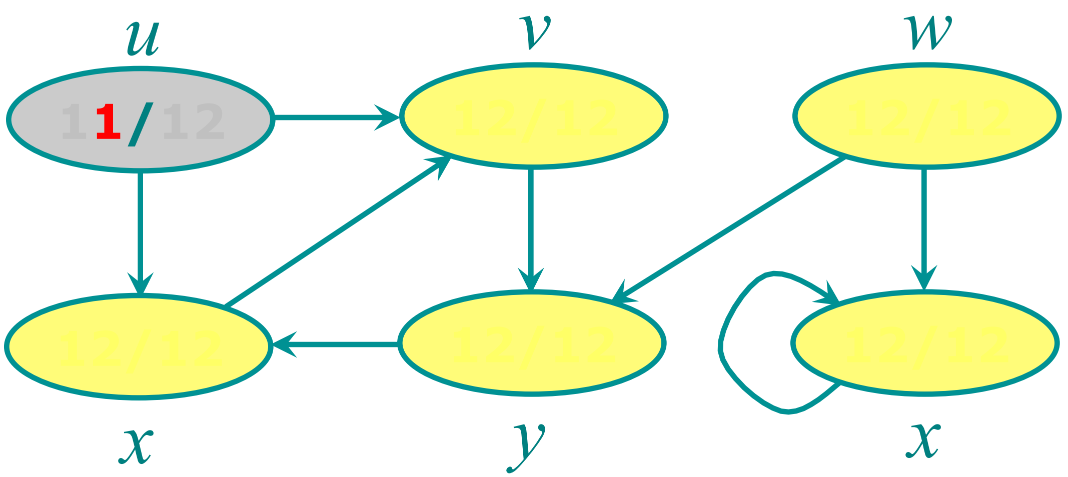
Predecessor subgraph of BFS is
$G_\pi = (V_\pi, E_\pi)$, where
$V_\pi = \{ v \in V : \pi[v] \neq \text{NIL} \} \cup \{s\}$ and
$E_\pi = \{ (\pi[v], v) : v \in V_\pi - \{s\} \}$.

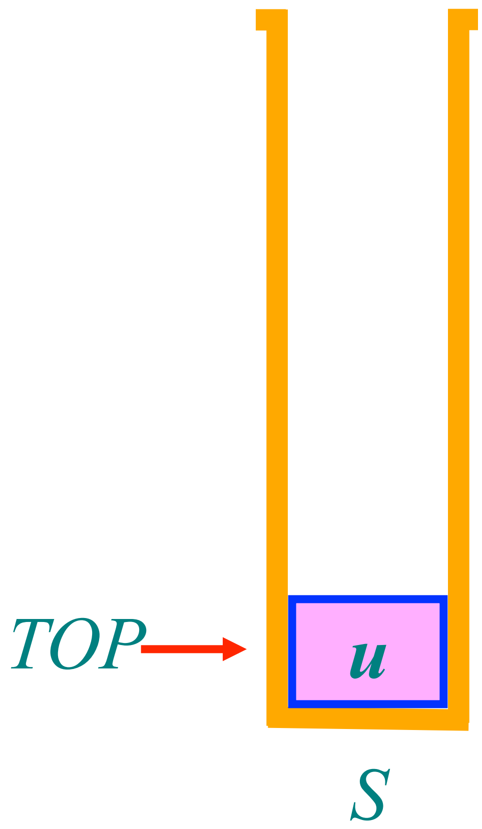
Depth-first search

Given a graph $G = (V, E)$, depth-first search is to search deeper in the graph whenever possible. Edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it.

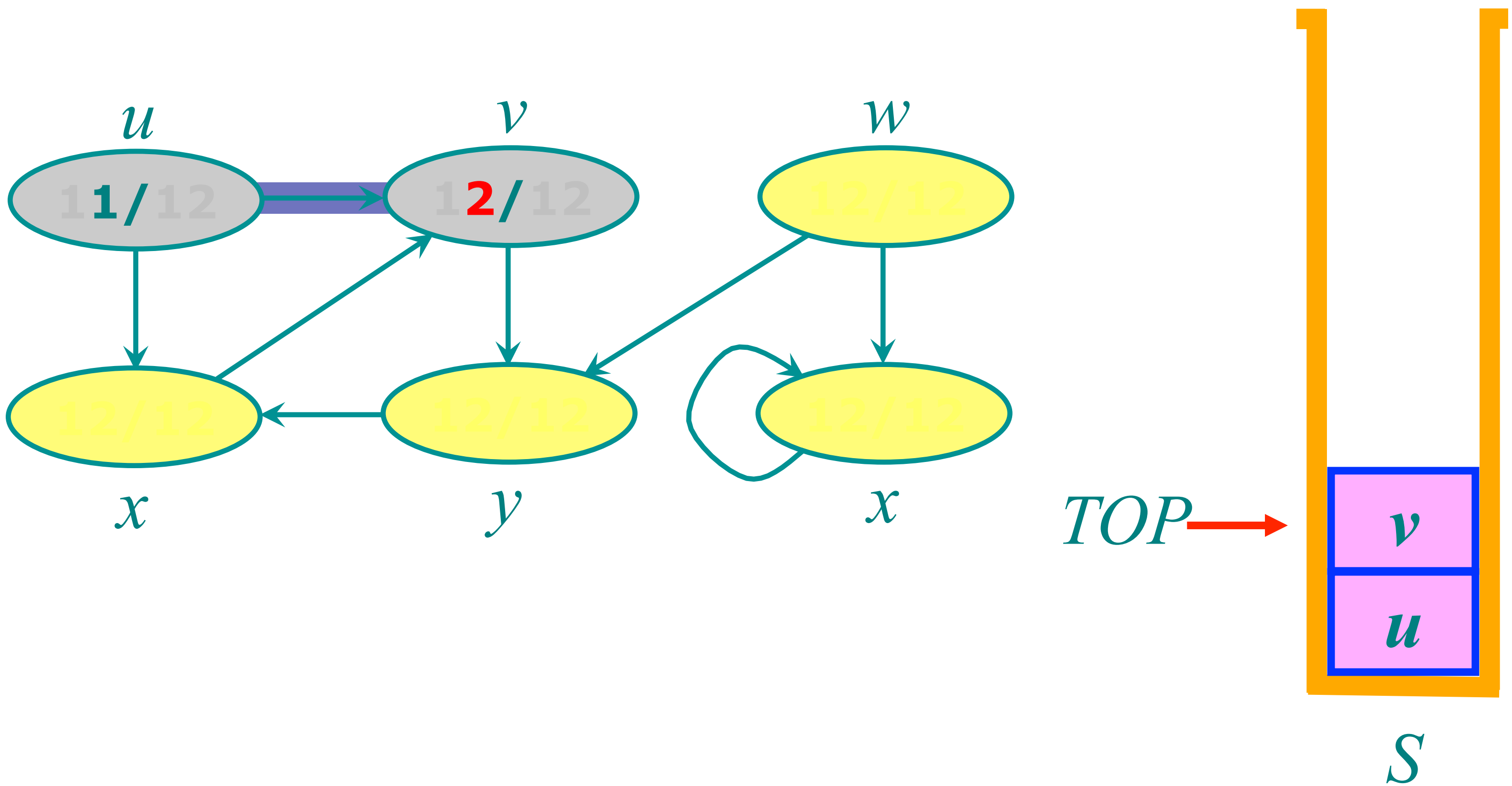


Depth-first search example

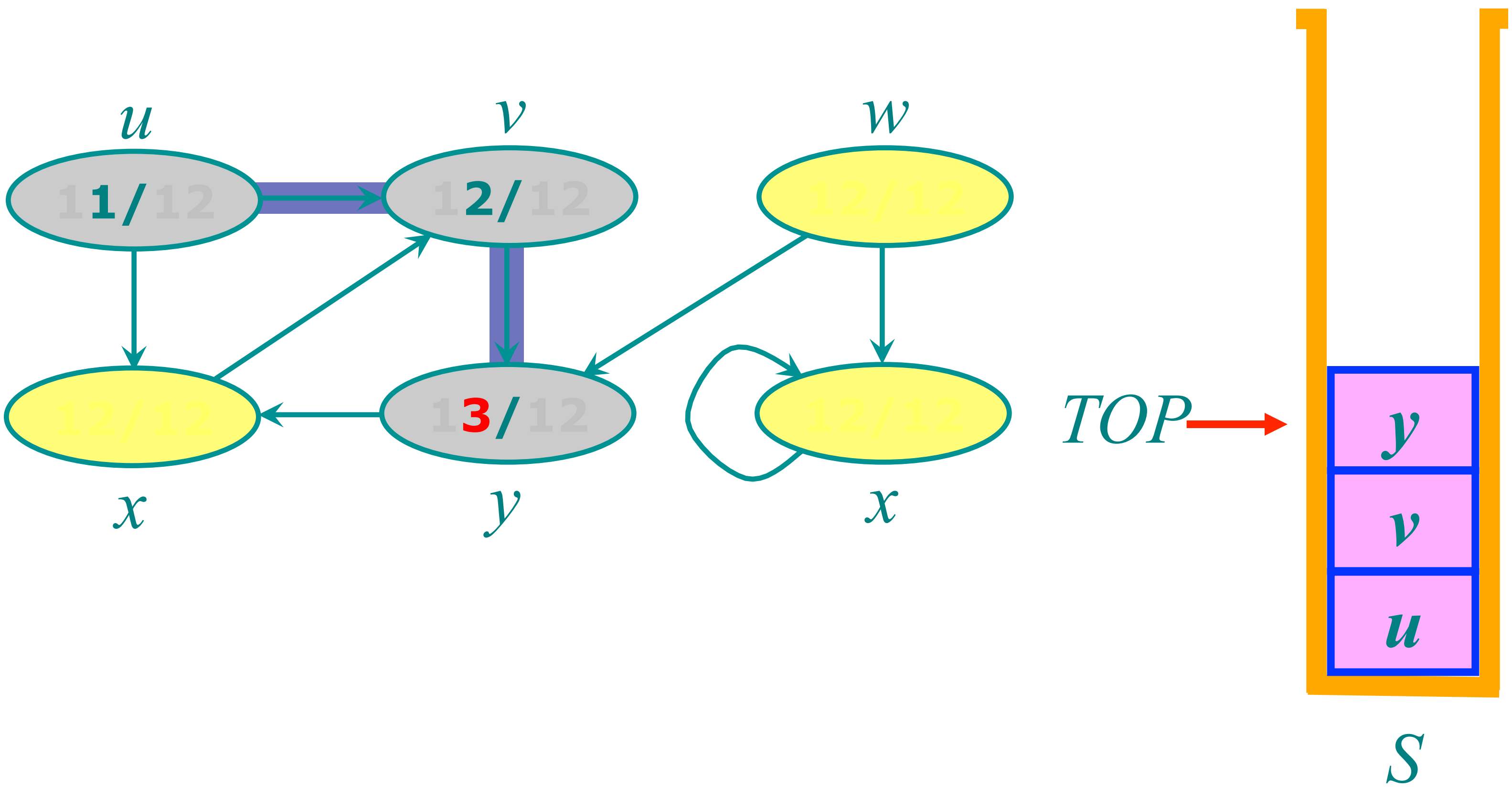




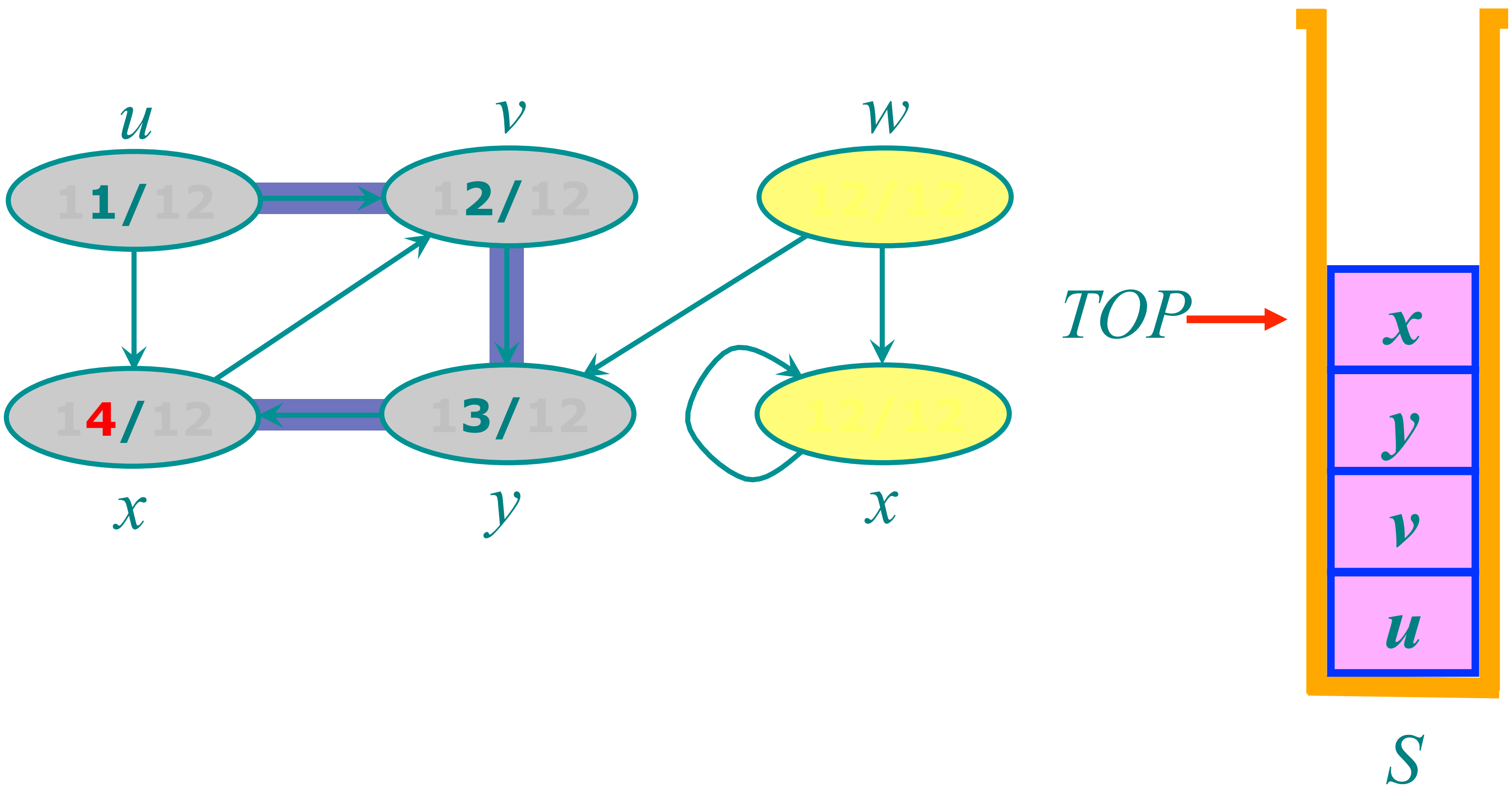
Depth-first search example



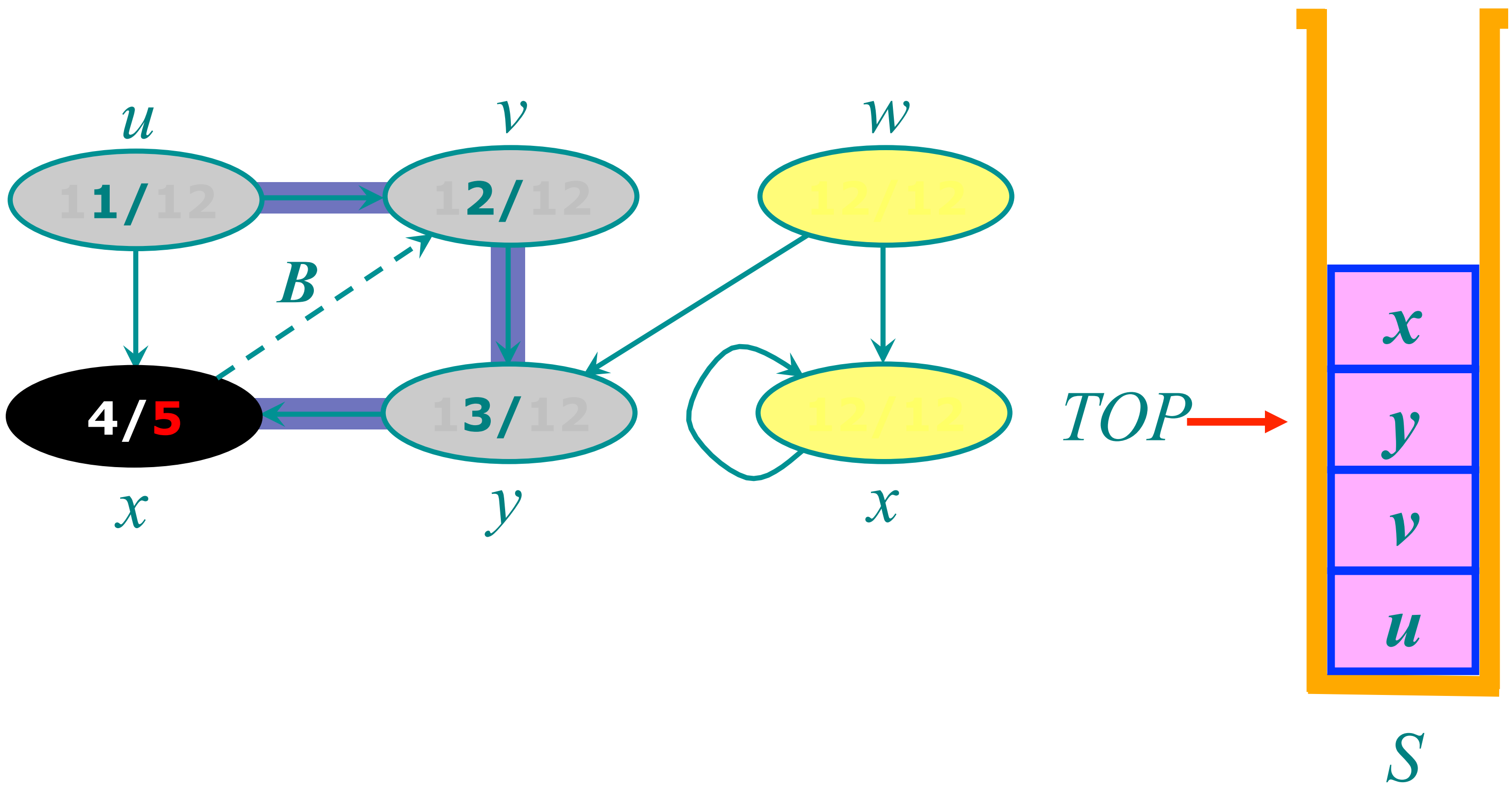
Depth-first search example



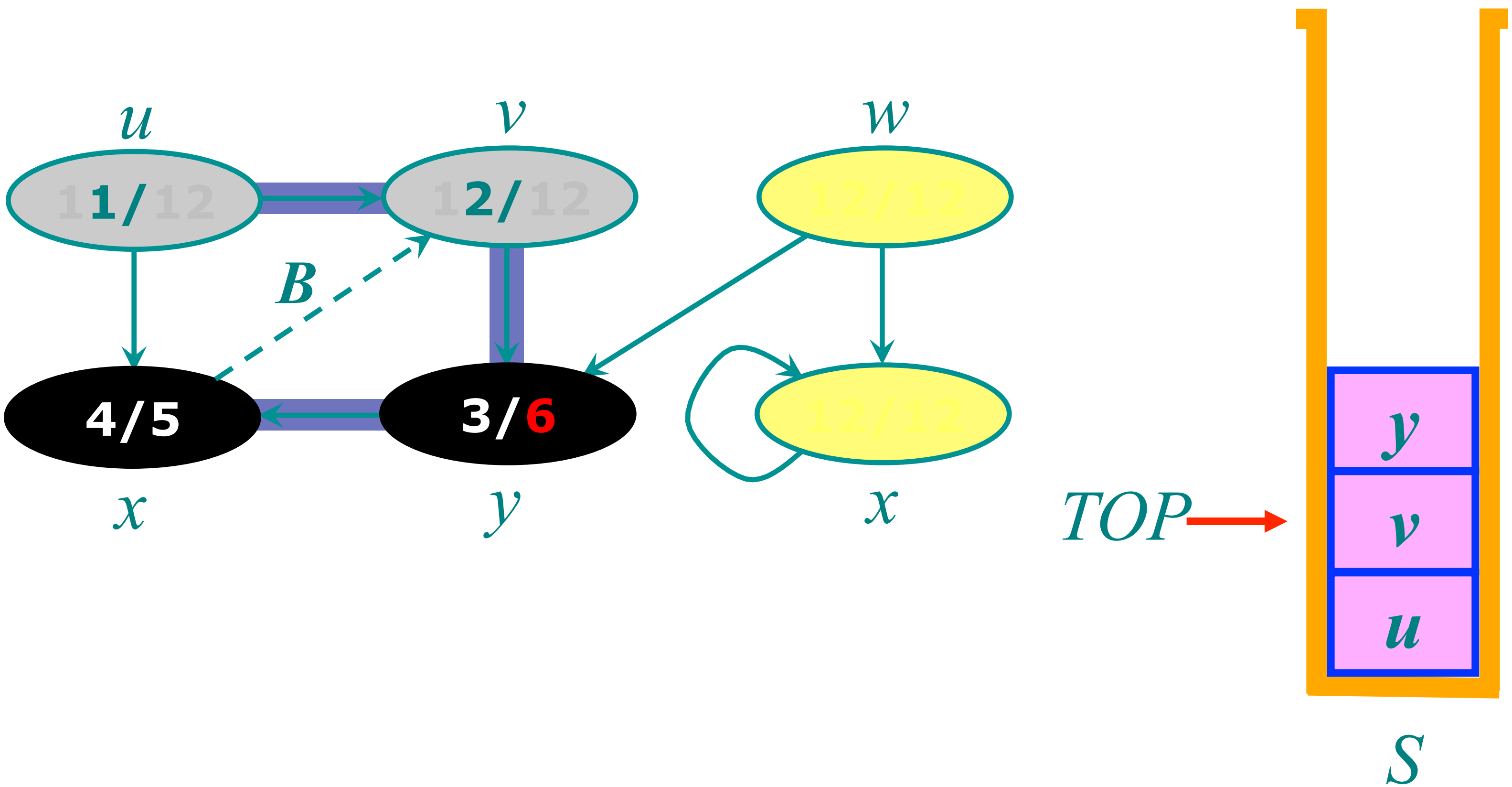
Depth-first search example



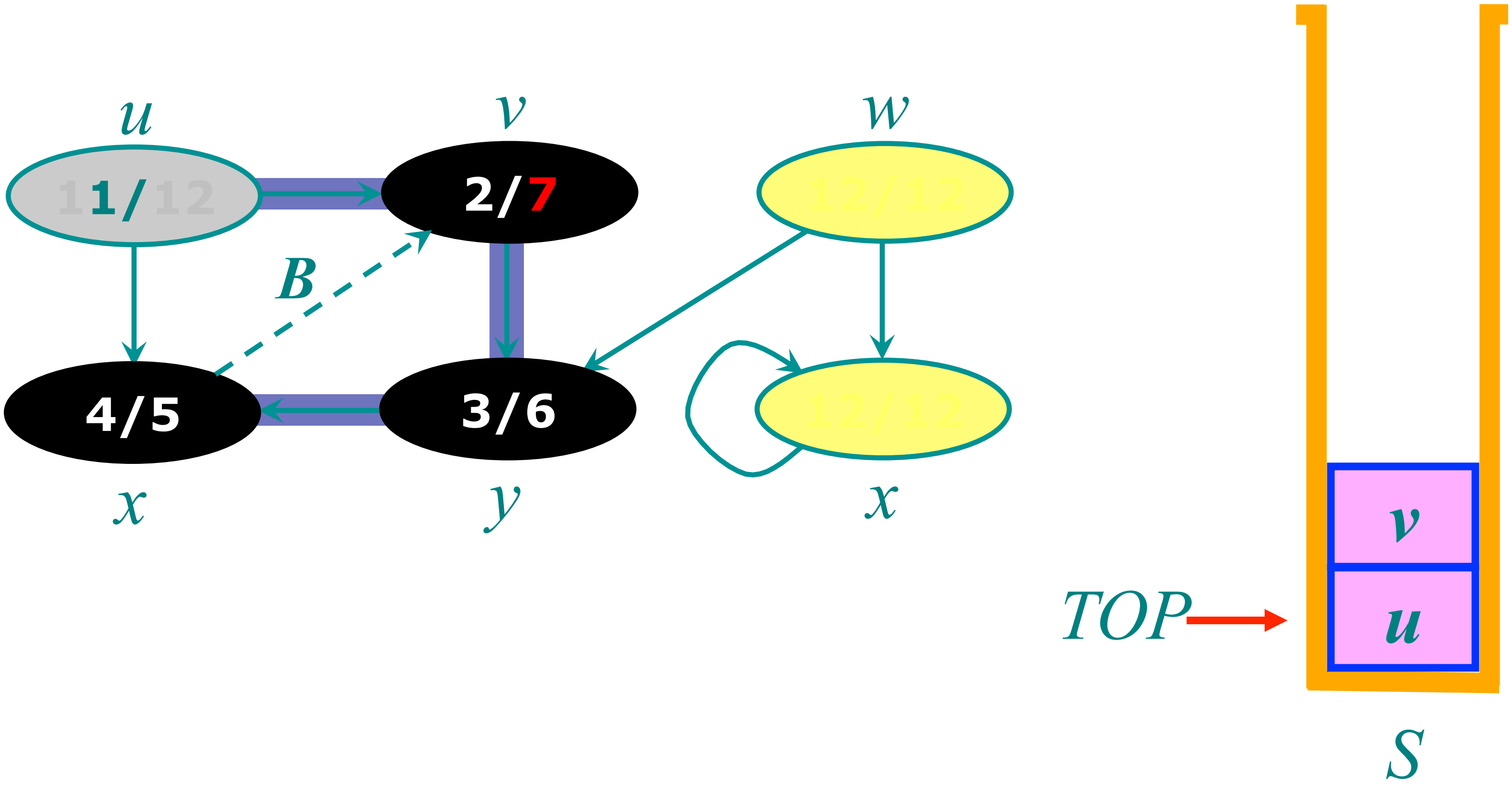
Depth-first search example



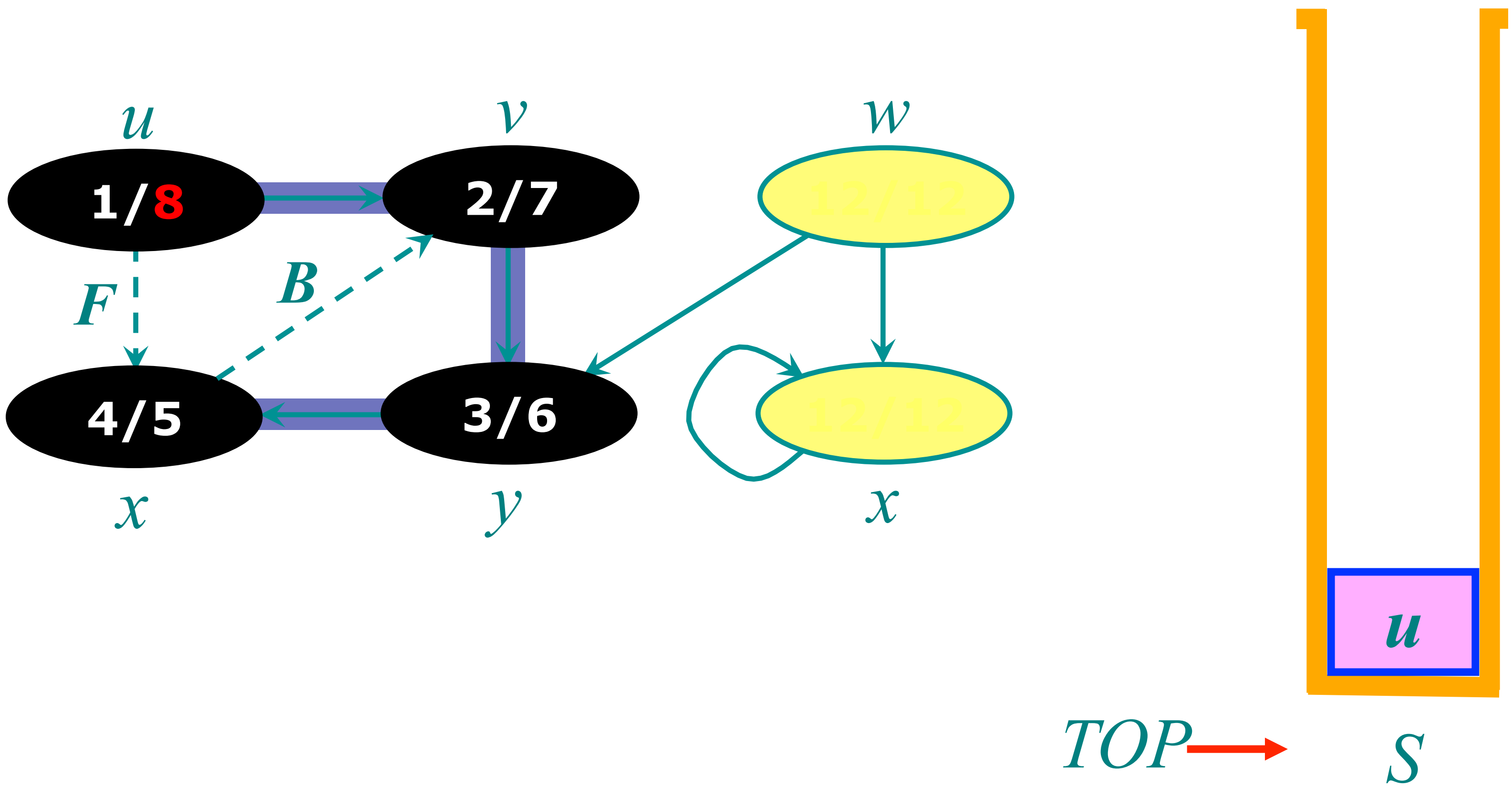
Depth-first search example



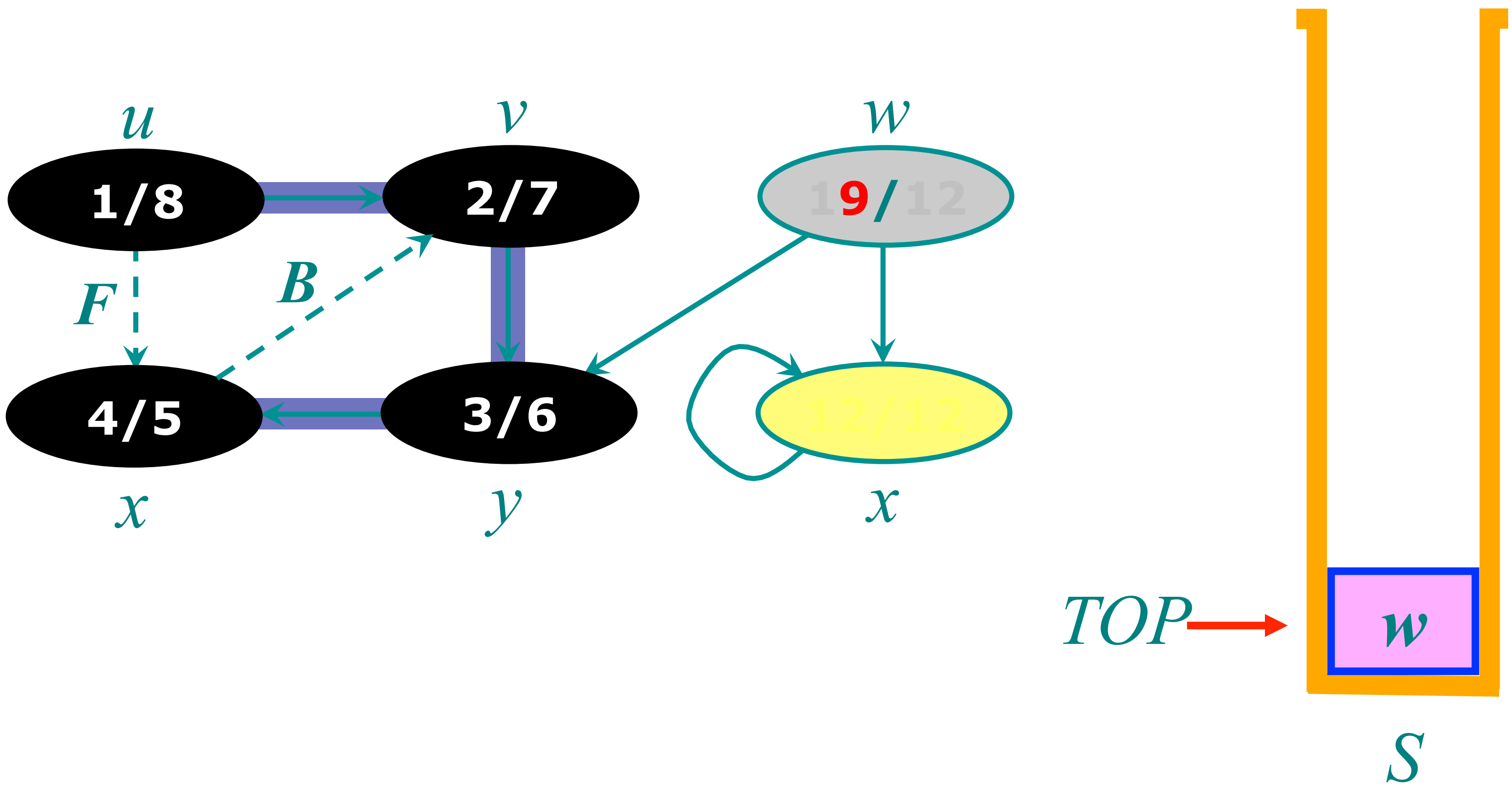
Depth-first search example



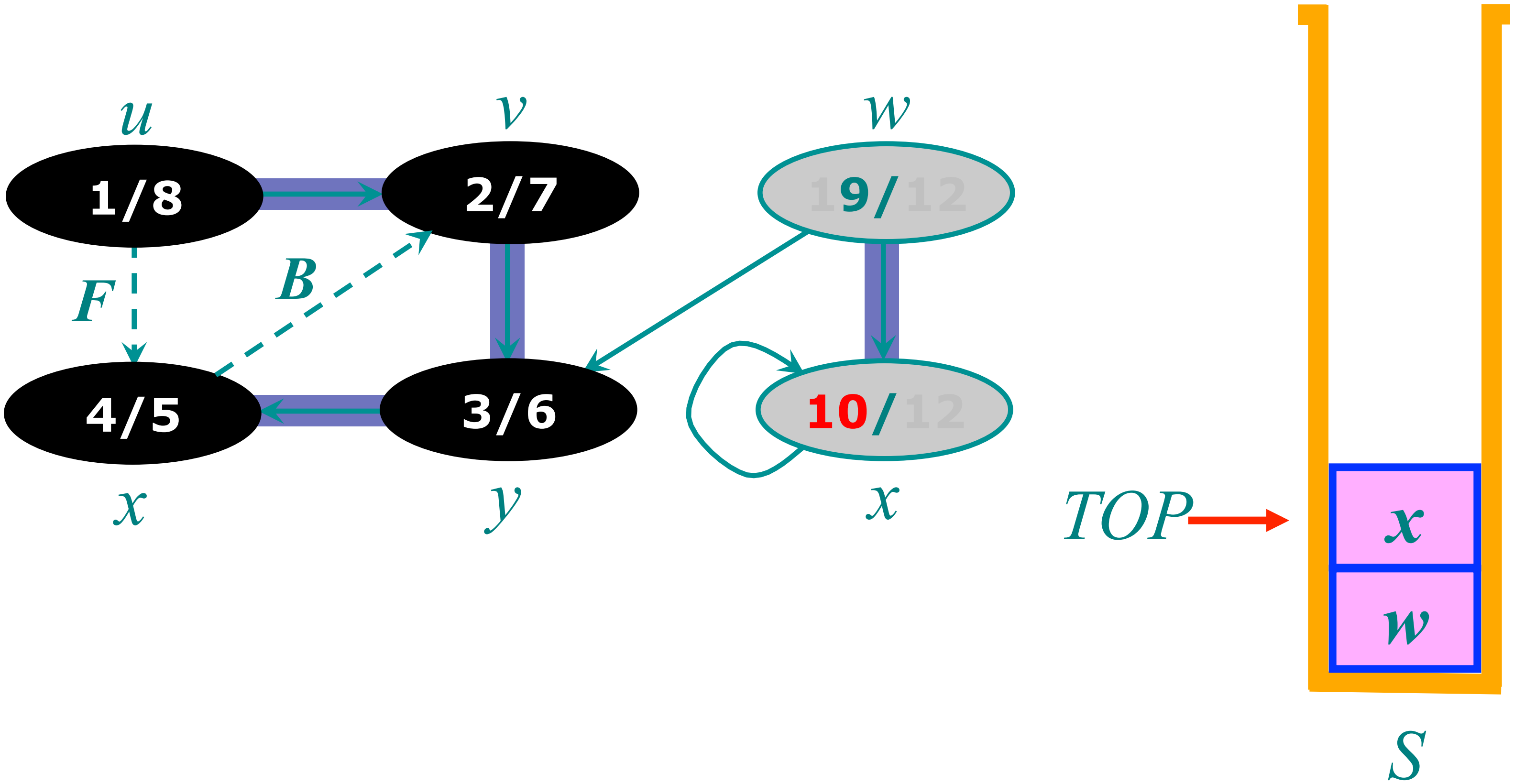
Depth-first search example



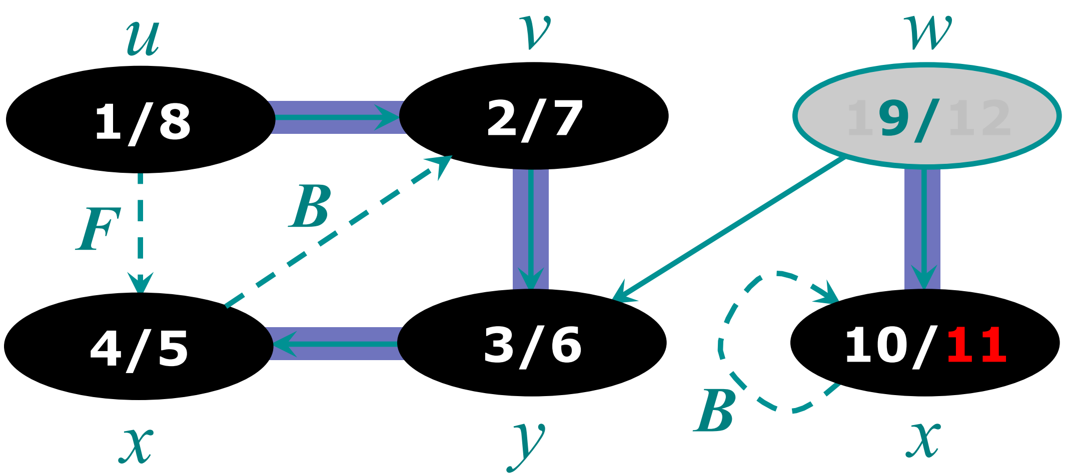
Depth-first search example

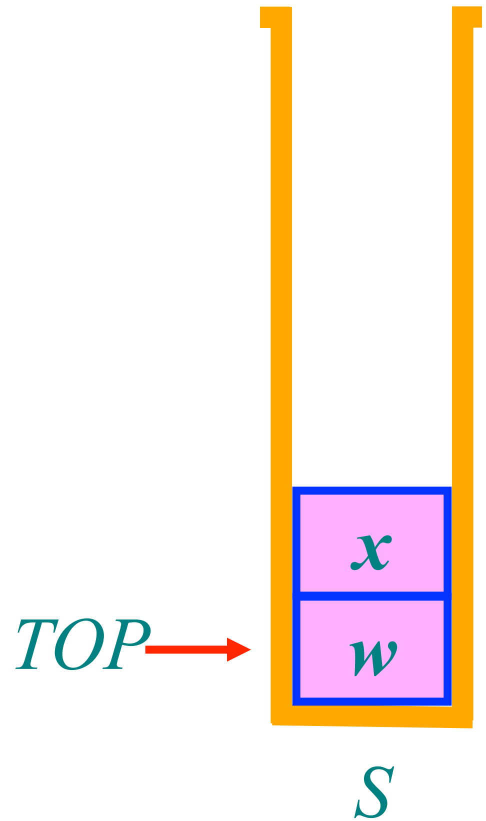


Depth-first search example

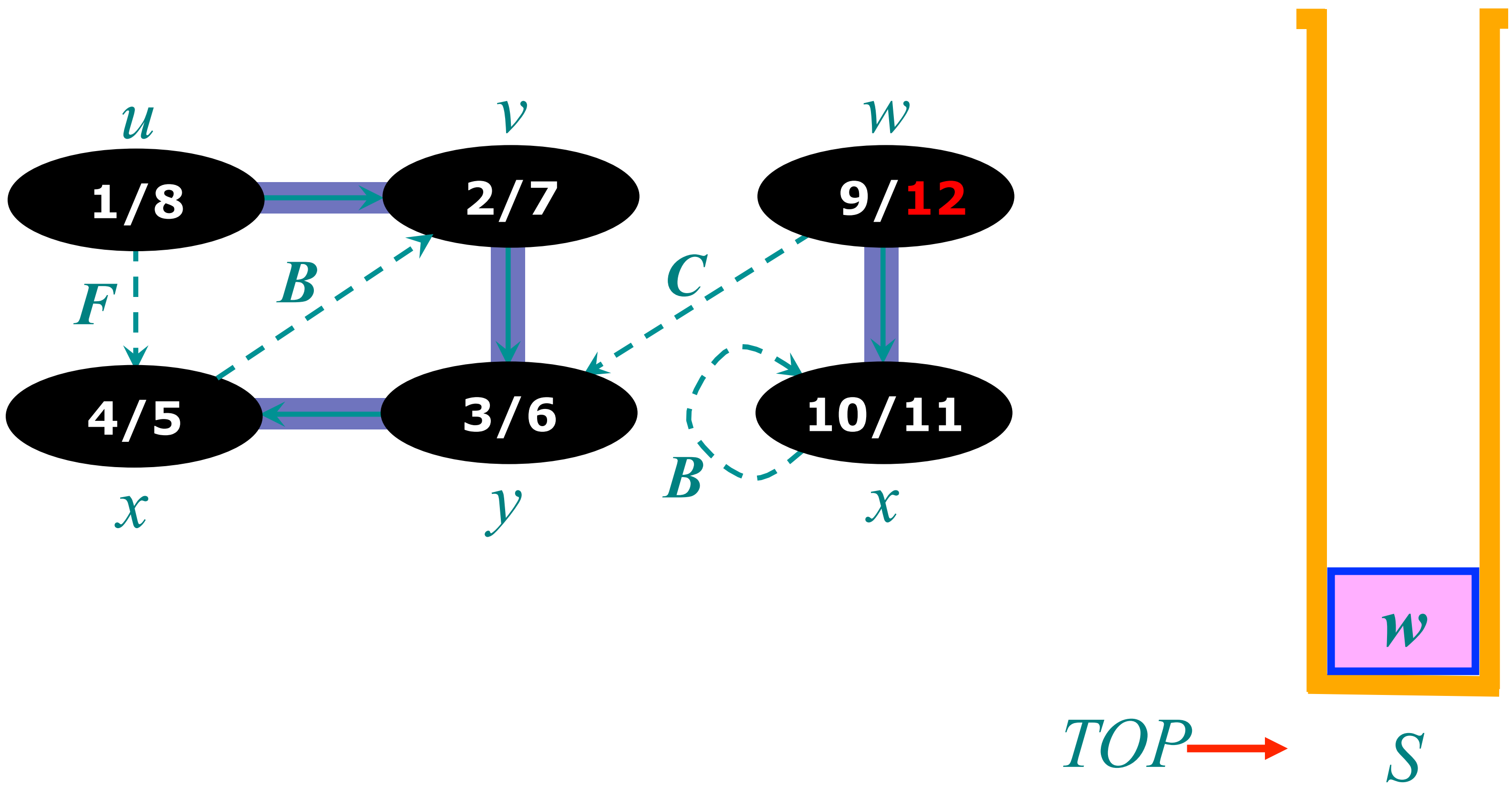


Depth-first search example

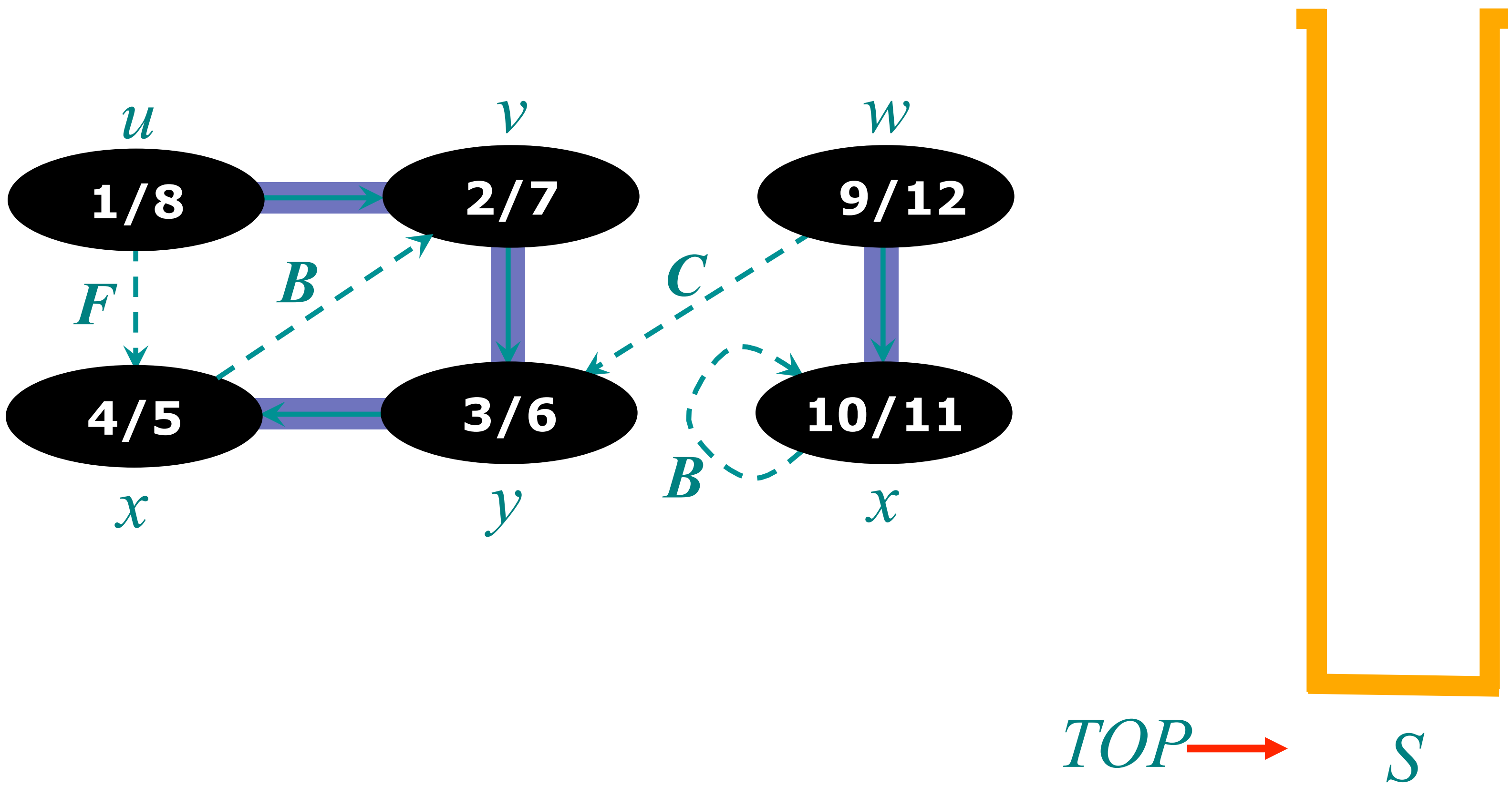




Depth-first search example



Depth-first search example



Depth-first search algorithm

DFS(G)

1. **for** each vertex $u \in V[G]$
2.   **do** $color[u] \leftarrow$ WHITE
3. $\pi[u] \leftarrow$ NIL
4. $time \leftarrow 0$
5. **for** each vertex $u \in V[G]$
6.   **do if** $color[u] =$ WHITE
7.   **then** DFS-VISIT(u)

$O(V)$ times

Depth-first search algorithm

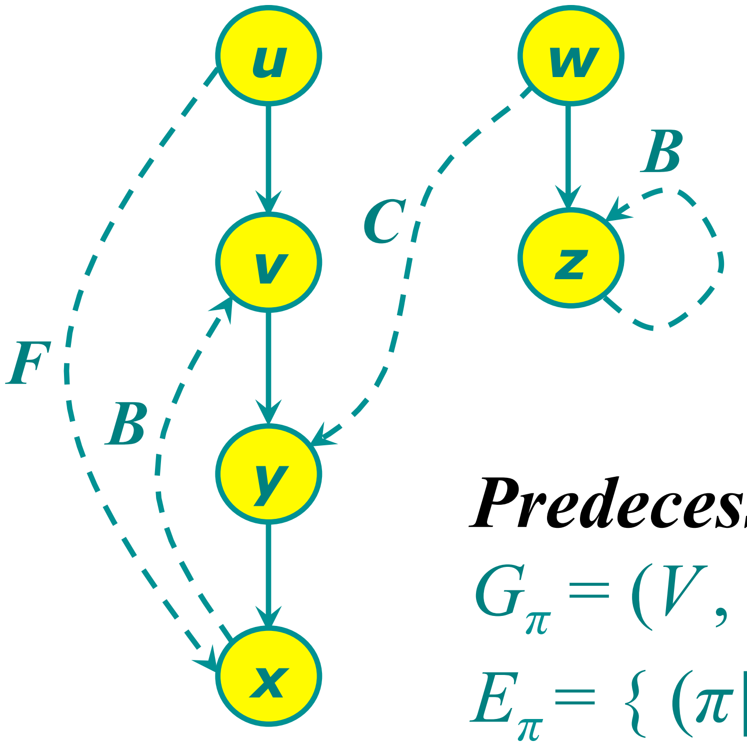
DFS-VISIT(u)

$color[u] \leftarrow \text{GRAY}$
$time \leftarrow time + 1$
$d[u] \leftarrow time$
for each vertex $v \in Adj[u]$
    do if $color[v] = \text{WHITE}$
        then $\pi[v] \leftarrow u$
            DFS-VISIT(v)
$color[u] \leftarrow \text{BLACK}$
$f[u] \leftarrow time \leftarrow time + 1$

$O(E)$ times

Running time is $O(V + E)$

Predecessor subgraph of DFS

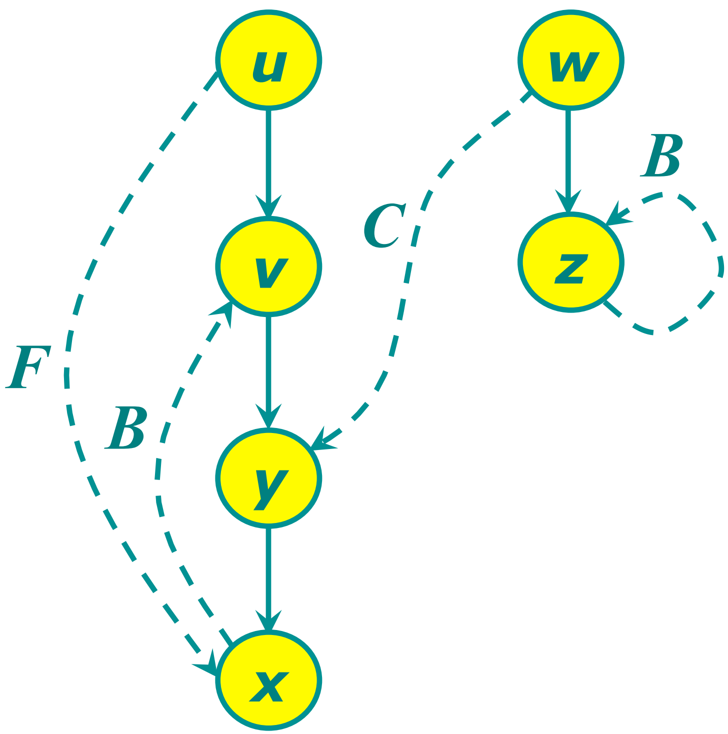


Predecessor subgraph of DFS is

$G_\pi = (V, E_\pi)$, where

$E_\pi = \{ (\pi[v], v) : v \in V$ and $\pi[v] \neq \text{NIL} \}$.

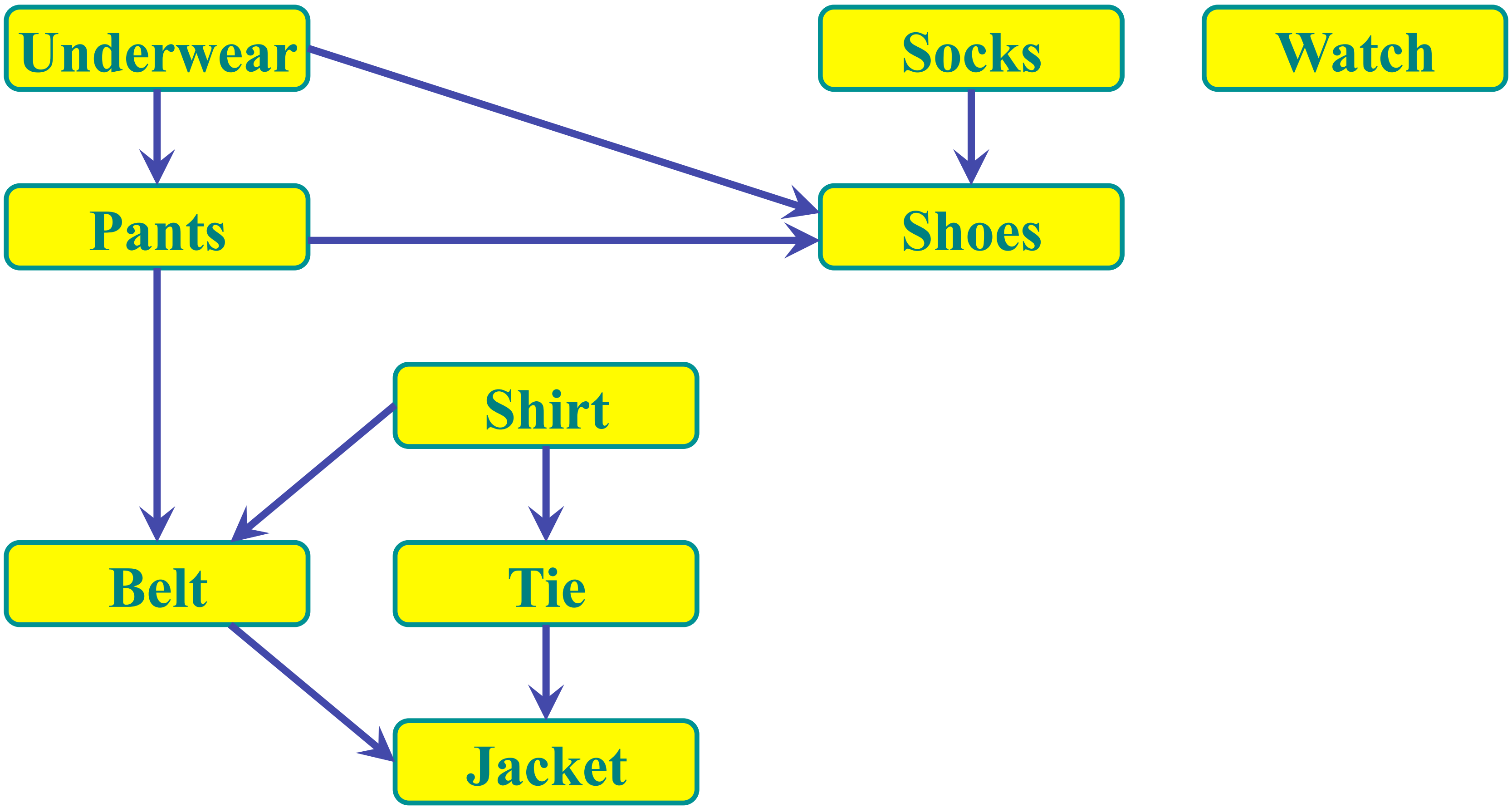
Predecessor subgraph of DFS



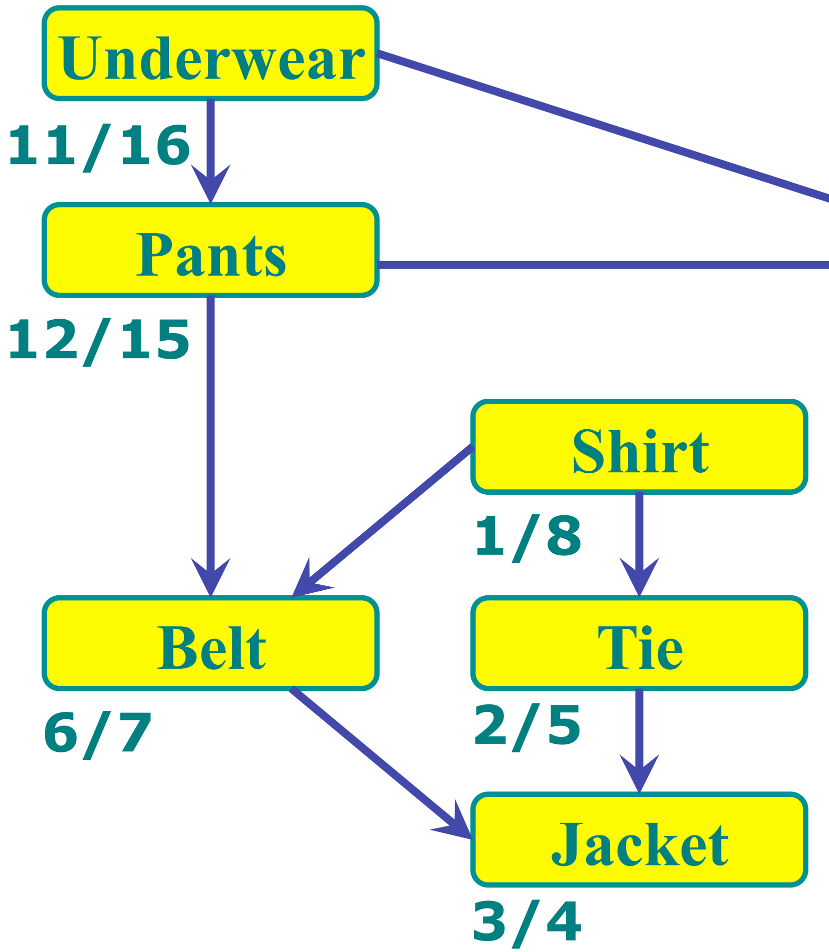
Each edge (u, v) can be classified by the color of the vertex v that is reached when the edge is first explored:

- **WHITE** indicates a tree edge;
- **GRAY** indicates a back edge;
- **BLACK** indicates a forward (if $d[u] < d[v]$) or cross edge (if $d[u] > d[v]$).

Precedence among events

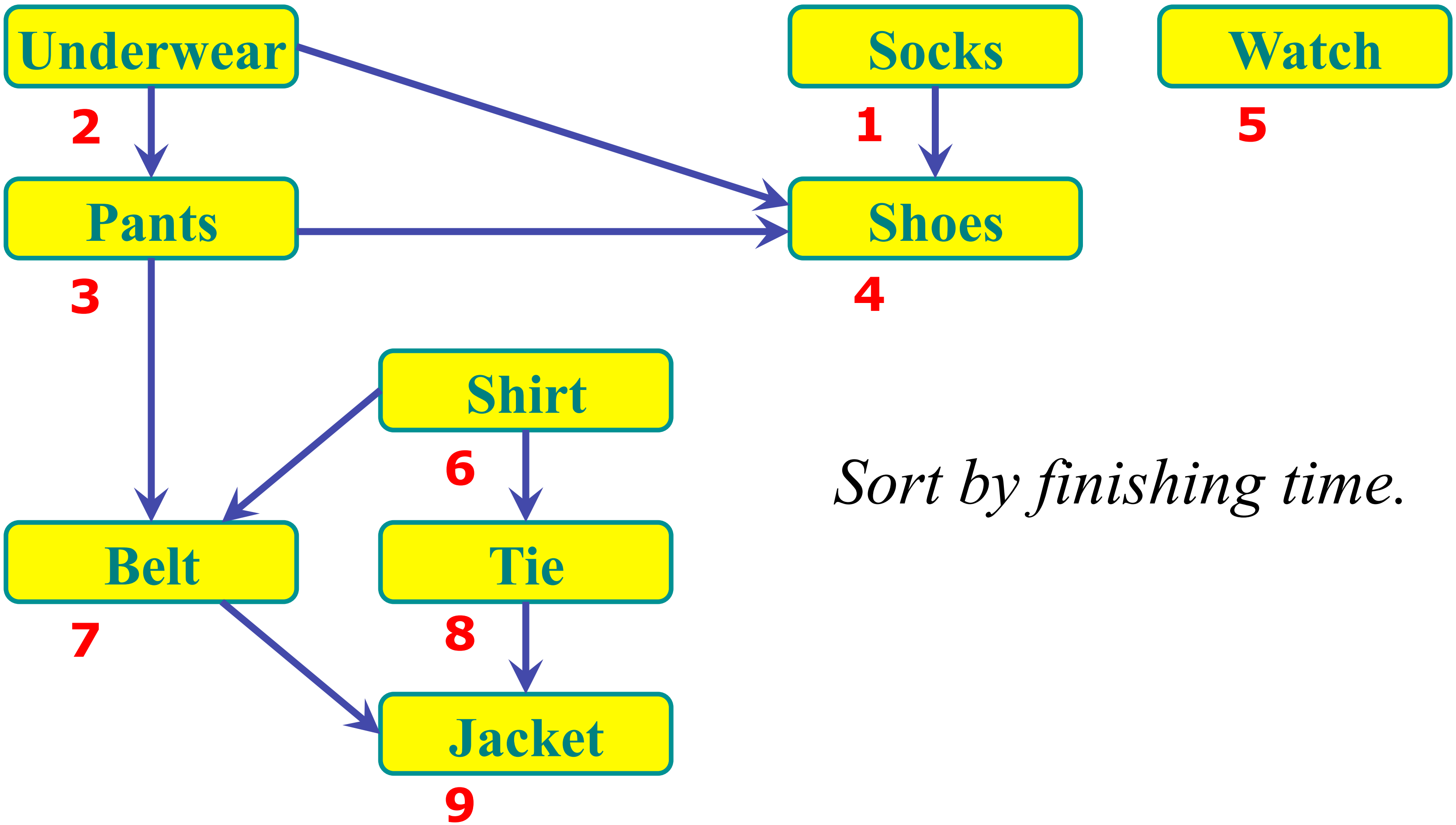


Precedence among events



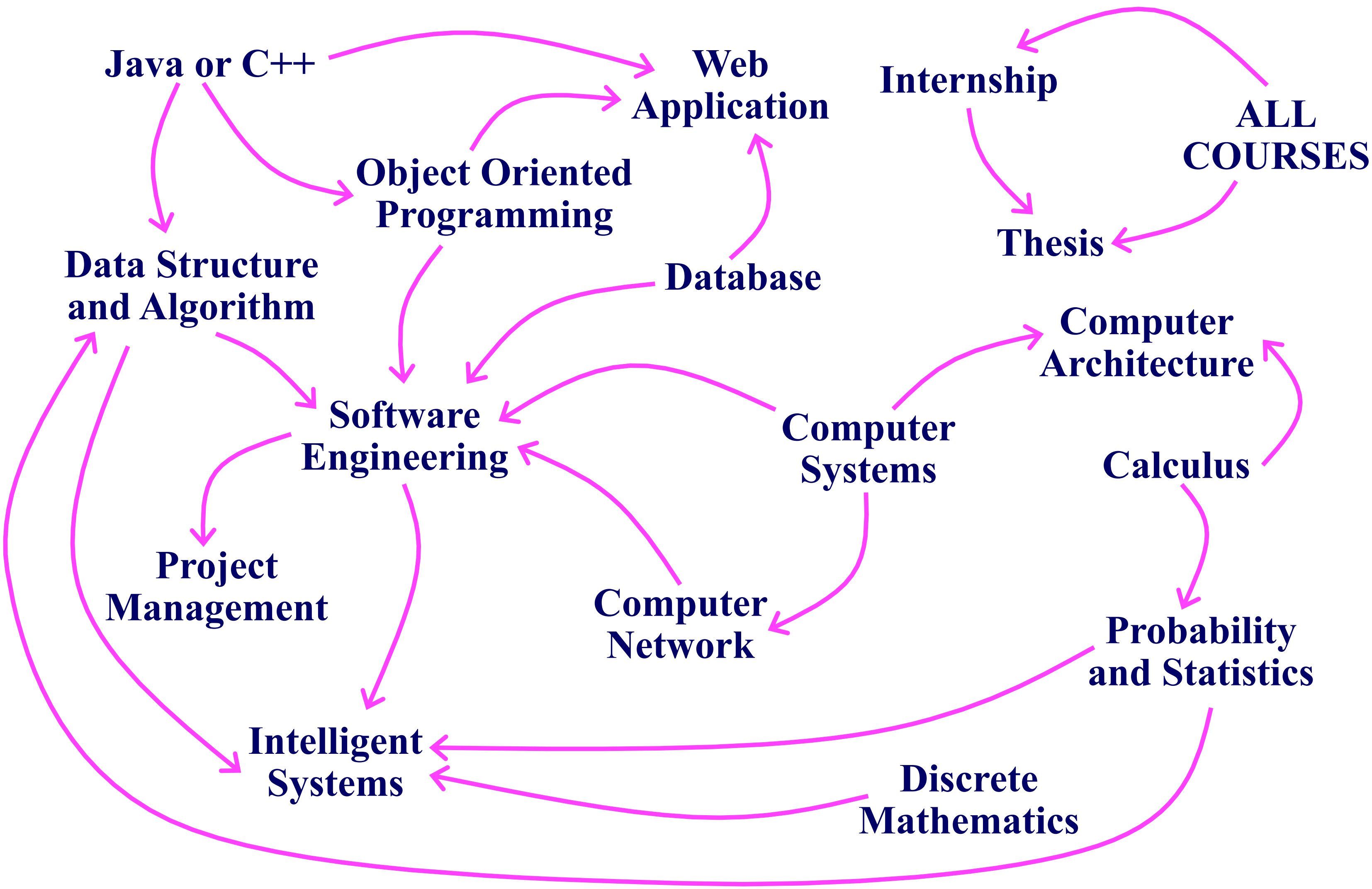
Call depth first search algorithm.

Precedence among events



Sort by finishing time.

Curriculum



Java or C++
Web Application
Internship
ALL COURSES
Object Oriented Programming
Thesis
Data Structure and Algorithm
Database
Computer Architecture
Software Engineering
Computer Systems
Calculus
Project Management
Computer Network
Probability and Statistics
Intelligent Systems
Discrete Mathematics

Topological sort

TOPOLOGICAL-SORT(G)

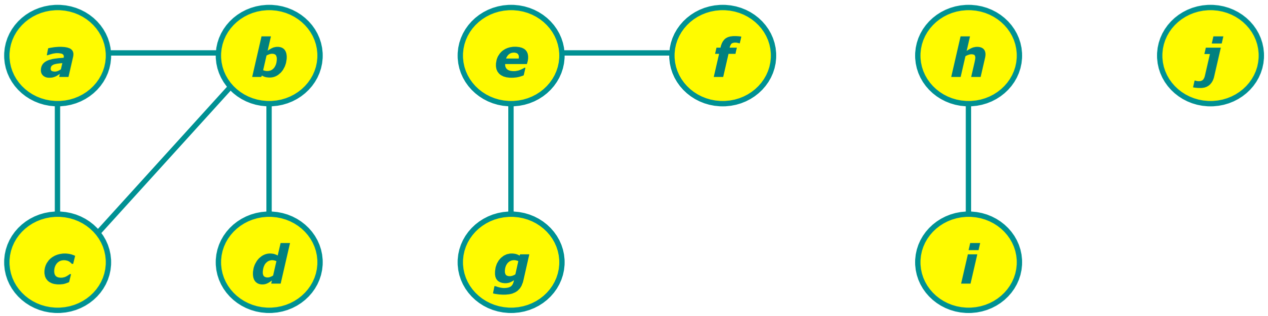
1. call DFS(G) to compute finishing times $f[v]$ for each vertex v.
2. as each vertex is finished, insert it onto the front of a linked list.
3. **return** the linked list of vertices.

Topological sort

A ***topological sort*** of a directed acyclic graph or "***dag***" $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.

- Topological sort of a graph can be viewed as an ***linear ordering*** of its vertices.
- Topological sorting is ***different*** from the usual kind of "sorting" studied before.
- If the graph is ***not acyclic***, then no linear ordering is possible.

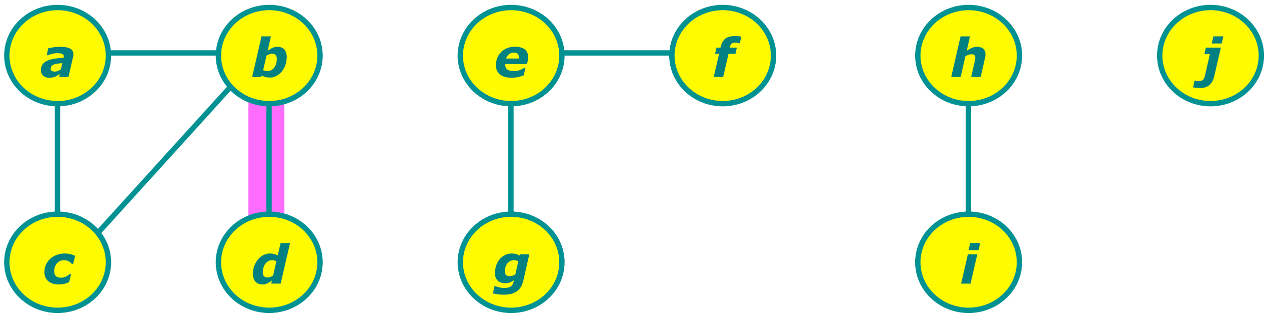
Disjoint sets



Connected components

Edge processed	Collection of disjoint sets
initial sets	$\{a\}$ $\{b\}$ $\{c\}$ $\{d\}$ $\{e\}$ $\{f\}$ $\{g\}$ $\{h\}$ $\{i\}$ $\{j\}$

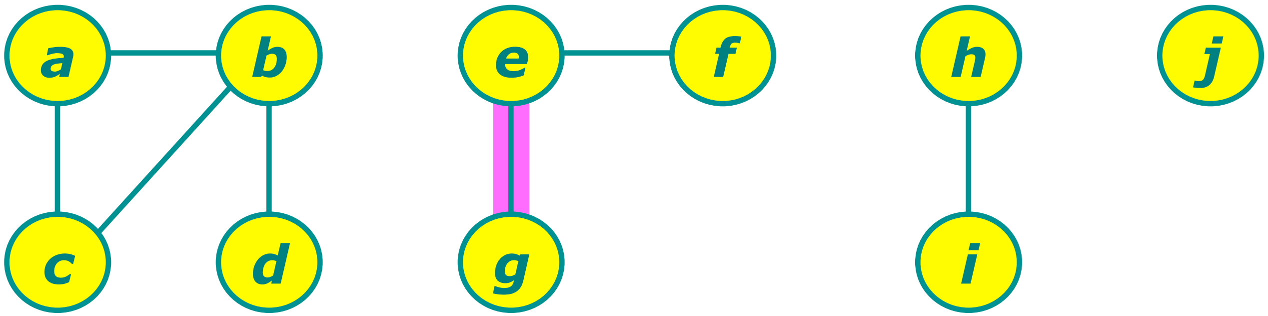
Disjoint sets



Connected components

Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b, d)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$

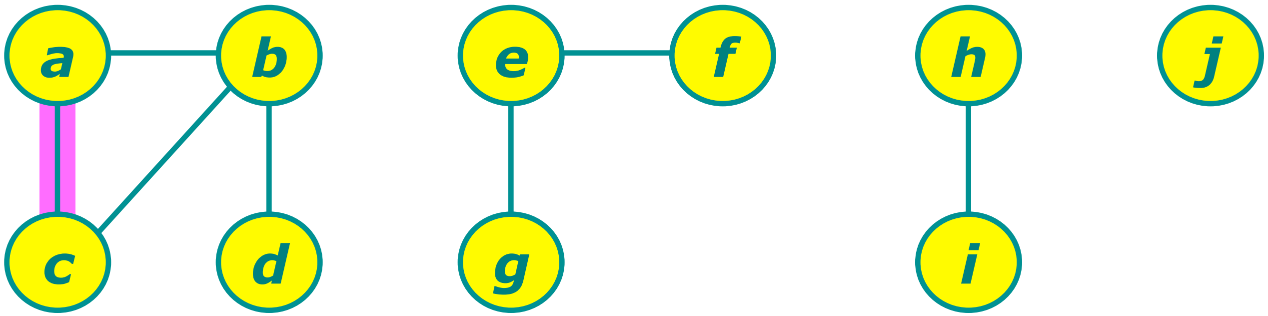
Disjoint sets



Connected components

Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b, d)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e, g)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$

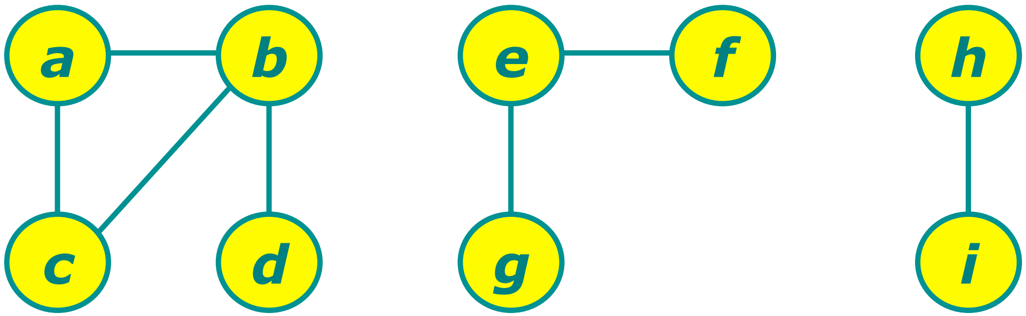
Disjoint sets



Connected components

Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b, d)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e, g)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a, c)	$\{a, c\}$	$\{b, d\}$			$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$

Disjoint sets



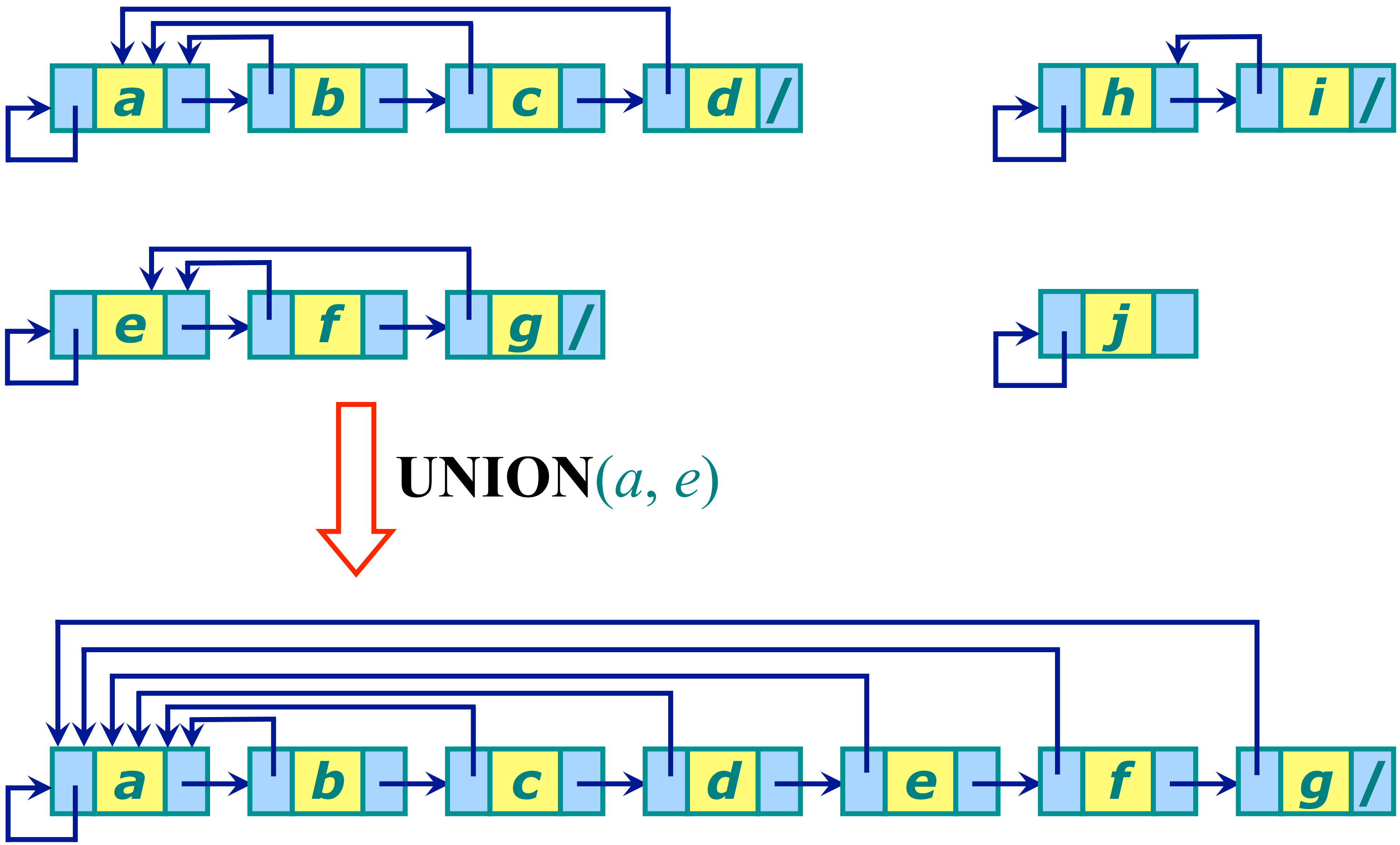
Connected components

Edge processed	**Collection of disjoint sets**									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b, d)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e, g)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a, c)	$\{a, c\}$	$\{b, d\}$			$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h, i)	$\{a, c\}$	$\{b, d\}$			$\{e, g\}$	$\{f\}$		$\{h, i\}$		$\{j\}$
(a, b)	$\{a, b, c, d\}$				$\{e, g\}$	$\{f\}$		$\{h, i\}$		$\{j\}$
(e, f)	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$
(b, c)	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$

Disjoint set operations

- **MAKE-SET**(x) creates a new set whose only member is x.
- **UNION**(x, y) unites the dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets. The two sets are assumed to be disjoint prior to the operation.
- **FIND-SET**(x) returns a pointer to the representative of the unique set containing x.

Linked list representation of disjoint sets



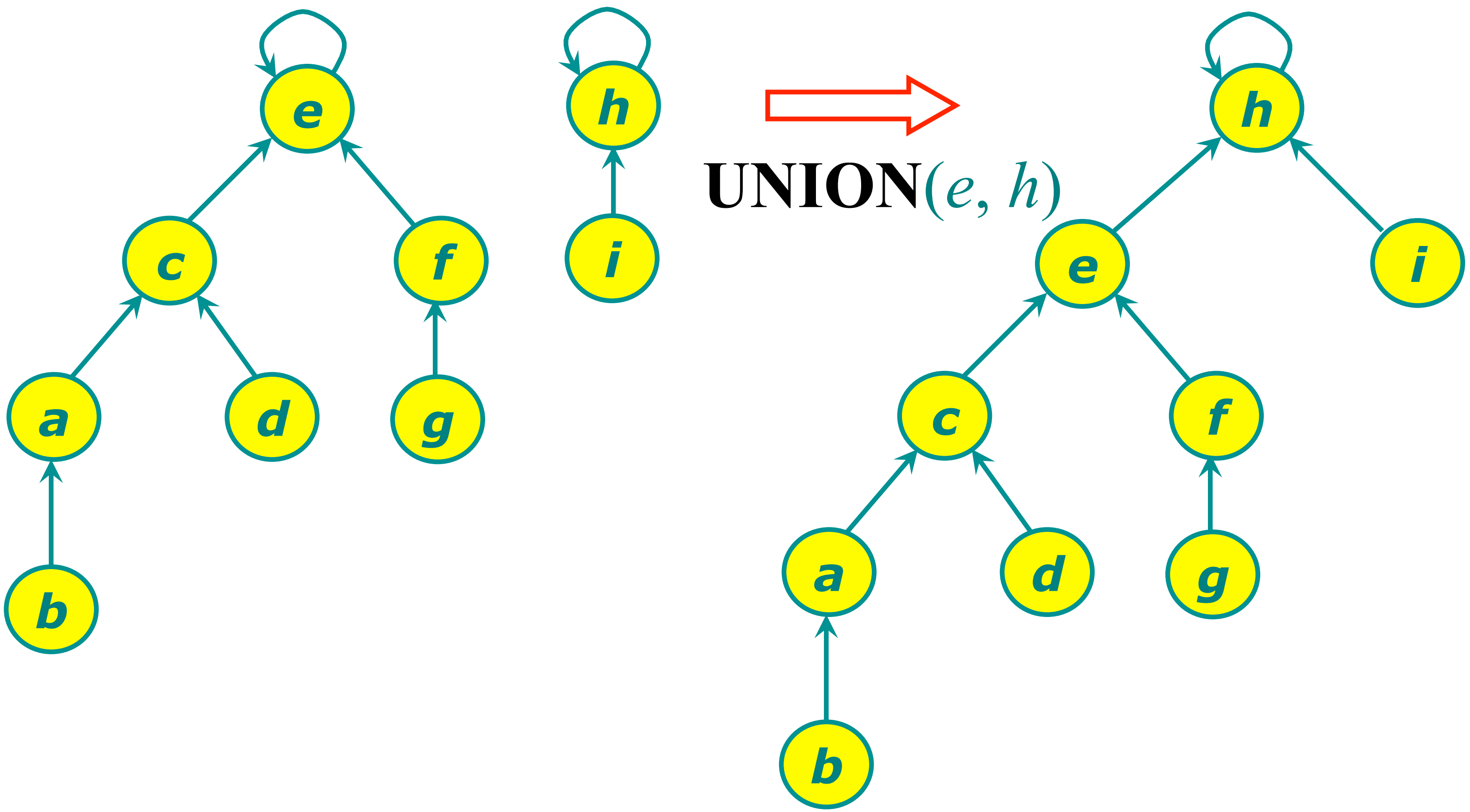
Analysis of linked list representation

Linked list representation of the **UNION** operation requires an average of $\Phi(n)$ time per call because we may be appending a longer list onto a shorter list.

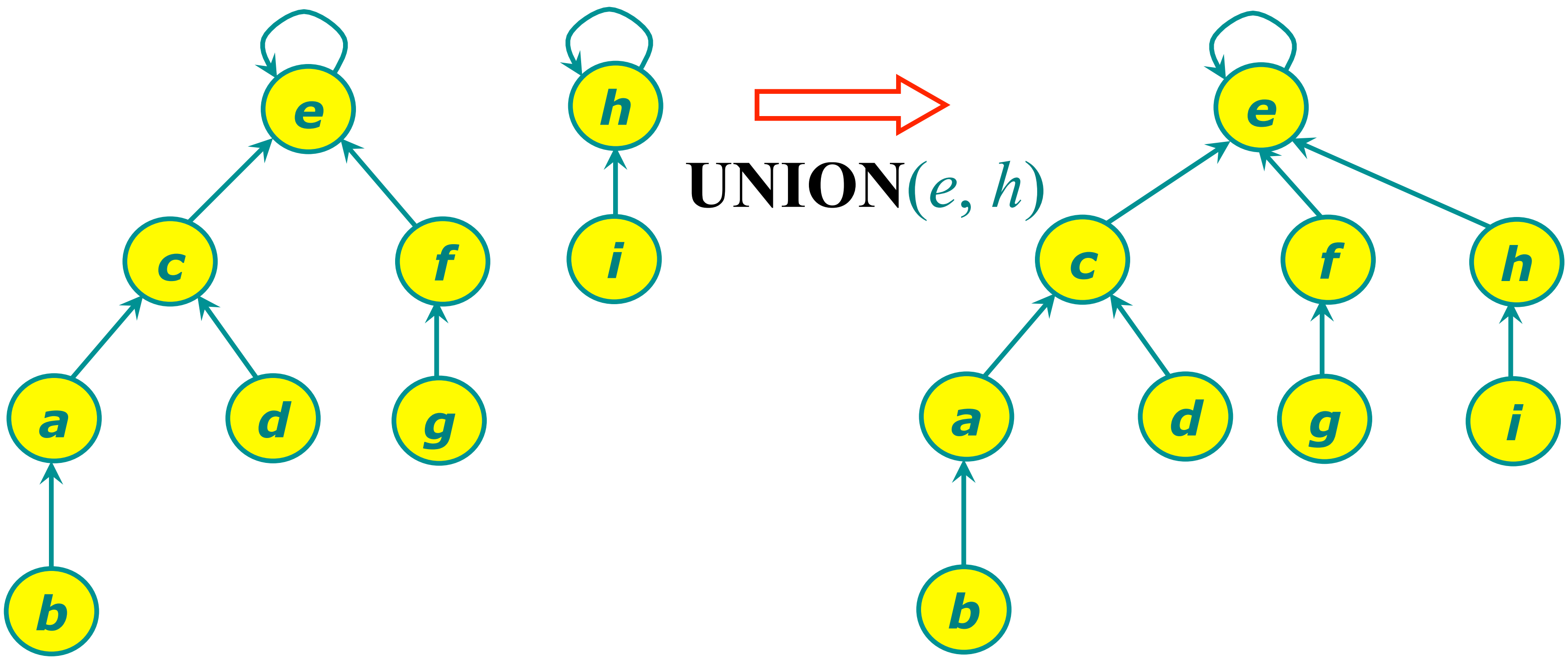
Weighted-union heuristic: Suppose that each list also includes the length of the list and that we always append the smaller list onto the longer.

- A sequence of m **MAKE-SET**, **UNION**, and **FIND-SET** operations, n of which are **MAKE-SET** operations, takes $O(m + nlgn)$ time.

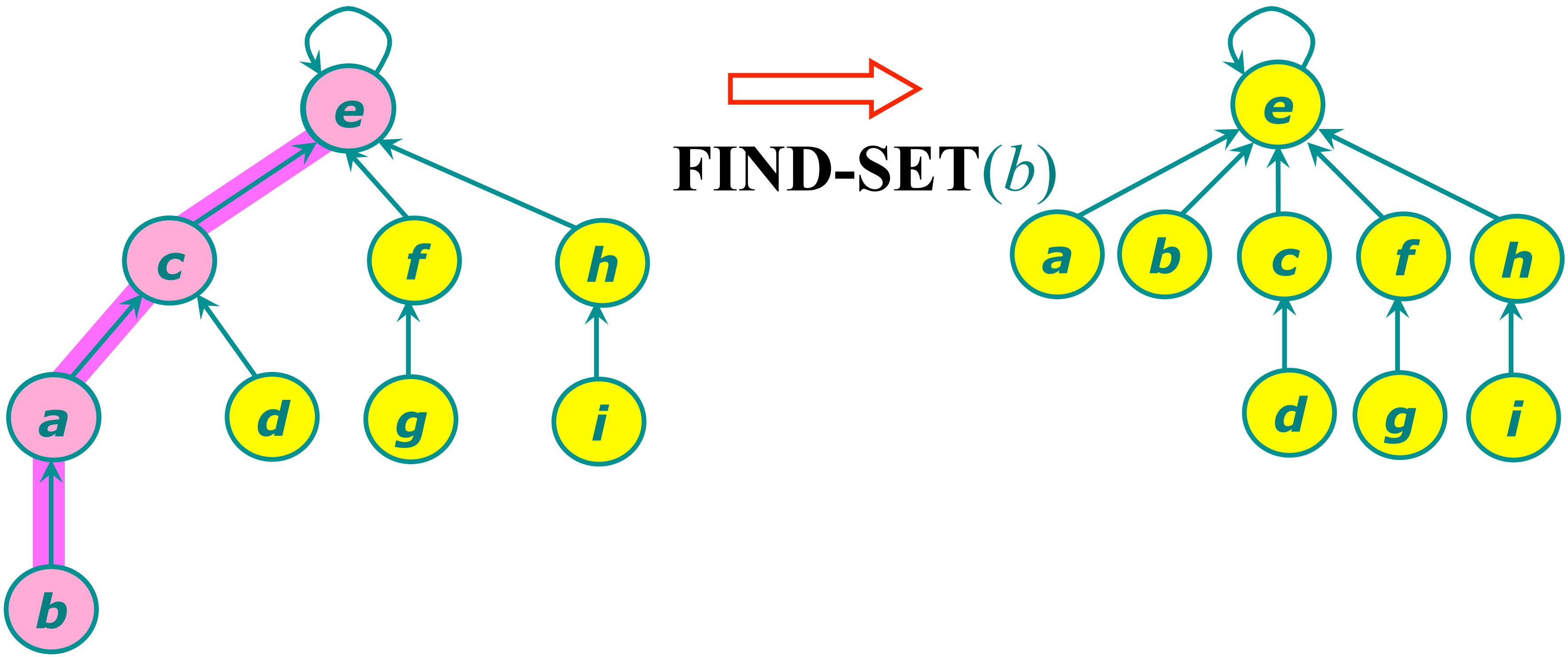
Disjoint set forests



Union by rank



Path compression



Disjoint set forests

MAKE-SET(x)

1. $p[x] \leftarrow x$
2. $rank[x] \leftarrow 0$

FIND-SET(x)

1. **if** $x \neq p[x]$
2. **then** $p[x] \leftarrow$ FIND-SET$(p[x])$
3. **return** $p[x]$

Disjoint set forests

UNION(x, y)

1. LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)

1. **if** $rank[x] > rank[y]$
2.   **then** $p[y] \leftarrow x$
3.   **else** $p[x] \leftarrow y$
4.   **if** $rank[x] = rank[y]$
5.   **then** $rank[y] \leftarrow rank[y] + 1$

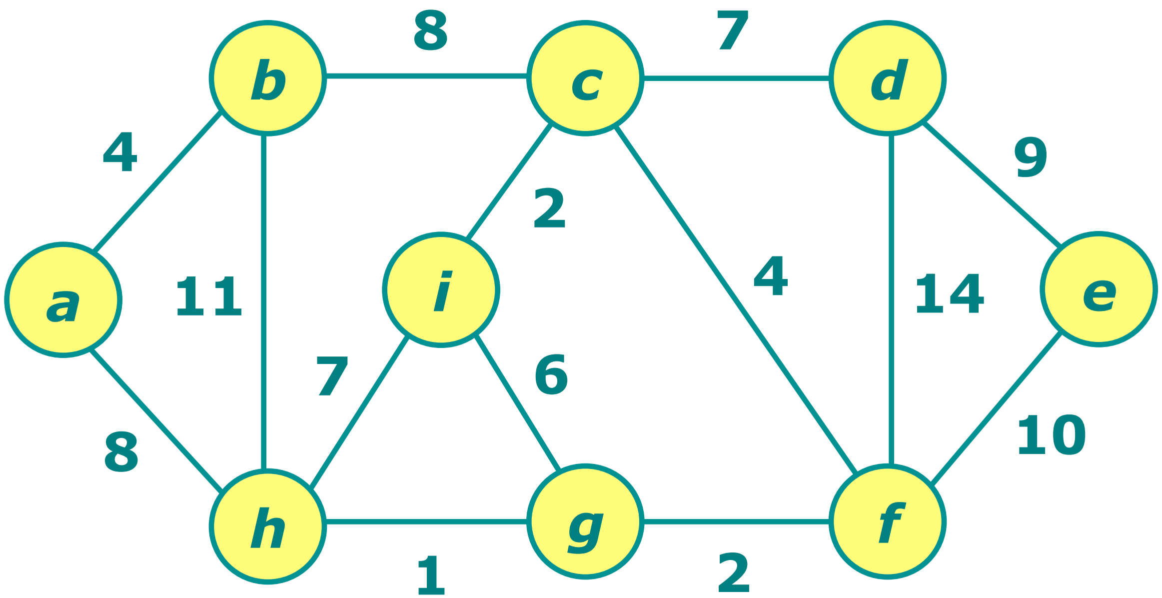
Union by rank and path compression

When we use both ***union by rank*** and ***path compression***, the worst-case running time is $O(m\,\alpha(n))$, where $\alpha(n)$ is a very ***slowly*** growing function.

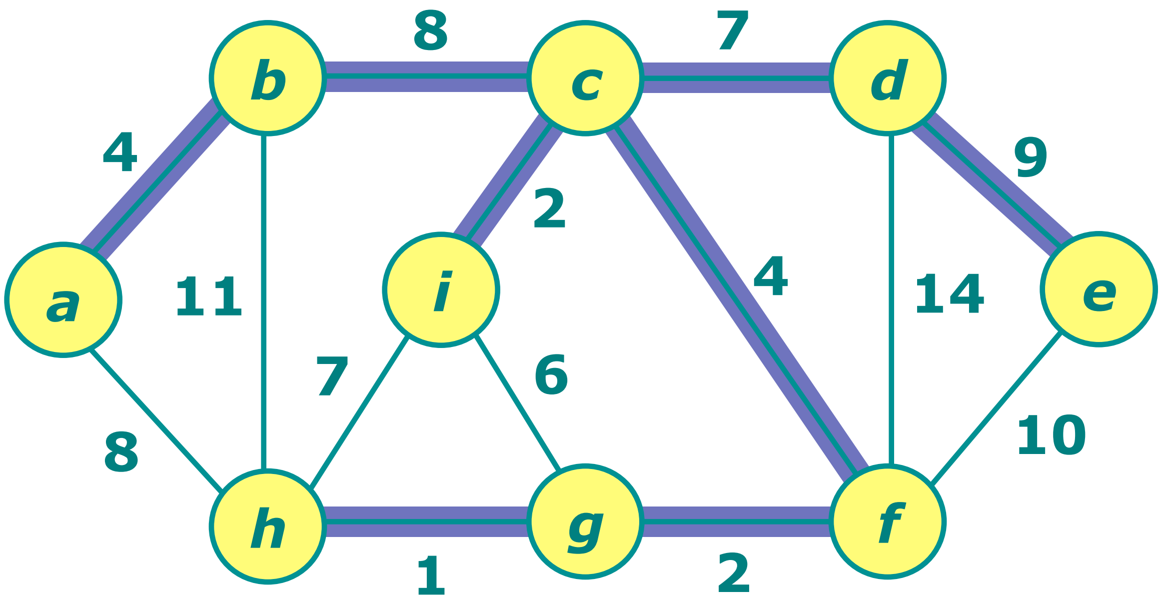
$$\alpha(n) \begin{cases} 0 & \text{for } 0 \leq n \leq 2, \\ 1 & \text{for } n = 3, \\ 2 & \text{for } 4 \leq n \leq 7, \\ 3 & \text{for } 8 \leq n \leq 2047, \\ 4 & \text{for } 2047 \leq n \leq A_4(1) >> 10^{80}. \end{cases}$$

$$A_k(j) \begin{cases} j + 1 & \text{if } k = 0, \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1. \end{cases}$$

Minimum spanning tree



Minimum spanning tree



Total weight = 1 + 2 + 2 + 4 + 4 + 7 + 8 + 9 = 37

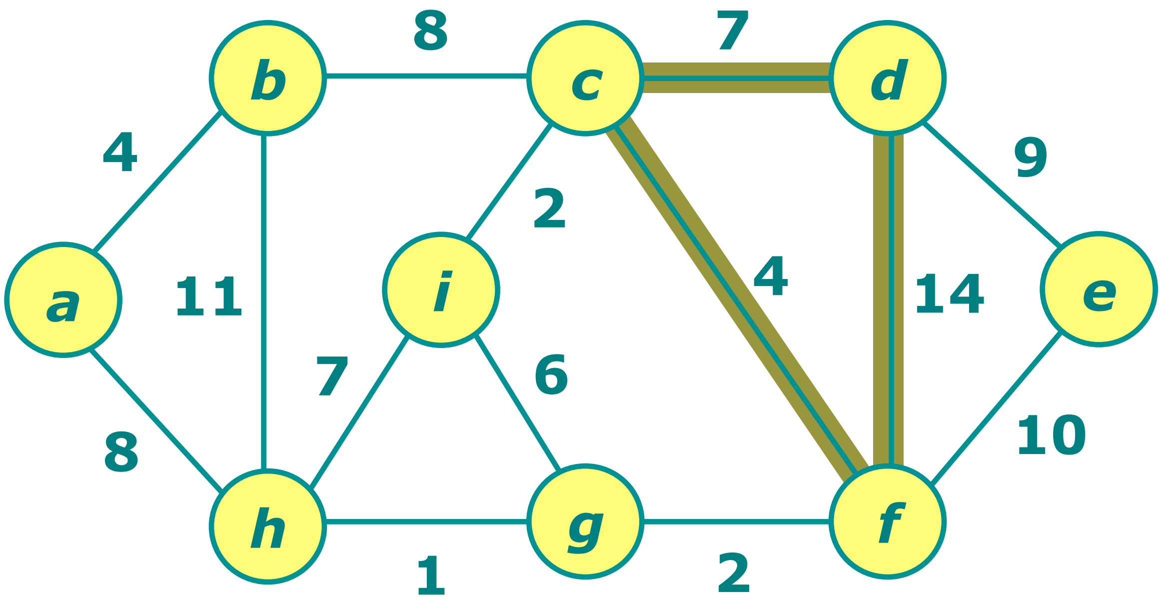
Minimum spanning tree

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

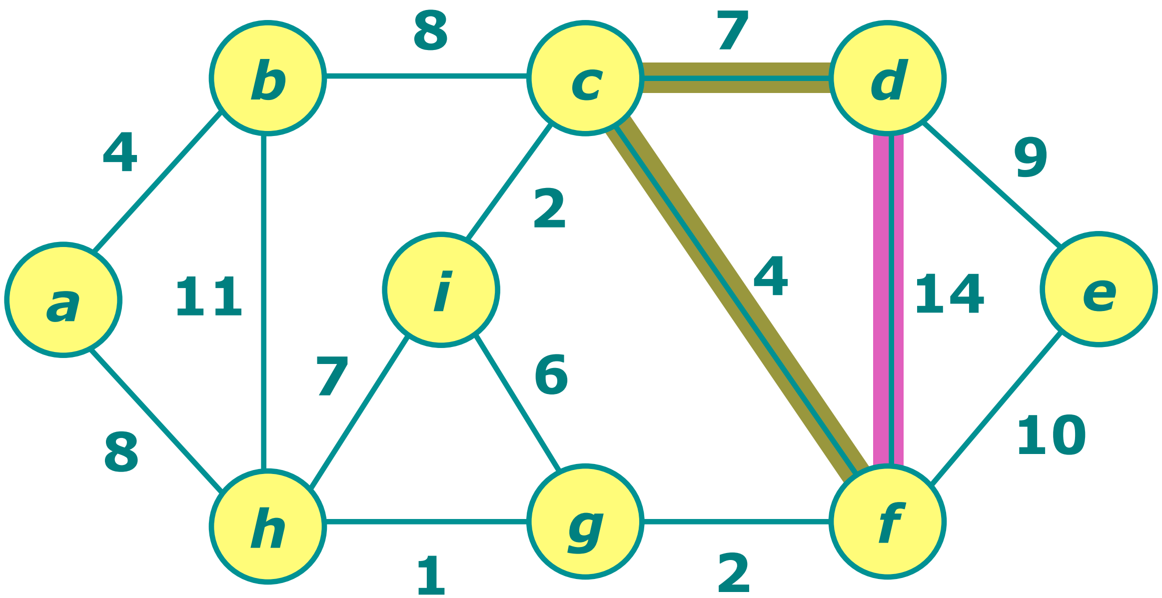
Output: A ***spanning tree*** T — a tree that connects all vertices — of minimum weight:

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

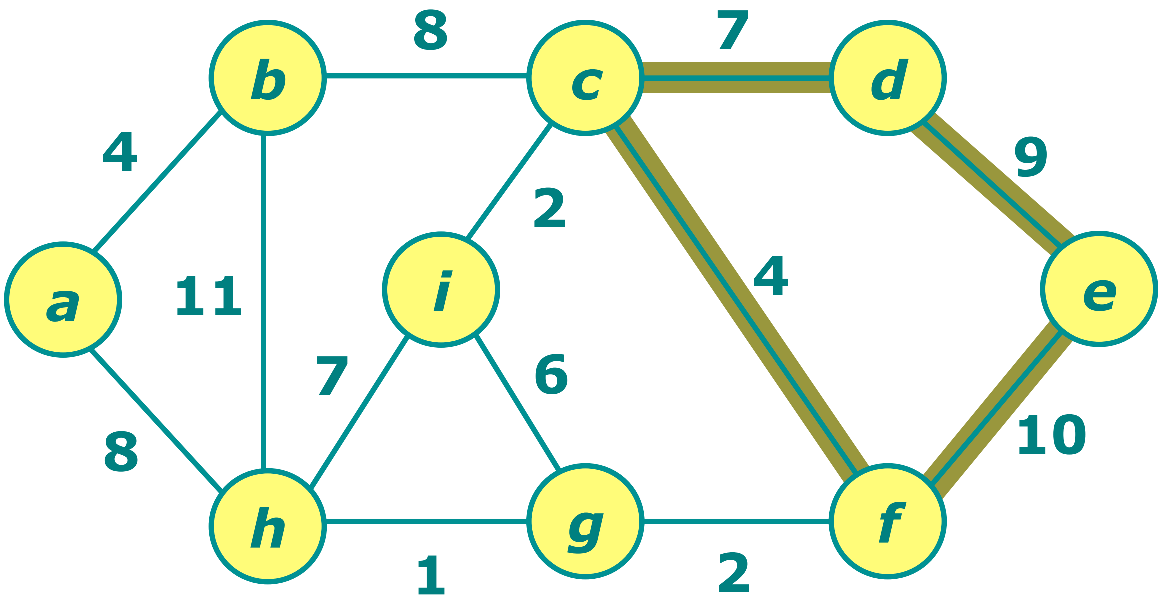
Destroy cycles



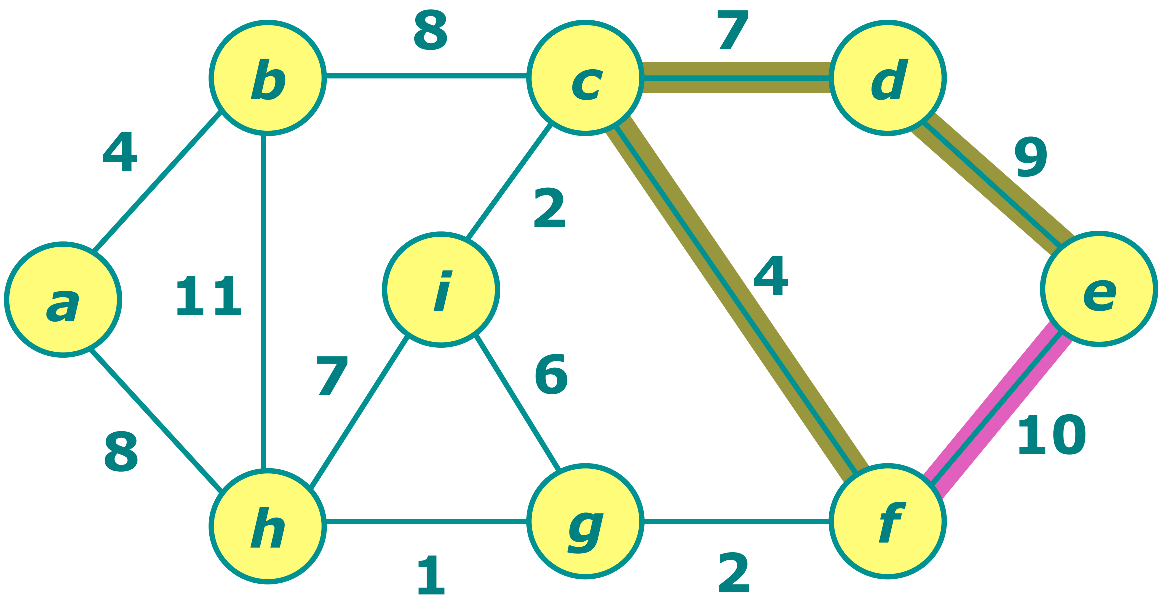
Destroy cycles



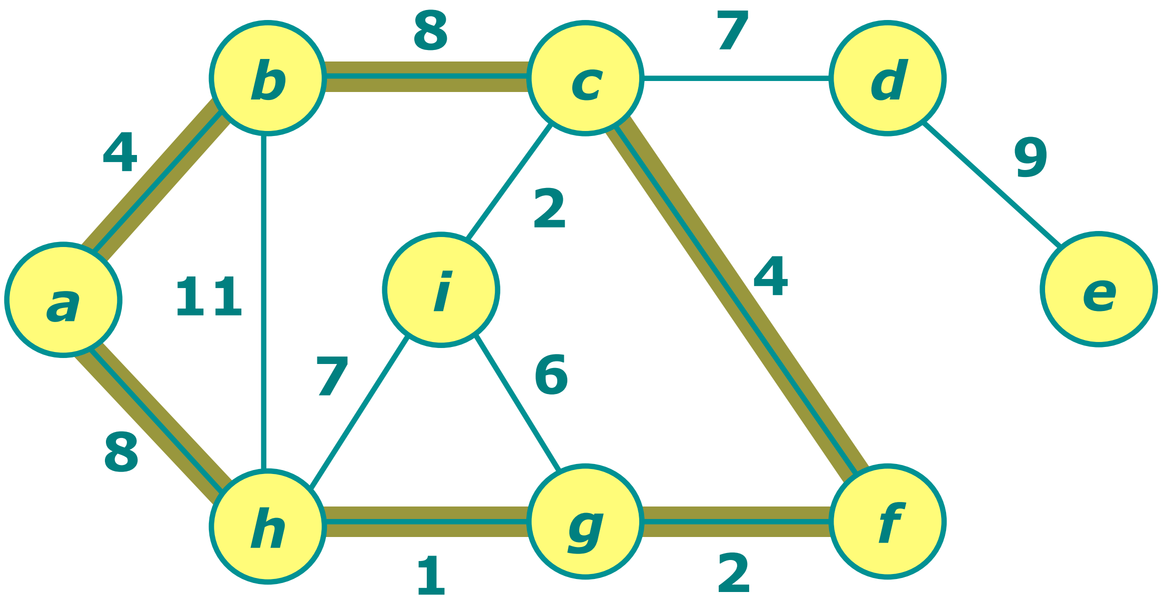
Destroy cycles



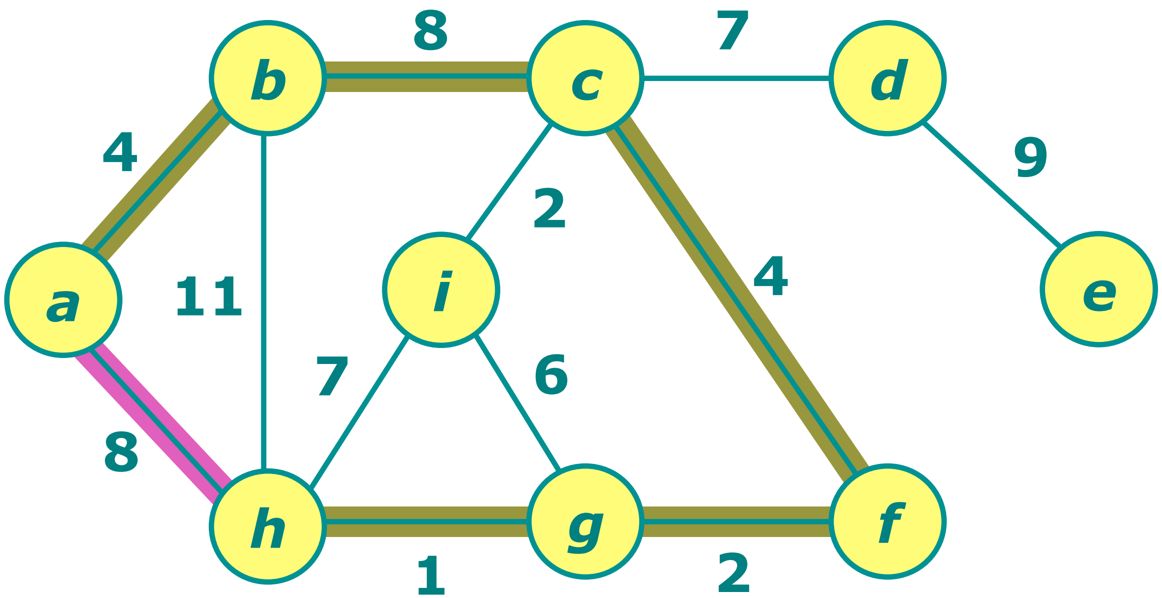
Destroy cycles



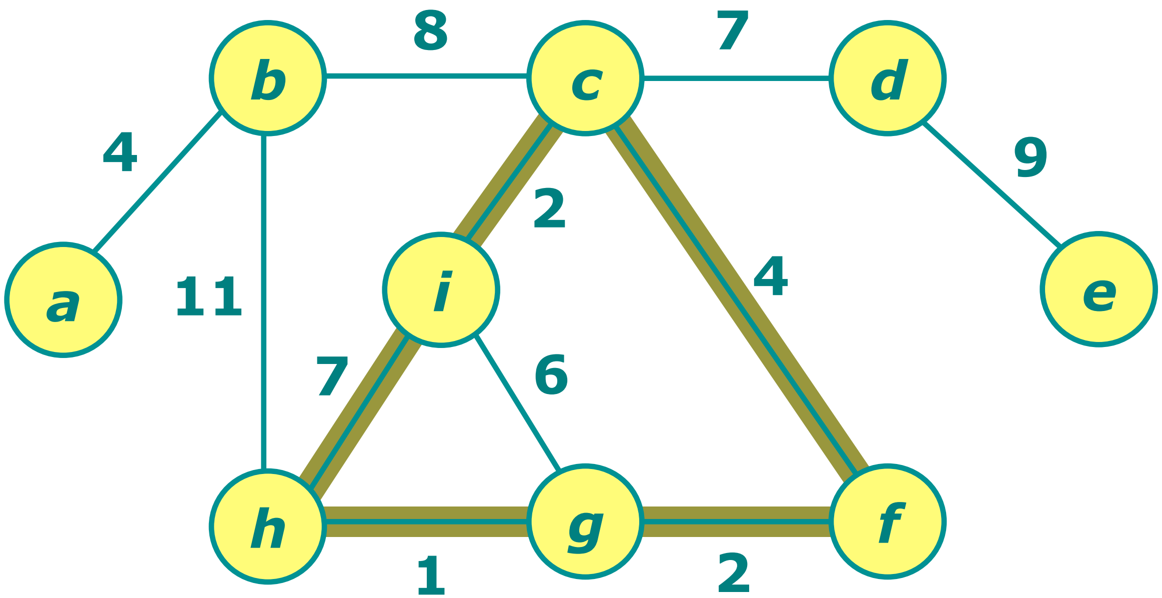
Destroy cycles



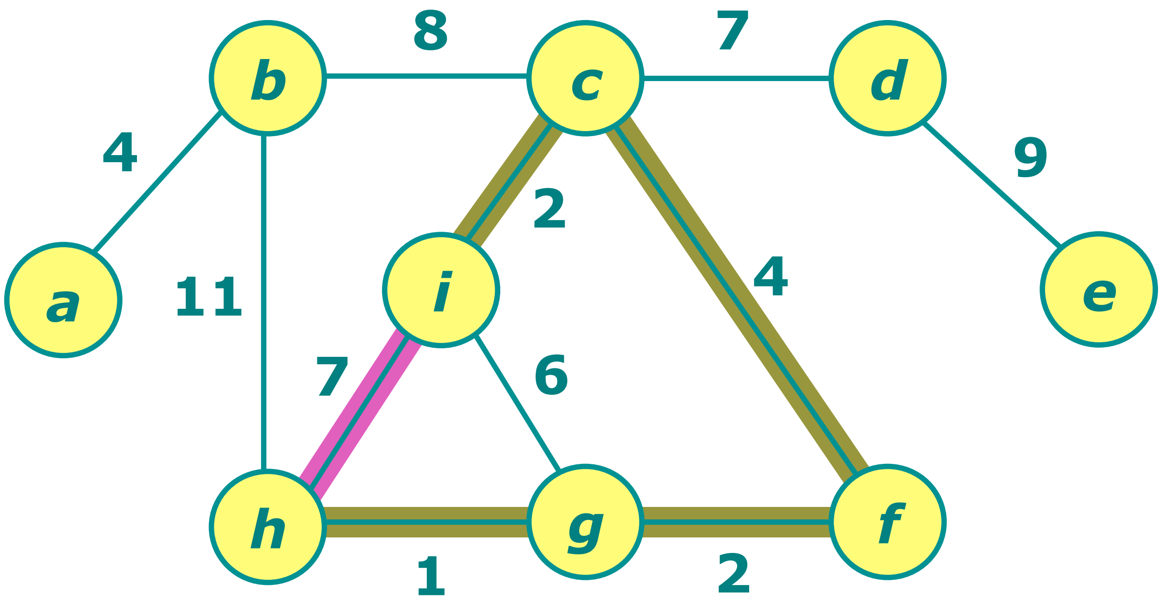
Destroy cycles



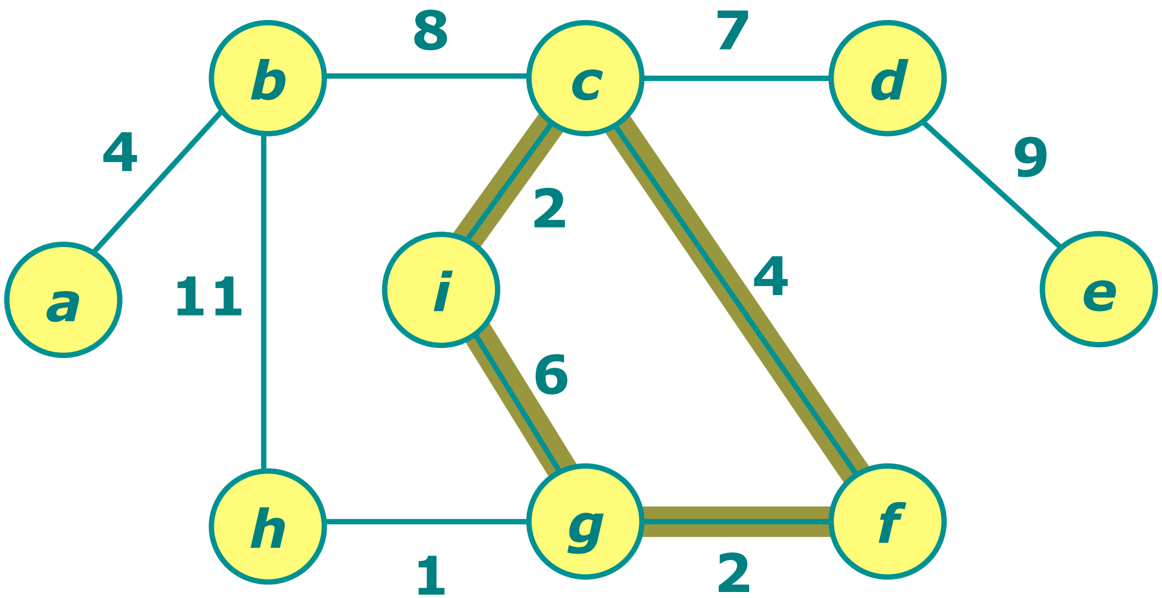
Destroy cycles



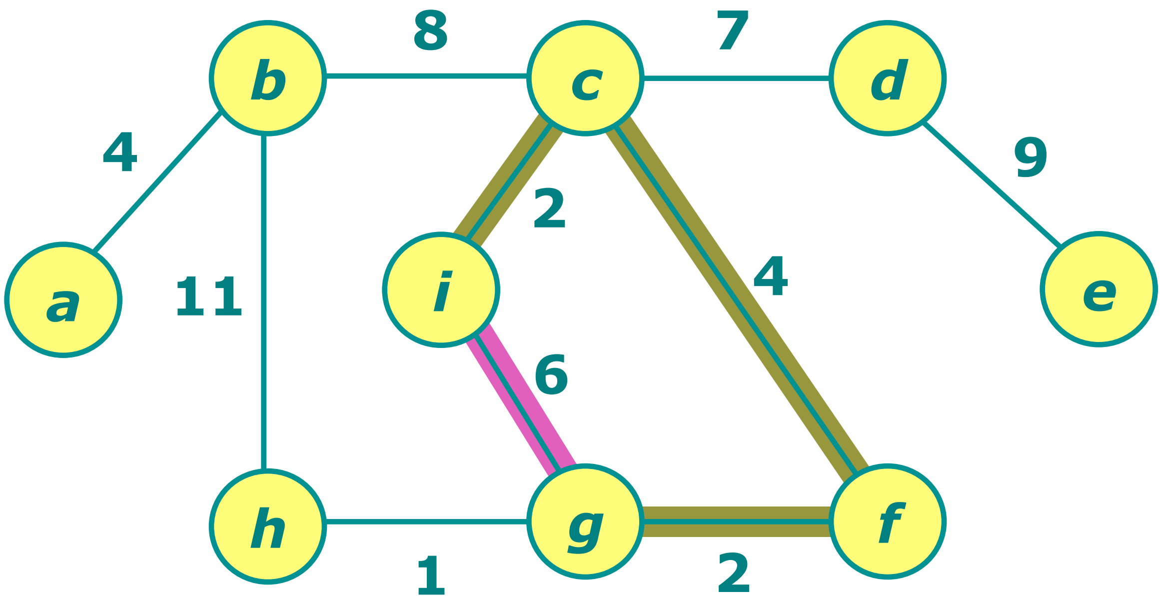
Destroy cycles



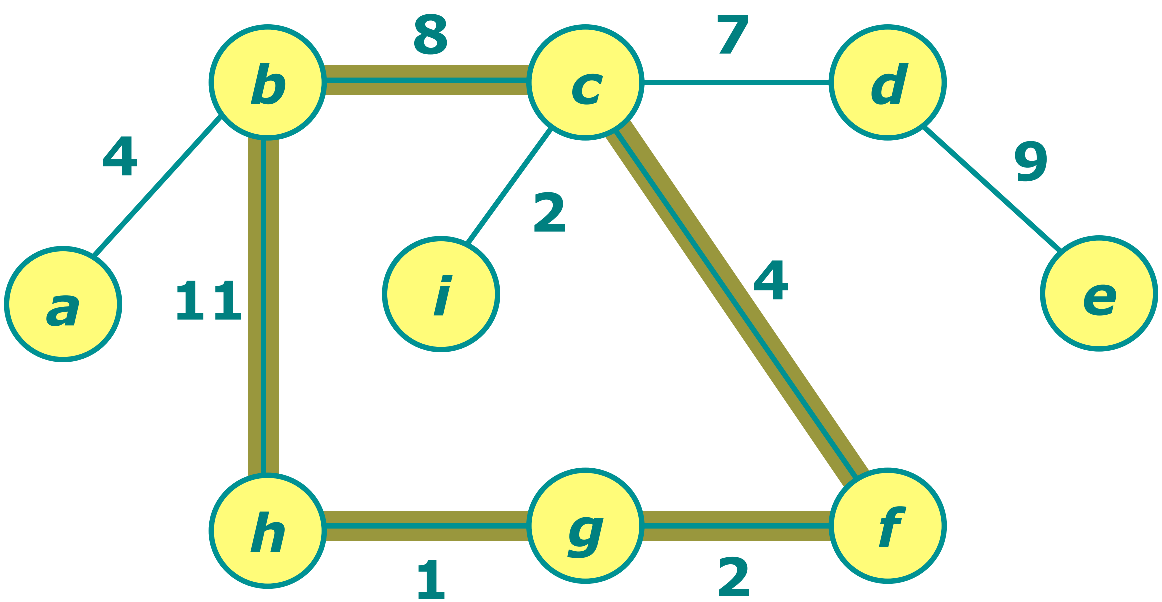
Destroy cycles



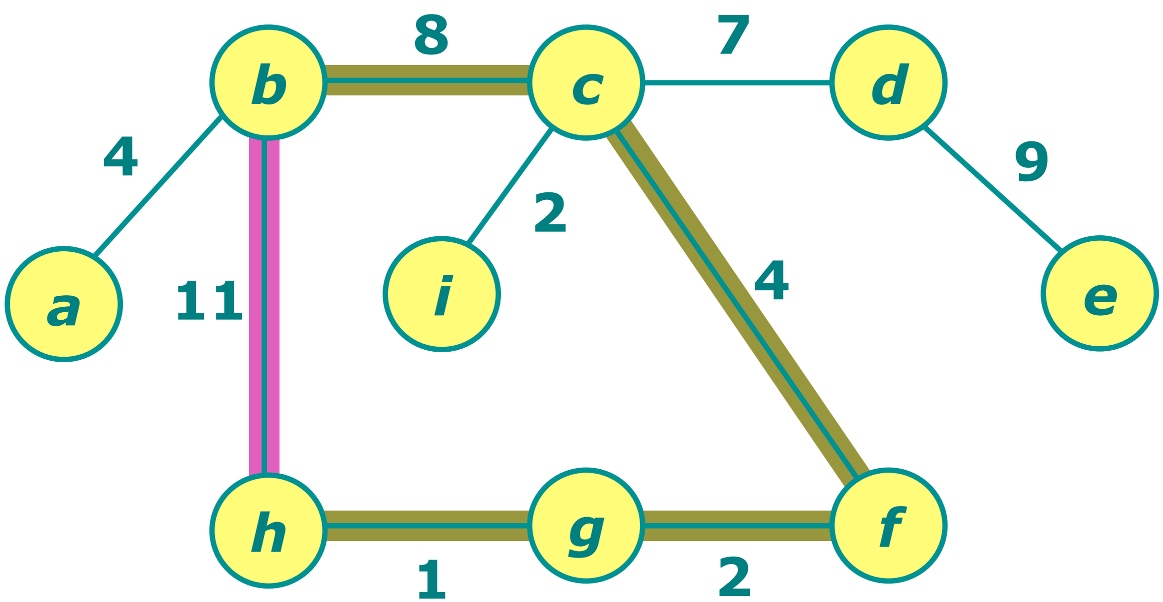
Destroy cycles



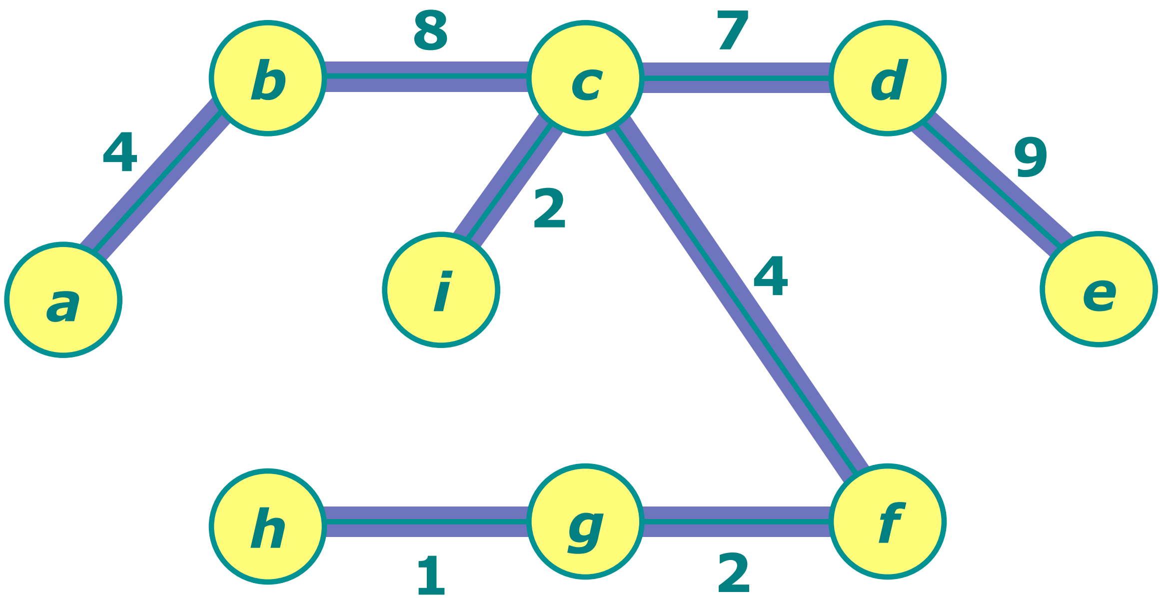
Destroy cycles



Destroy cycles

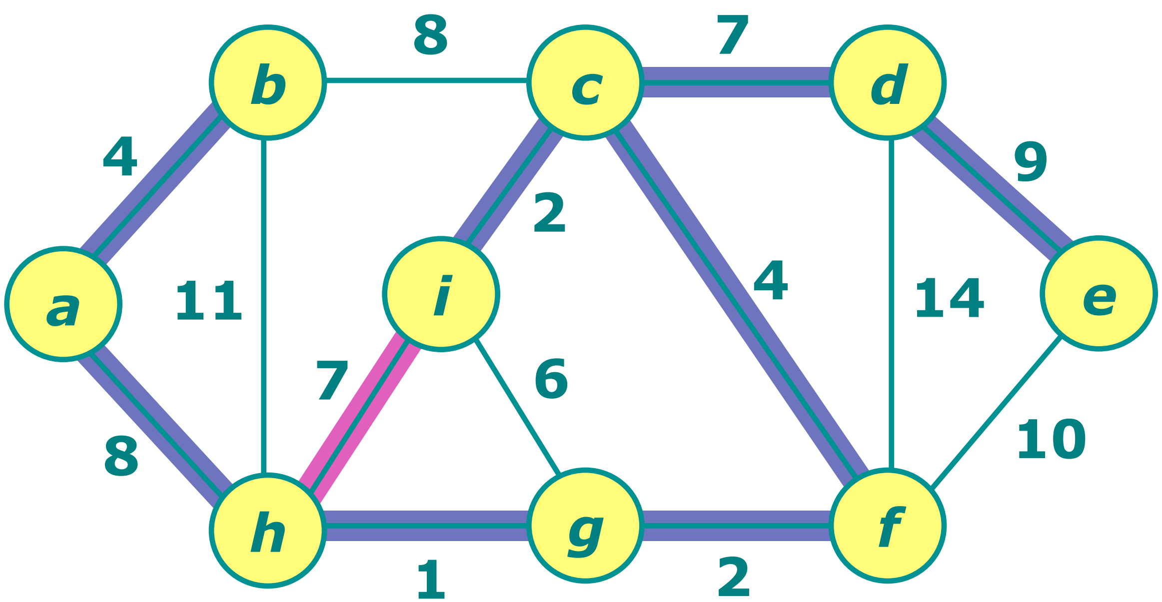


Destroy cycles



Total weight = 1 + 2 + 2 + 4 + 4 + 7 + 8 + 9 = 37

Avoid cycles



Total weight = 1 + 2 + 2 + 4 + 4 + 7 + 8 + 9 = 37

Kruskal's algorithm

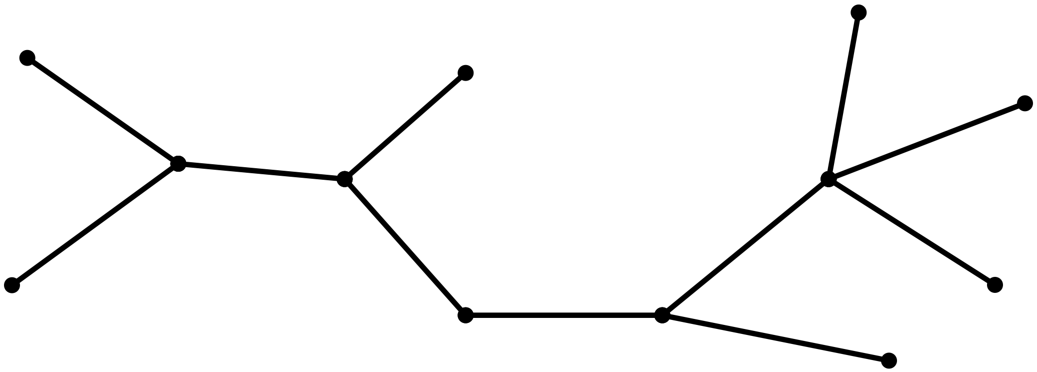
MST-KRUSKAL(G, w)

1. $A \leftarrow \varnothing$
2. **for** each vertex $v \in V[G]$
3.   **do** MAKE-SET(v)
4. sort the edges of E into nondecreasing order by weight w.
5. **for** each edge $(u, v) \in E$, taken in nondecreasing order by weight.
6.   **do if** FIND-SET$(u) \neq$ FIND-SET(v)
7.   **then** $A \leftarrow A \cup \{(u, v)\}$
8.   UNION(u, v)
9. **return** A

Running time is $O(E \lg E)$

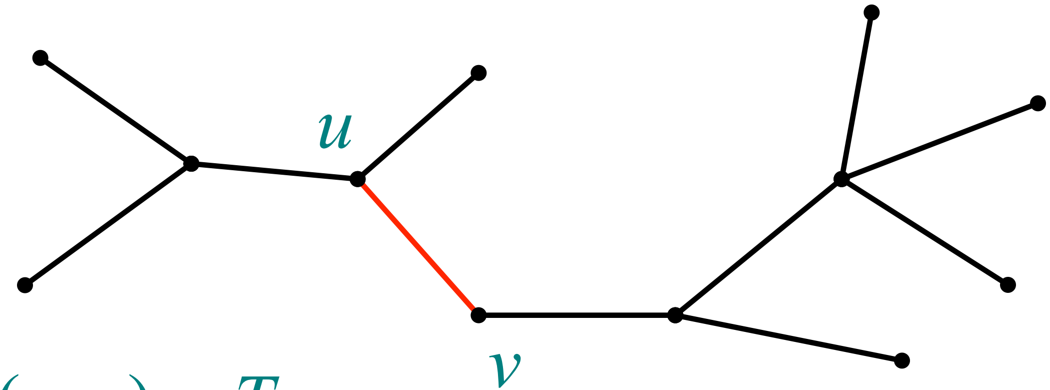
Optimal substructure

Minimum spanning tree T of $G = (V, E)$.



Optimal substructure

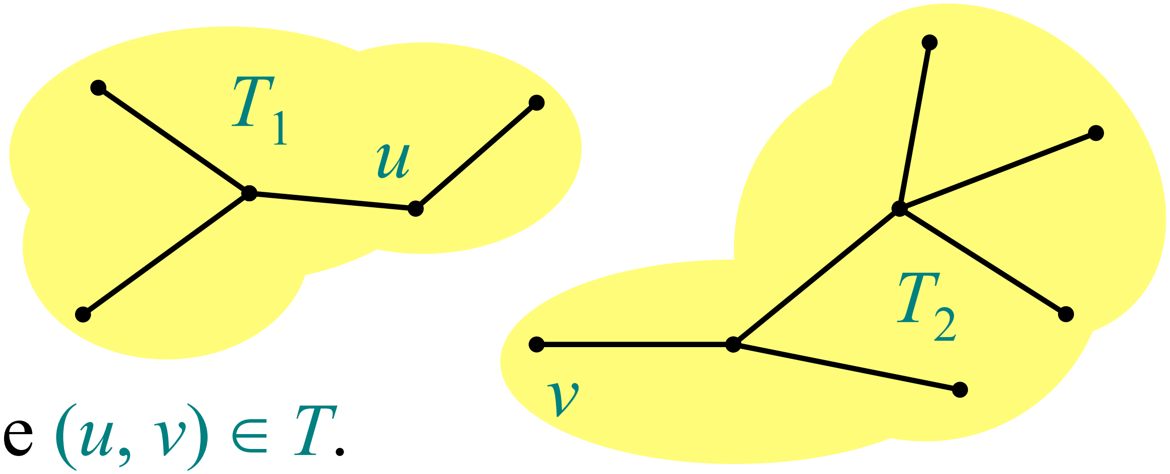
Minimum spanning tree T of $G = (V, E)$.



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

Optimal substructure

Minimum spanning tree T of $G = (V, E)$.



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

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

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

Similarly for T_2.

Proof of optimal substructure

Proof. Cut and paste:

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

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

- Do we also have overlapping subproblems? **Yes!**
- Great, then dynamic programming may work! **Yes!**

But minimum spanning tree exhibits another powerful property which leads to an even more efficient algorithm.

Hallmark for "greedy" algorithms

Greedy-choice property

A locally optimal choice is globally optimal.

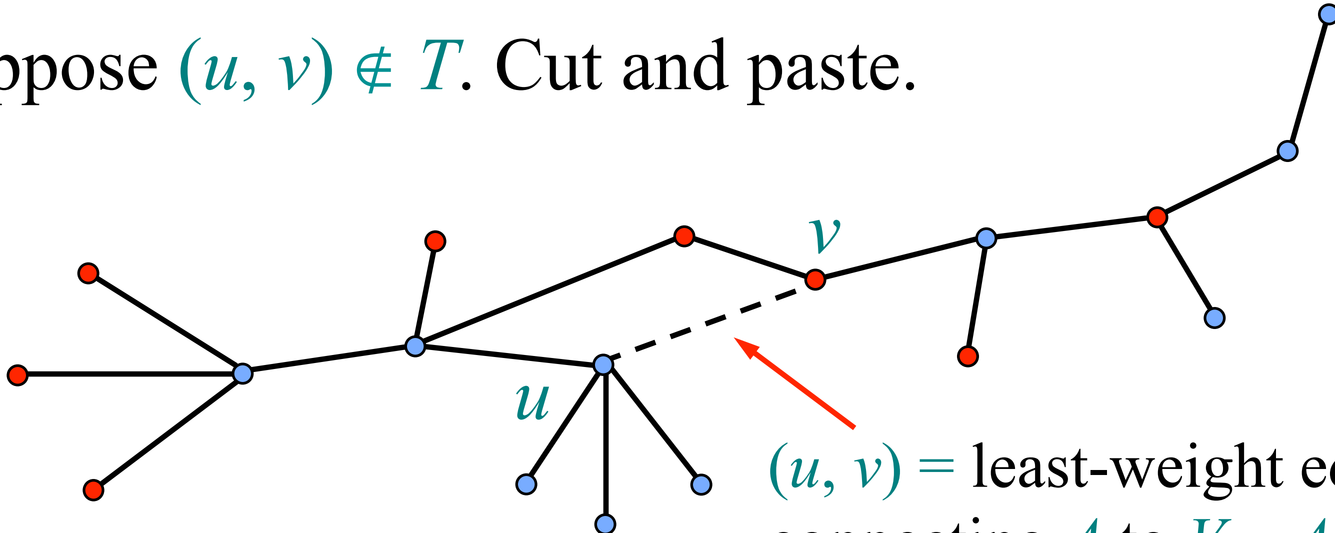
Theorem. Let T be the minimum spanning tree of $G = (V, E)$, and let $A \subseteq V$. Suppose that $(u, v) \in E$ is the least-weight edge connecting A to $V - A$. Then, $(u, v) \in T$.

Proof of theorem

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

T:

- $\in A$
- $\in T - A$



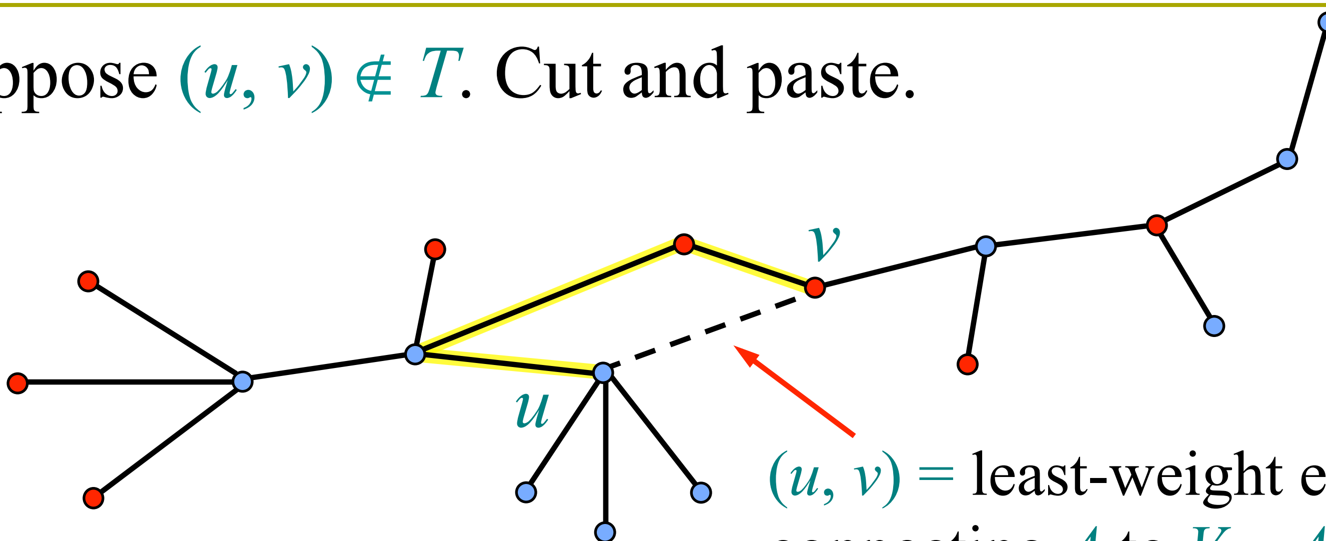
$(u, v) =$ least-weight edge connecting A to $V - A$.

Proof of theorem

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

T:

- $\in A$
- $\in T - A$



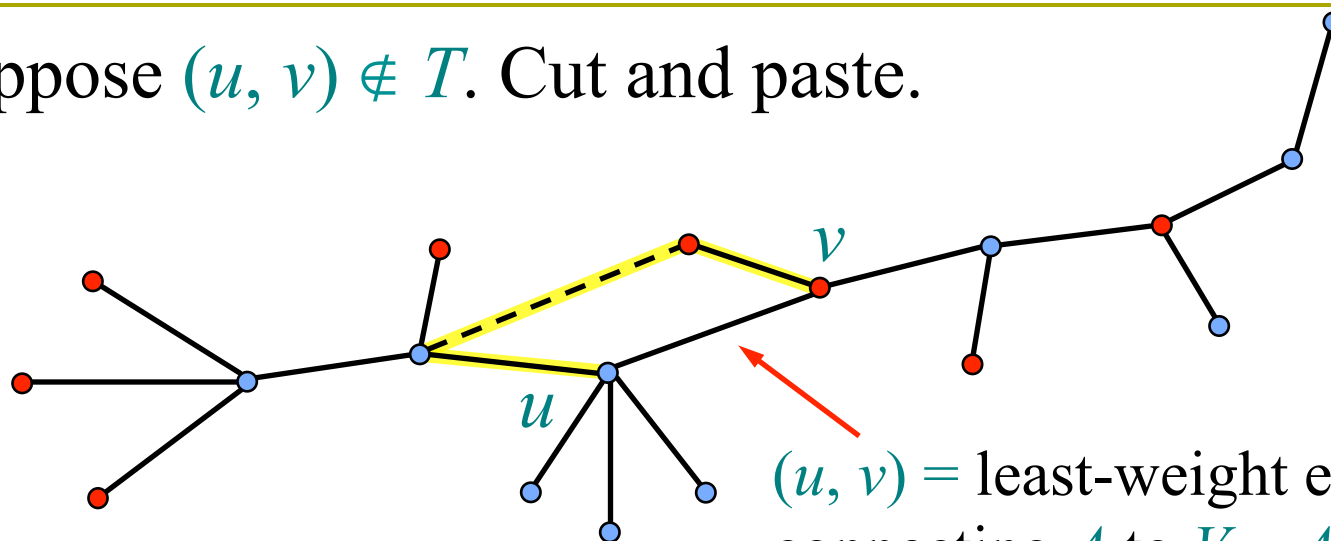
(u, v) = least-weight edge connecting A to $V - A$.

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

Proof of theorem

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

T:
- $\bullet \in A$
- $\bullet \in T - A$



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

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

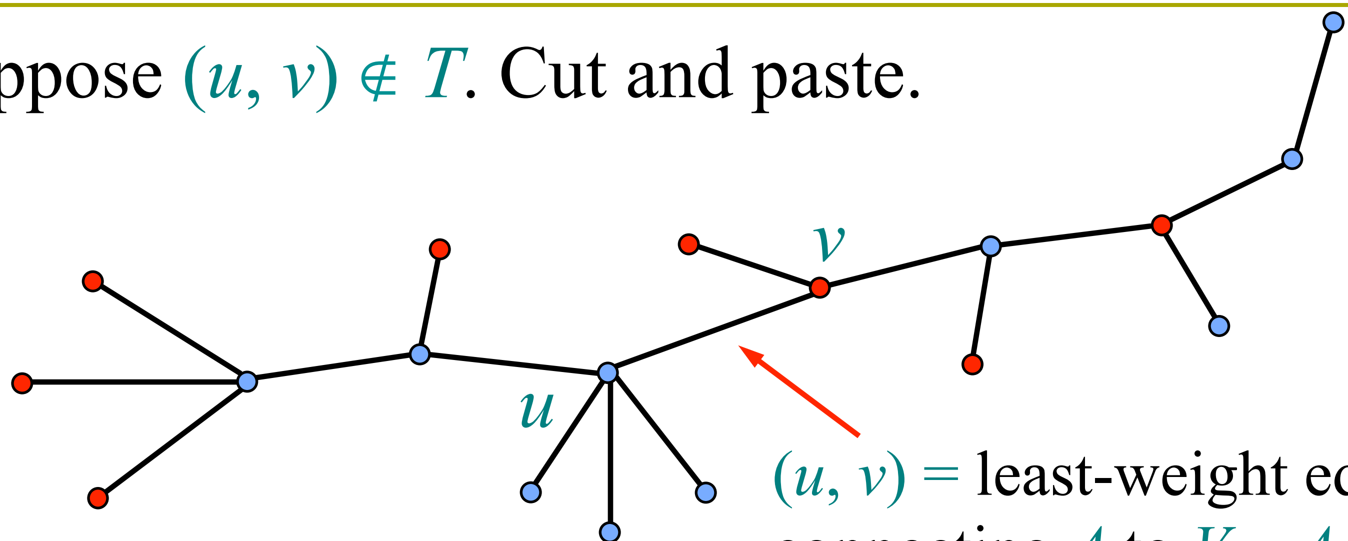
Swap (u, v) with the first edge on this path that connects a vertex in A to a vertex in $V - A$.

Proof of theorem

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

T:

- • $\in A$
- • $\in T - A$



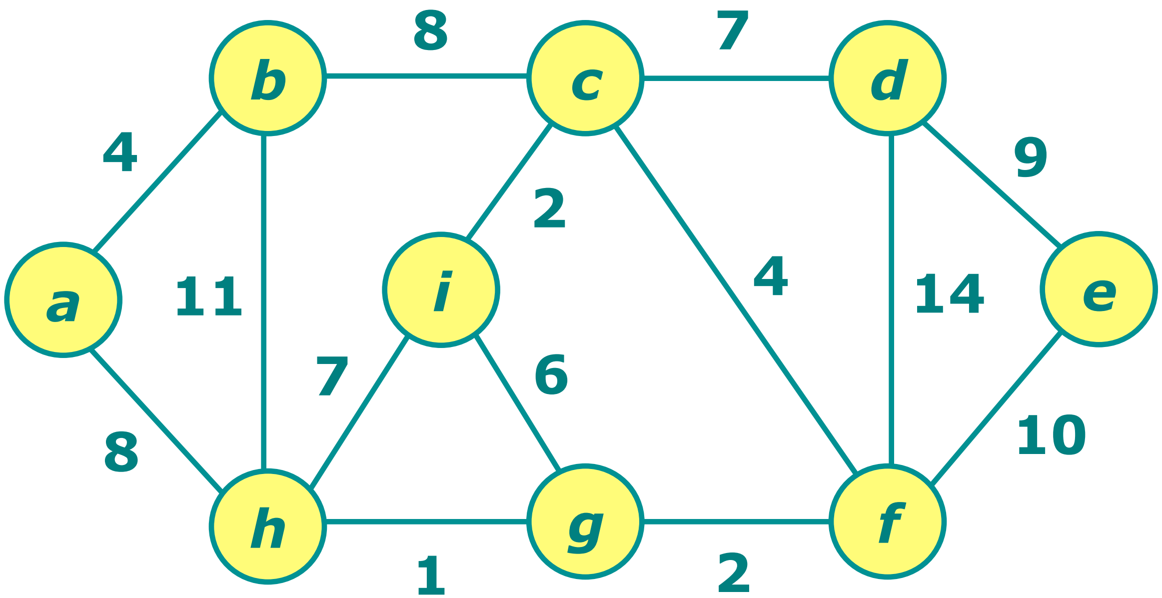
(u, v) = least-weight edge connecting A to $V - A$.

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

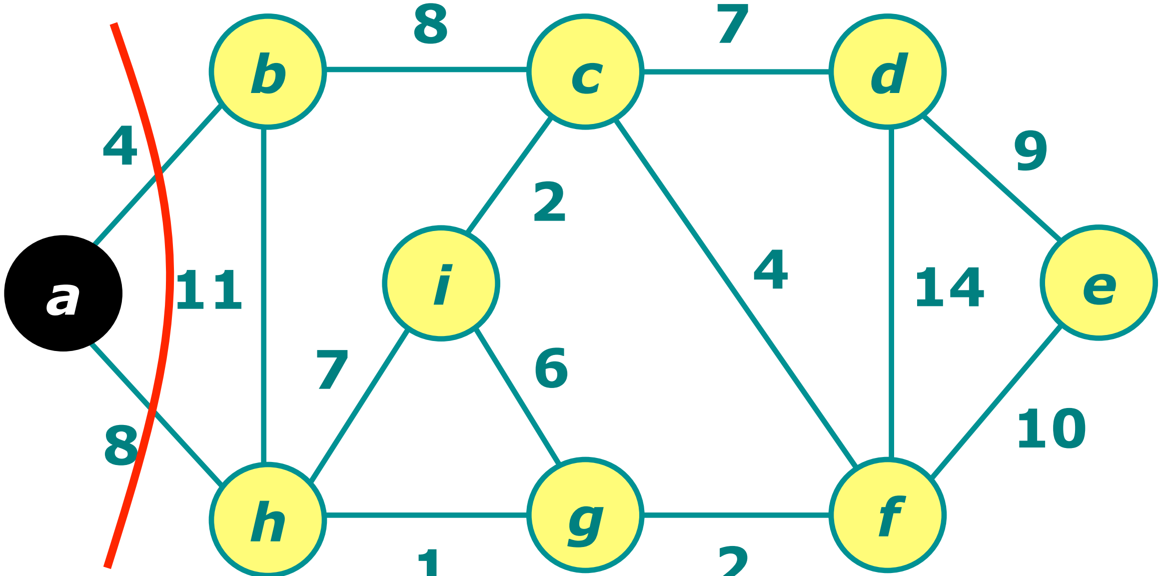
Swap (u, v) with the first edge on this path that connects a vertex in A to a vertex in $V - A$.

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

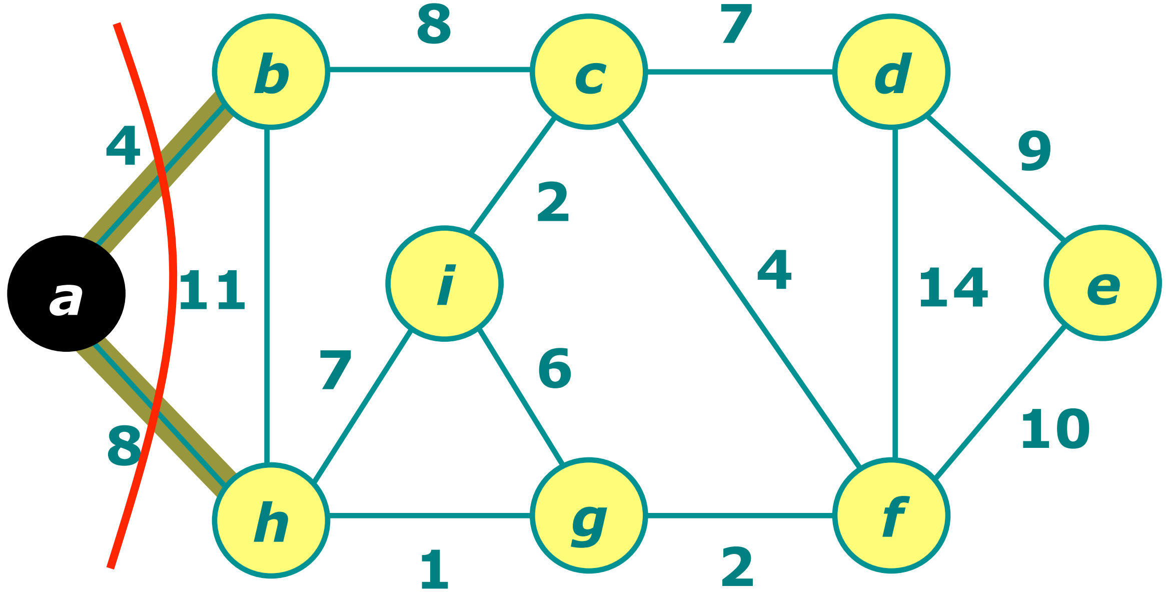
Example of Prim's algorithm



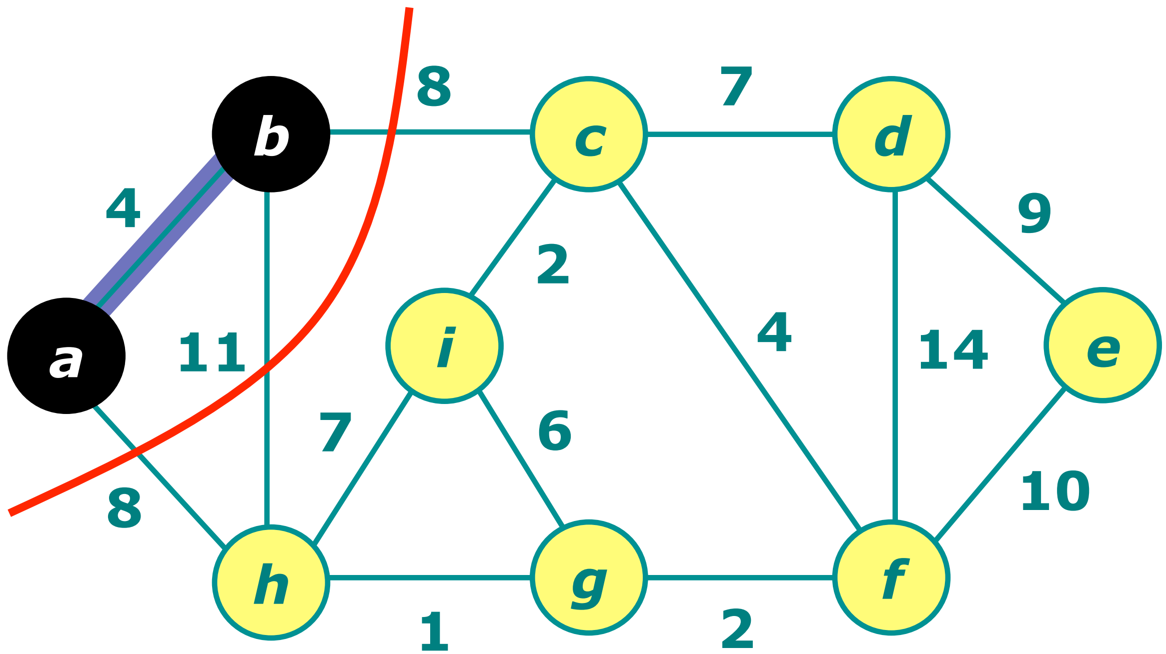
Example of Prim's algorithm



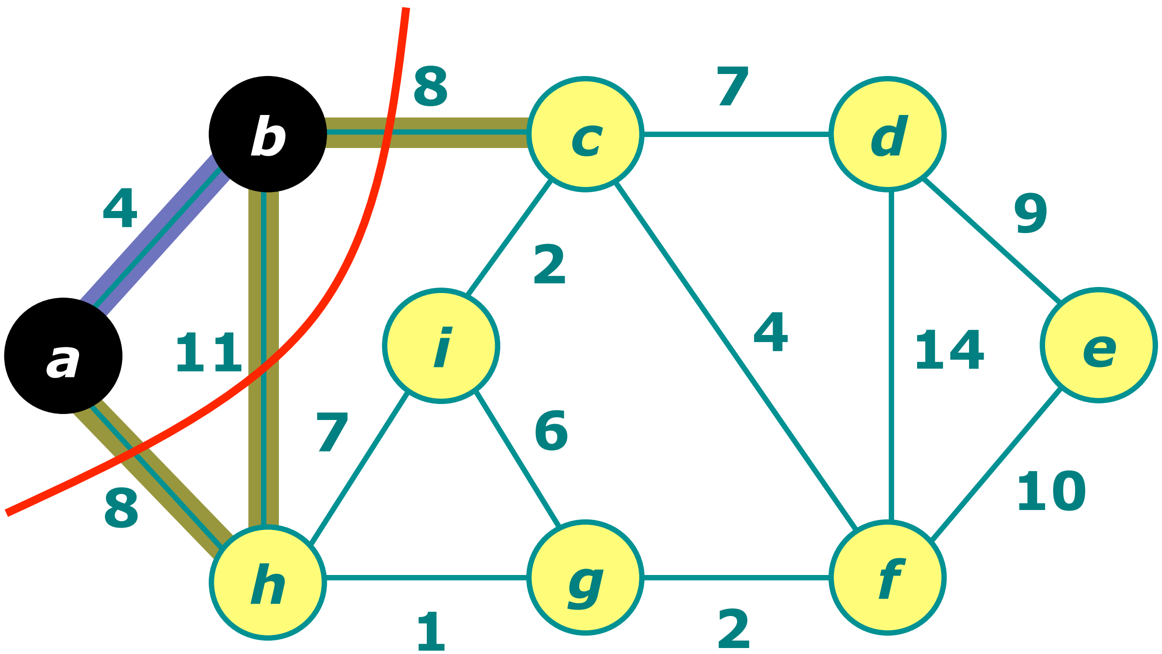
Example of Prim's algorithm



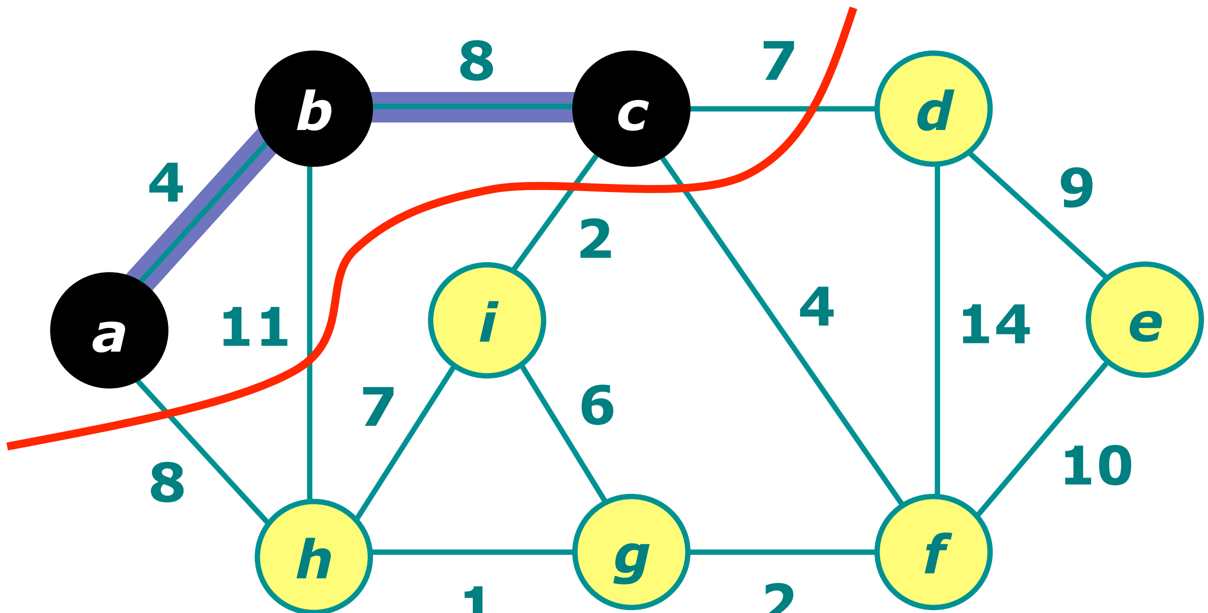
Example of Prim's algorithm



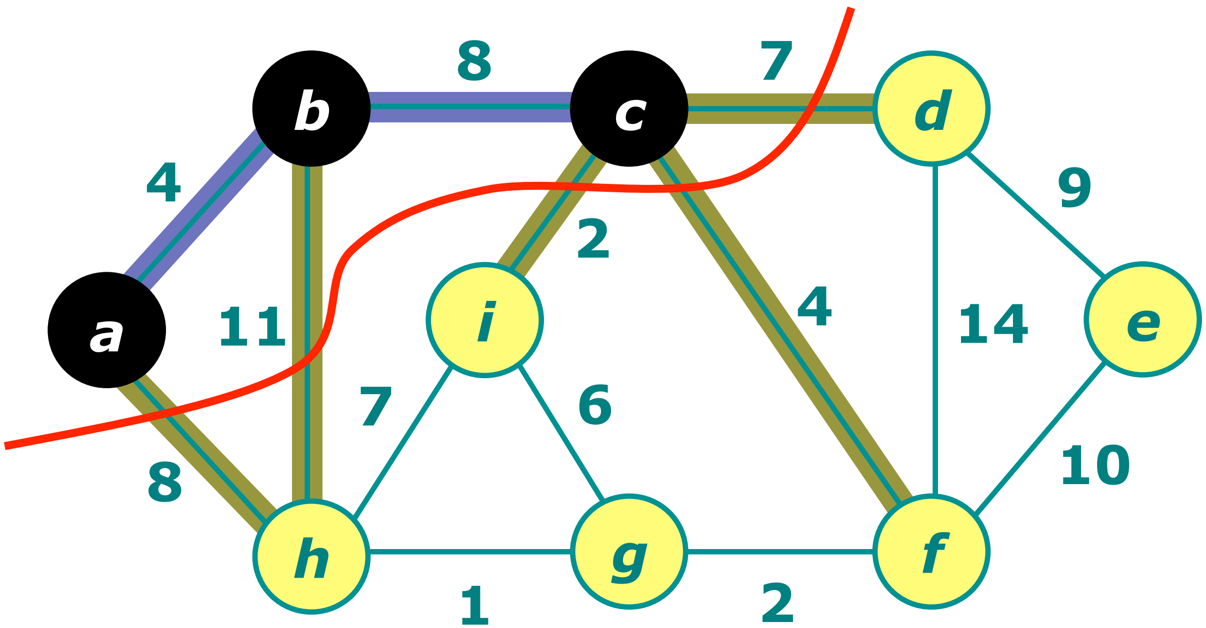
Example of Prim's algorithm



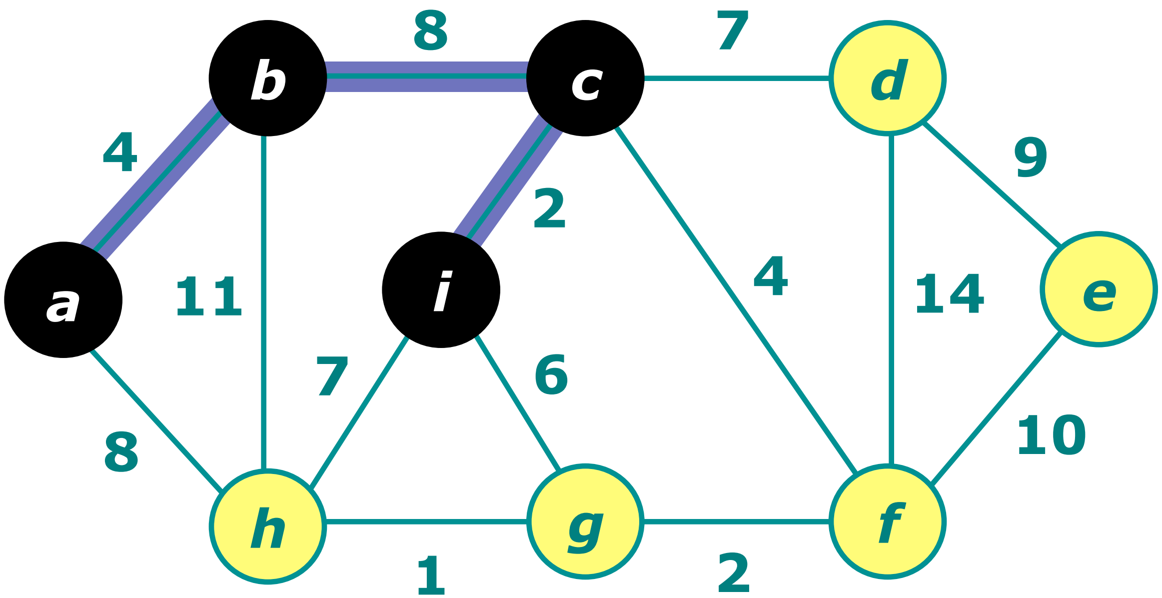
Example of Prim's algorithm



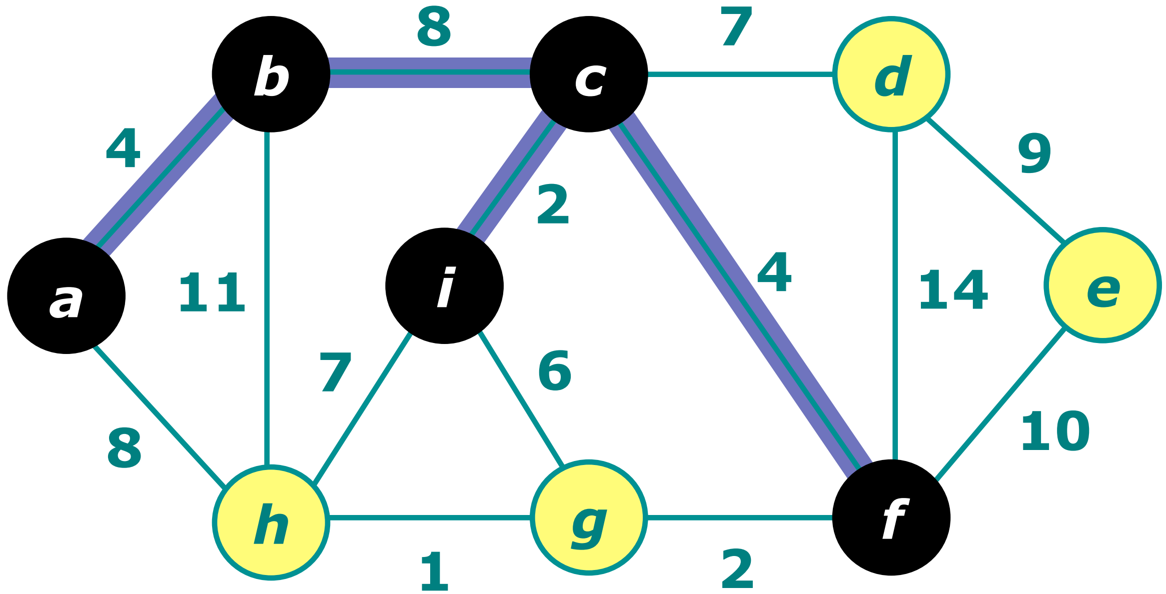
Example of Prim's algorithm



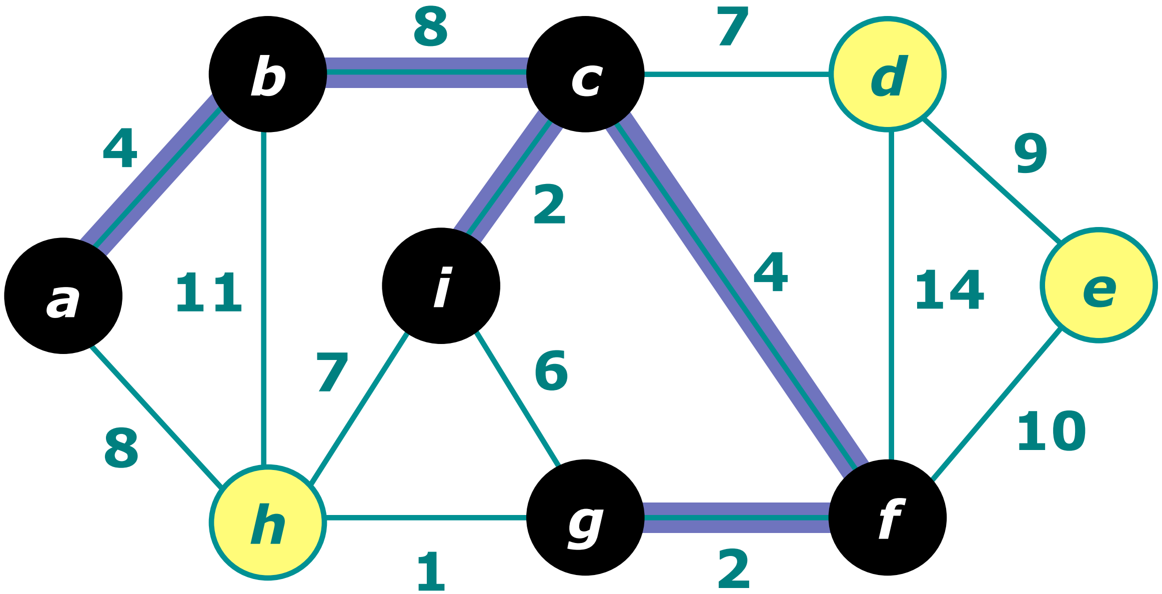
Example of Prim's algorithm



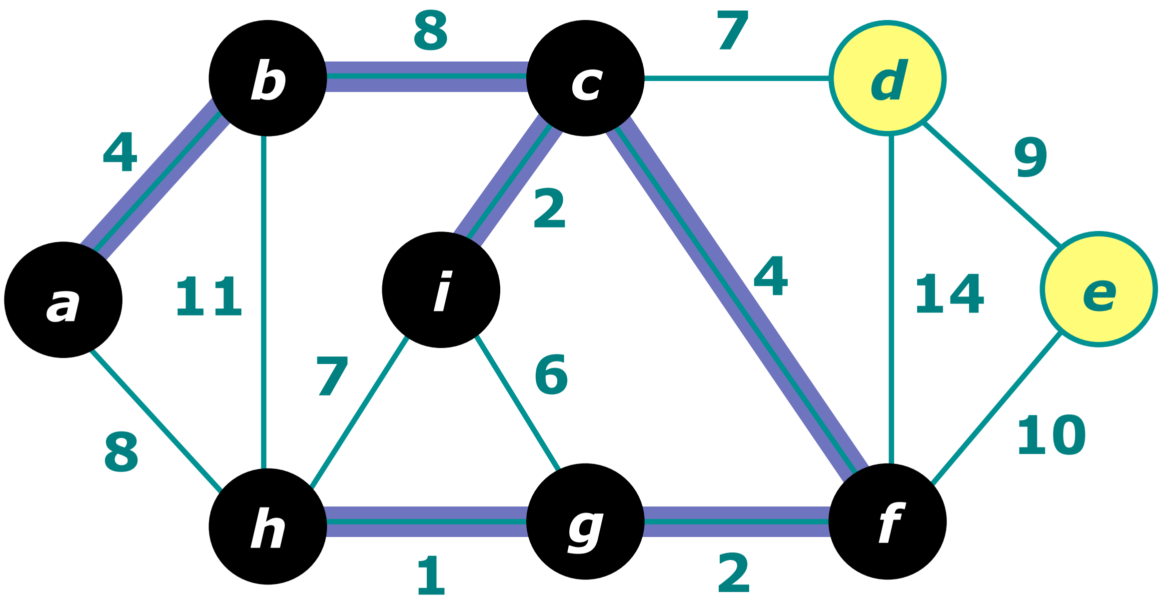
Example of Prim's algorithm



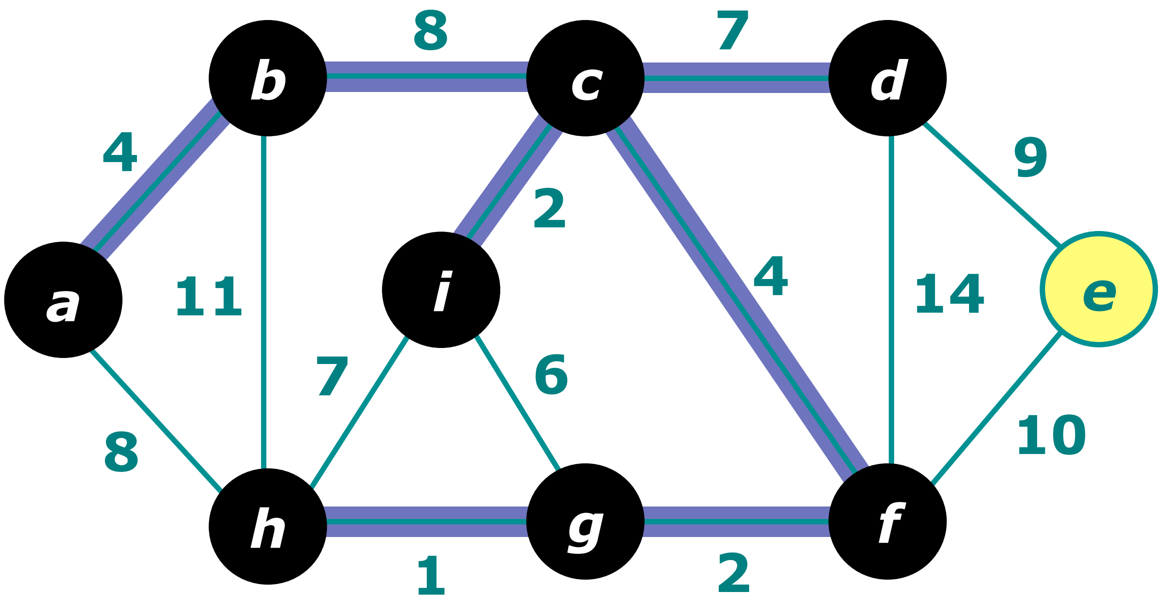
Example of Prim's algorithm



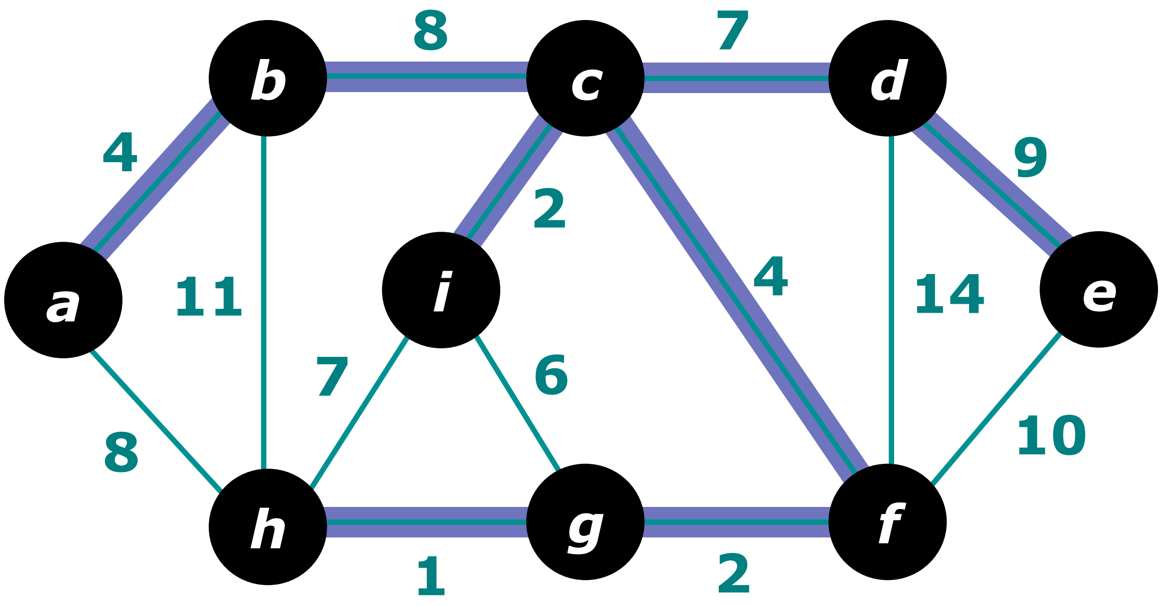
Example of Prim's algorithm



Example of Prim's algorithm



Example of Prim's algorithm



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.

MST-PRIM(G, w, r)

for each $u \in V[G]$
    do $key[u] \leftarrow \infty$
        $\pi[u] \leftarrow \mathrm{NIL}$
$key[r] \leftarrow \varnothing$
$Q \leftarrow V[G]$
while $Q \neq 0$
    do $u \leftarrow$ EXTRACT-MIN(Q)
        for each $v \in Adj[u]$
            do if $v \in Q$ and $w(u, v) < key(v)$
                then $\pi[v] \leftarrow u$
                    $key[v] \leftarrow w(u, v)$

Minimum spanning tree A for G is thus $A = \{ (v, \pi[v]) : v \in V - \{r\} \}$

Analysis of Prim algorithm

MST-PRIM(G, w, r)

for each $u \in V[G]$ $\Phi(V)$ total

  do $key[u] \leftarrow \infty$

   $\pi[u] \leftarrow \text{NIL}$

$key[r] \leftarrow \varnothing$

$Q \leftarrow V[G]$

while $Q \neq 0$

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

   for each $v \in Adj[u]$

$degree(u)$ times   **do if** $v \in Q$ and $w(u, v) < key(v)$ $|V|$ times

     then $\pi[v] \leftarrow u$

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

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

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

Analysis of Prim algorithm

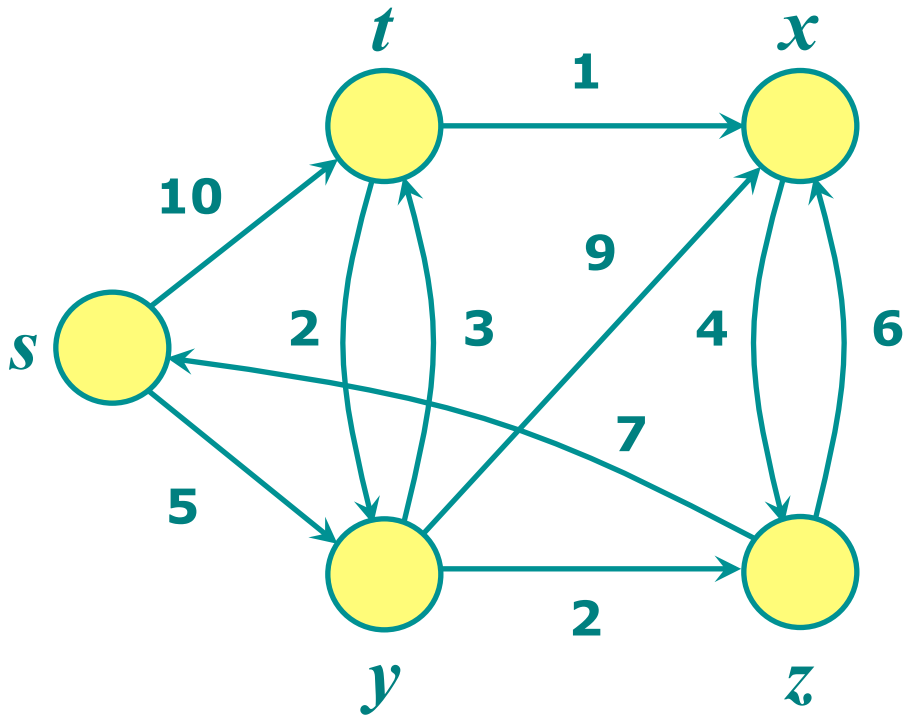
Time $= \Theta(V) \cdot Time_{\text{EXTRACT-MIN}} + \Theta(E) \cdot Time_{\text{DECREASE-KEY}}$

Q	$Time_{\text{EXTRACT-MIN}}$	$Time_{\text{DECREASE-KEY}}$	Total
Array	$O(V)$	$O(1)$	$O(V^2)$
Binary heap	$O(lgV)$	$O(lgV)$	$O(E\lg V)$

Running time of Prim's algorithm is $O(E\lg V)$

Pudong Shanghai

Shortest paths



Shortest paths problem

Consider a directed graph $G = (V, E)$, with weight function $w: E \rightarrow \mathbb{R}$ mapping edges to real-valued weights. The ***weight*** of path $p = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is defined to be

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

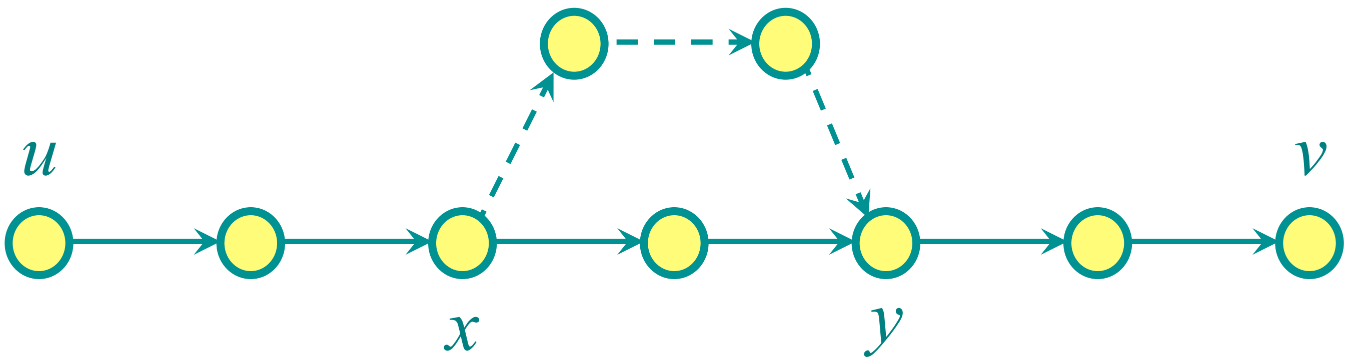
We define the shortest path weight from u to v by

$$\delta(u,v) = \begin{cases} \min\{w(p): u \xrightarrow{p} v\} & \text{If there is a path from } u \text{ to } v, \\ \infty & \text{Otherwise.} \end{cases}$$

Optimal substructure

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

Proof. Cut and paste:



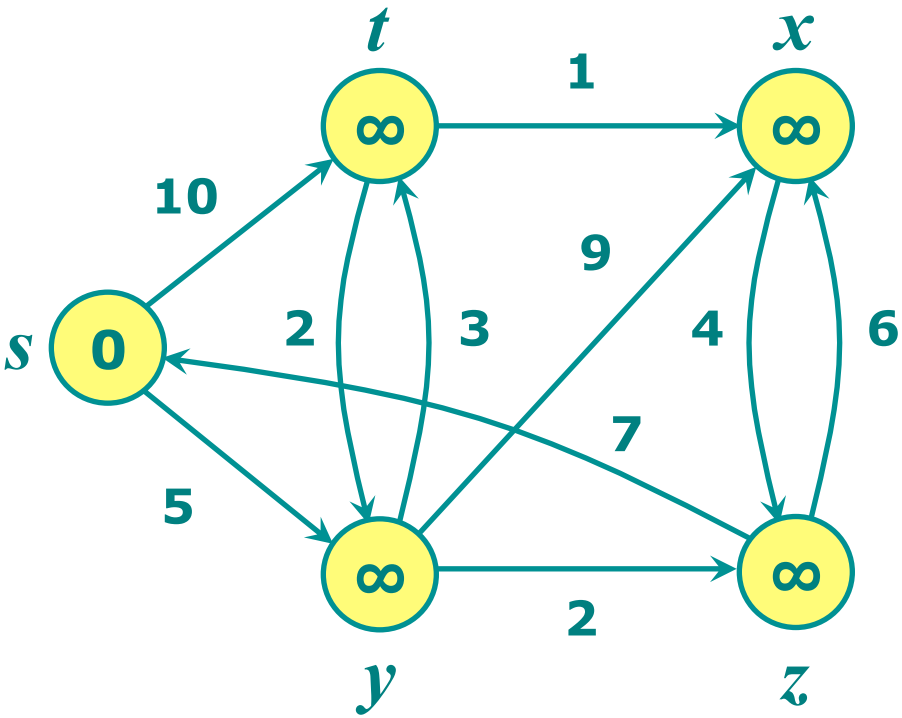
Single-source shortest paths

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.

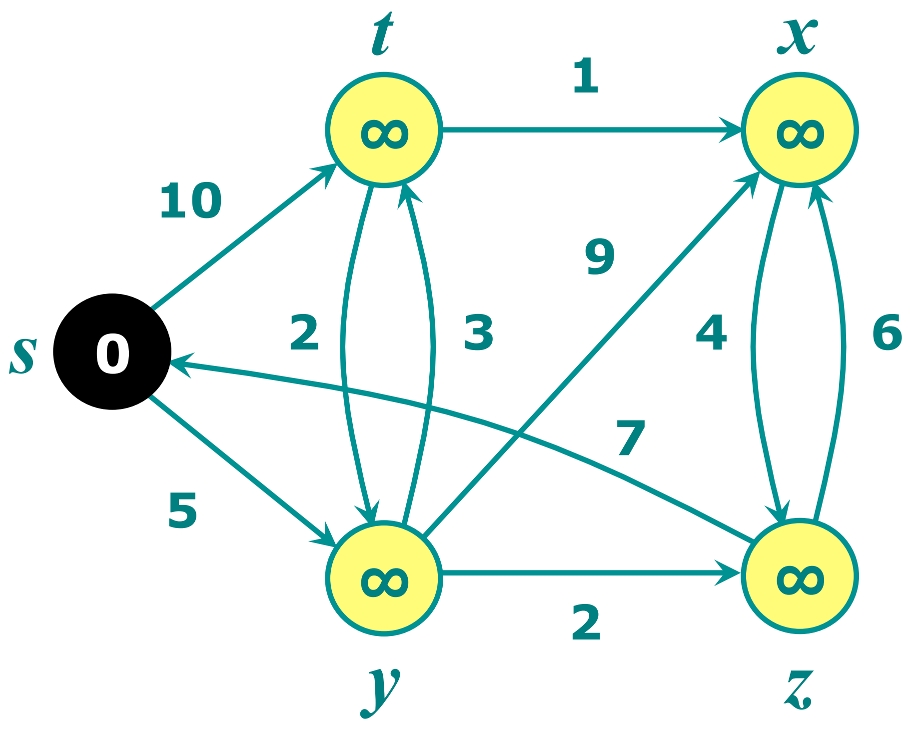
IDEA: Greedy.

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.

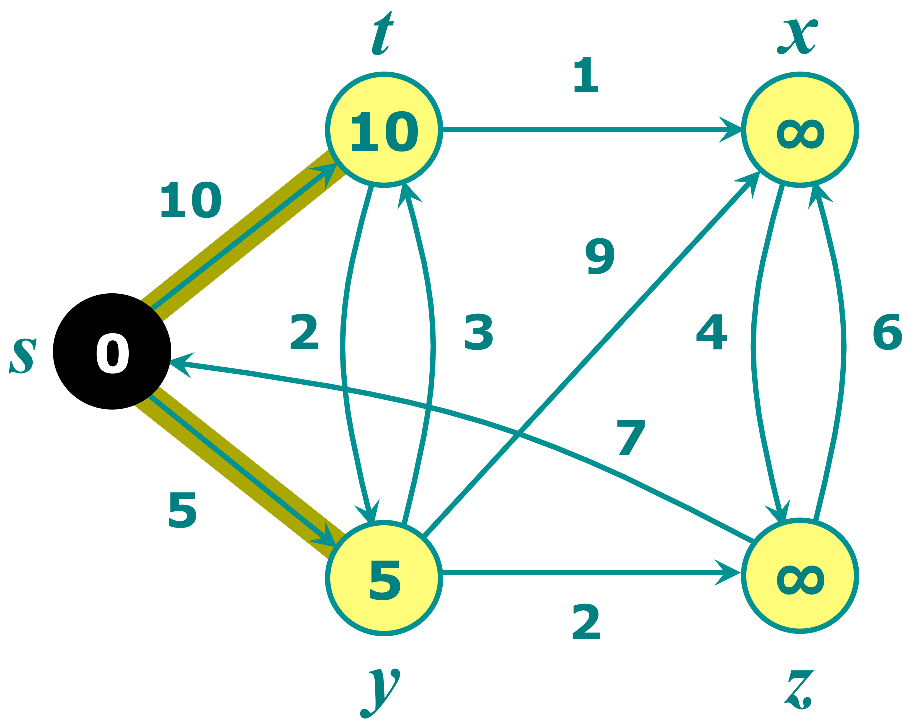
Example of Dijkstra's algorithm



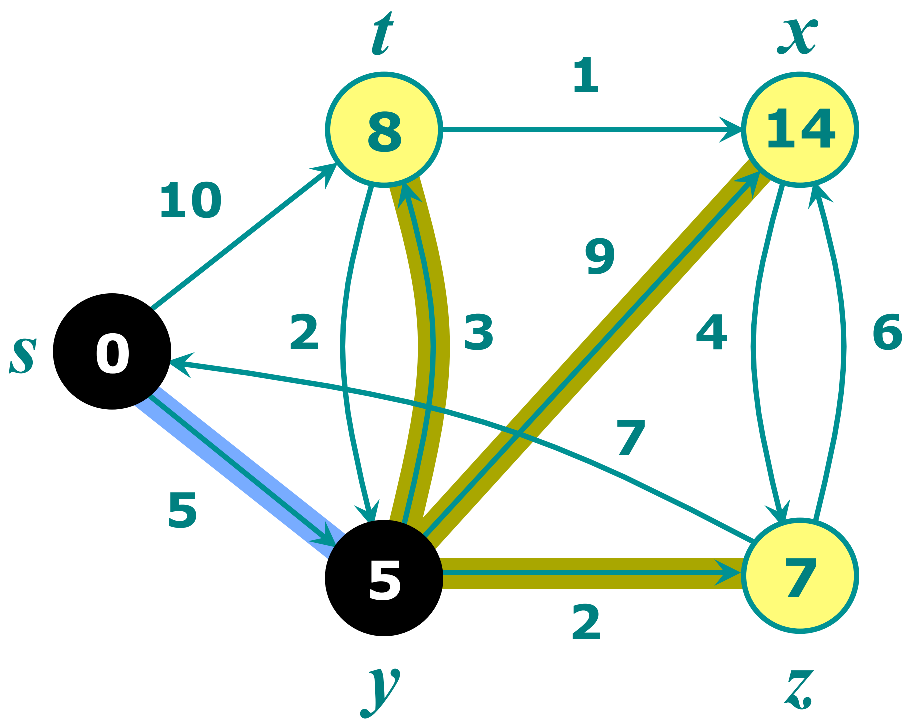
Example of Dijkstra's algorithm



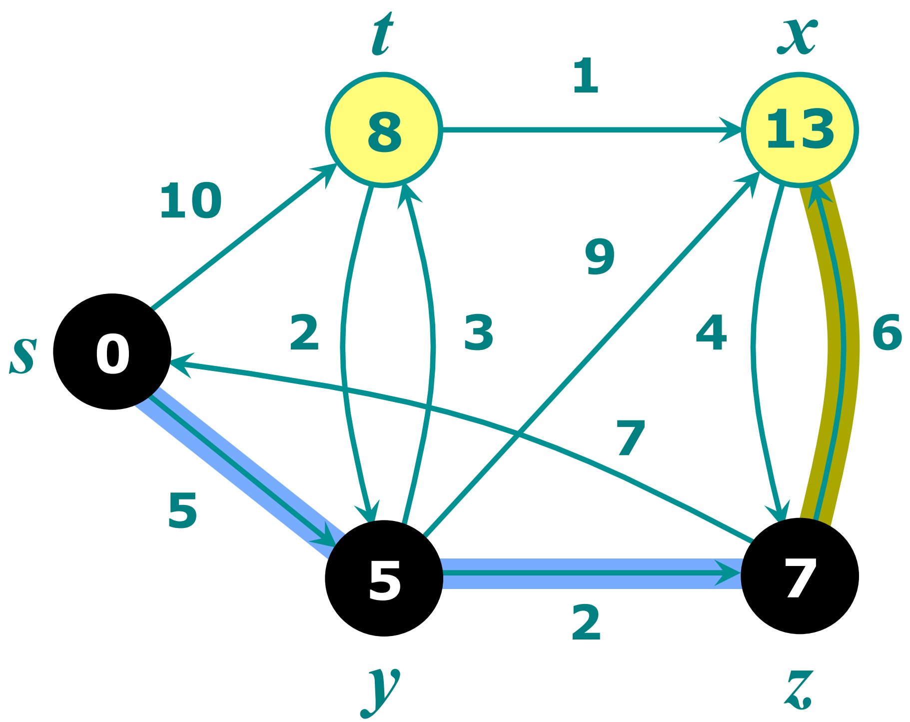
Example of Dijkstra's algorithm



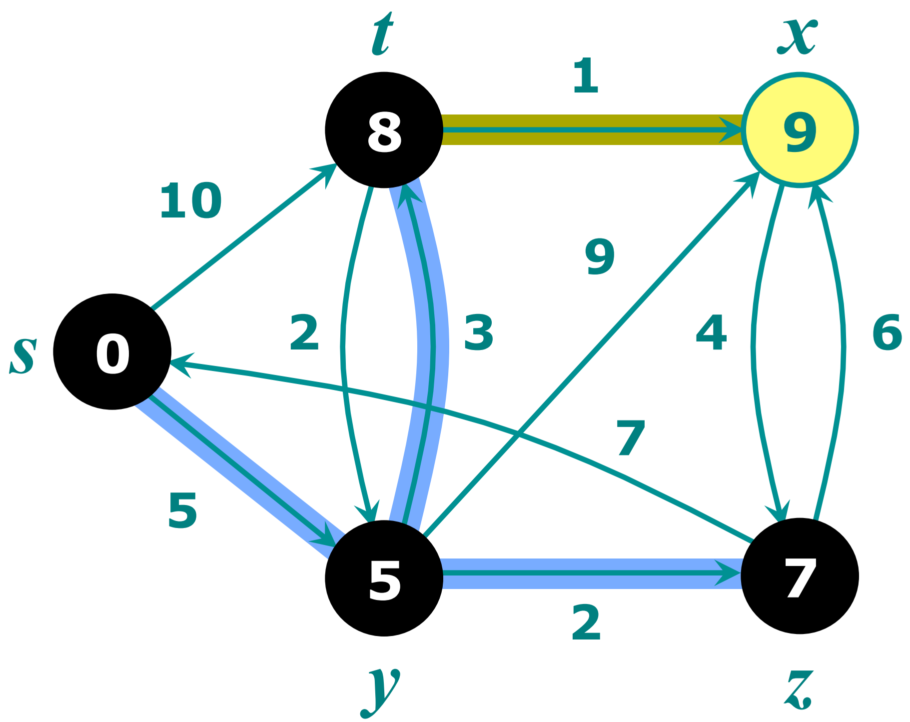
Example of Dijkstra's algorithm



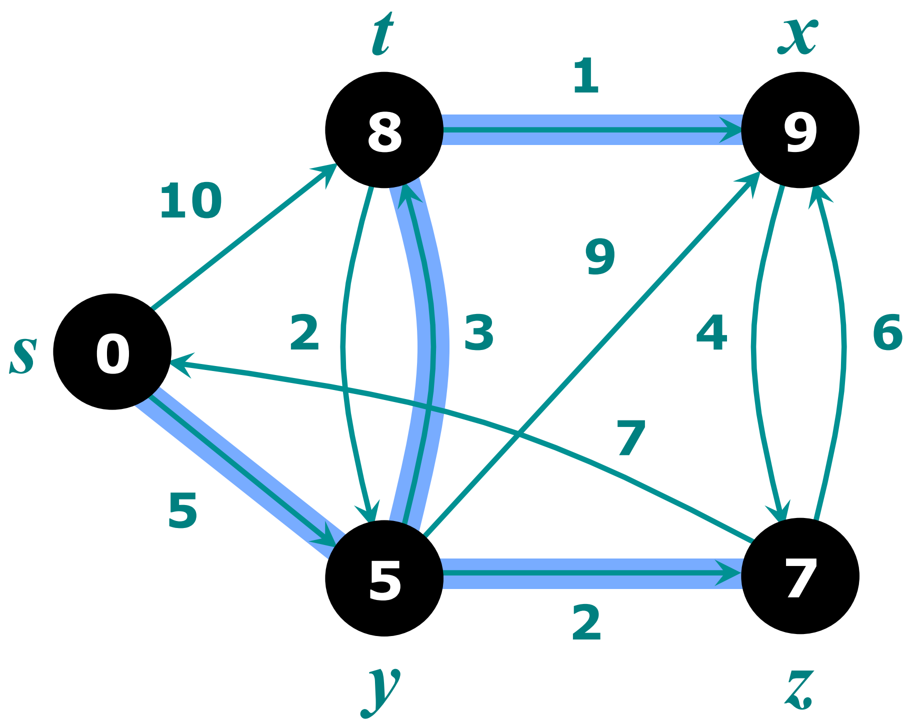
Example of Dijkstra's algorithm



Example of Dijkstra's algorithm



Example of Dijkstra's algorithm



Dijkstra's algorithm

```
DIJKSTRA(G, w, s)
for each vertex v ∈ V[G]
        do d[v] ← ∞
           π(v) ← NIL
d[s] ← 0
S ← ∅
Q ← V[G]
while Q ≠ ∅
        do u ← EXTRACT-MIN(Q)
           S ← S ∪ {u}
           for each vertex v ∈ Adj[u]
               do if d[v] > d[u] + w(u, v)
                     then d[v] ← d[u] + w(u, v)
                          π(v) ← u
```

Relaxation step.
Implicit DECREASE-KEY.

Correctness of Dijkstra's algorithm

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,

$d[v] < \delta(s, v)$	supposition
$\leq \delta(s, u) + \delta(u, v)$	triangle inequality
$\leq \delta(s, u) + w(u, v)$	sh. path $\leq$ specific path
$\leq d[u] + w(u, v)$	v is first violation

Contradiction $\square$

Correctness of Dijkstra's algorithm

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.

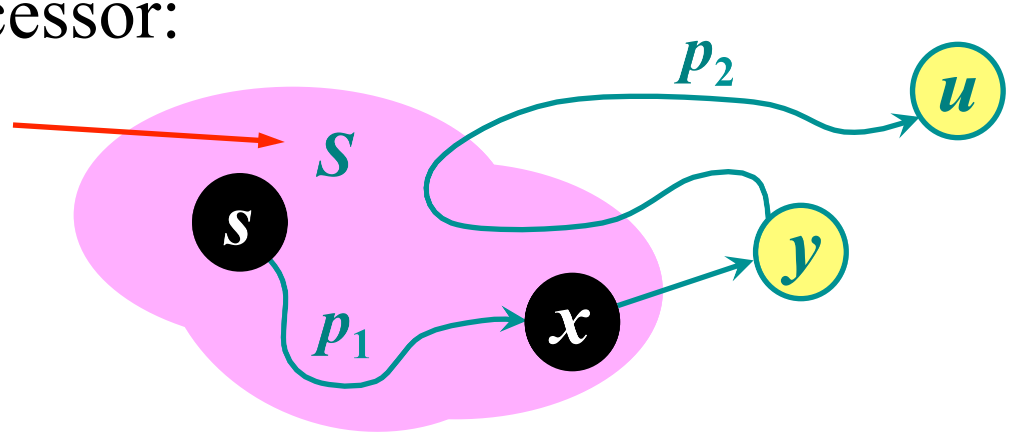
Proof. Observe that $\delta(s, v) = \delta(s, u) + w(u, v)$. Suppose that $d[v] > \delta(s, v)$ before the relaxation. (Otherwise, we're done.) Then, the test $d[v] > d[u] + w(u, v)$ succeeds, because $d[v] > \delta(s, v) = \delta(s, u) + w(u, v) = d[u] + w(u, v)$, and the algorithm sets $d[v] = d[u]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

Correctness of Dijkstra's algorithm

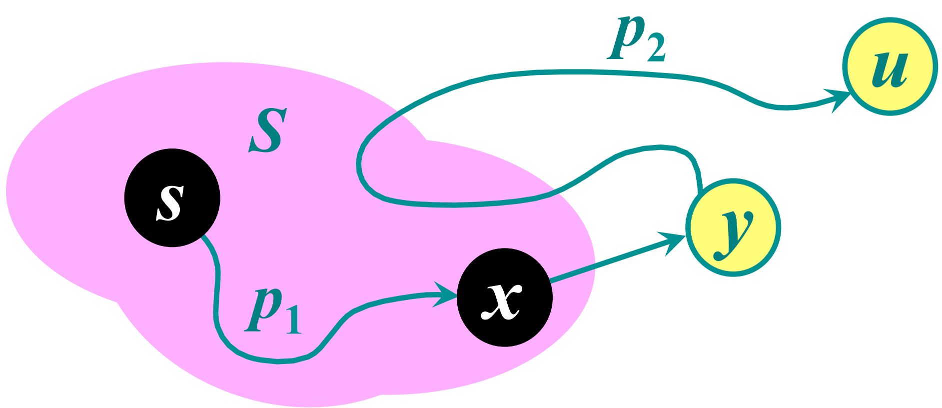
Theorem.
Dijkstra's algorithm terminates with $d[v] = \delta(s, v)$ for all $v \in V$.

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:

Just before adding u.



Correctness of Dijkstra's algorithm



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's algorithm

DIJKSTRA(G, w, s)

for each vertex $v \in V[G]$
  do $d[v] \leftarrow \infty$
    $\pi(v) \leftarrow \text{NIL}$
$d[s] \leftarrow 0$
$S \leftarrow \varnothing$
$Q \leftarrow V[G]$
while $Q \neq \varnothing$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    $S \leftarrow S \cup \{u\}$
    for each vertex $v \in Adj[u]$
      do if $d[v] > d[u] + w(u, v)$
        then $d[v] \leftarrow d[u] + w(u, v)$
          $\pi(v) \leftarrow u$

degree(u) times

$|V|$ times

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

Analysis of Dijkstra's algorithm

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

Q	$Time_{\text{EXTRACT-MIN}}$	$Time_{\text{DECREASE-KEY}}$	Total
Array	$O(V)$	$O(1)$	$O(V^2)$
Binary heap	$O(lgV)$	$O(lgV)$	$O(E\lg V)$

Running time of Dijkstra's algorithm is $O(E\lg V)$

Unweighted graphs

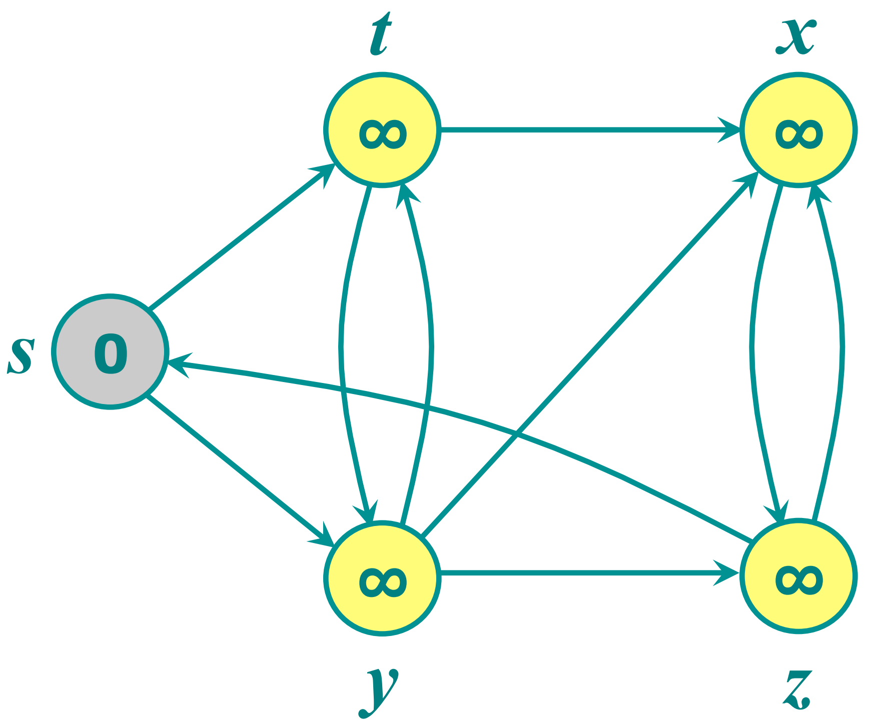
Suppose that $w(u, v) = 1$ for all $(u, v) \in E$.
Can Dijkstra's algorithm be improved?

- ***Breadth-first search***.
- Use a simple **FIFO** queue instead of a **priority** queue.

1. **while** $Q \neq \emptyset$
2. **do** $u \leftarrow$ DEQUEUE(Q)
3. **for** each vertex $v \in Adj[u]$
4. **do if** $d[v] = \infty$
5. **then** $d[v] \leftarrow d[u] + 1$
6. ENQUEUE(Q, v)

Running time is $O(V + E)$.

Example of breadth-first search

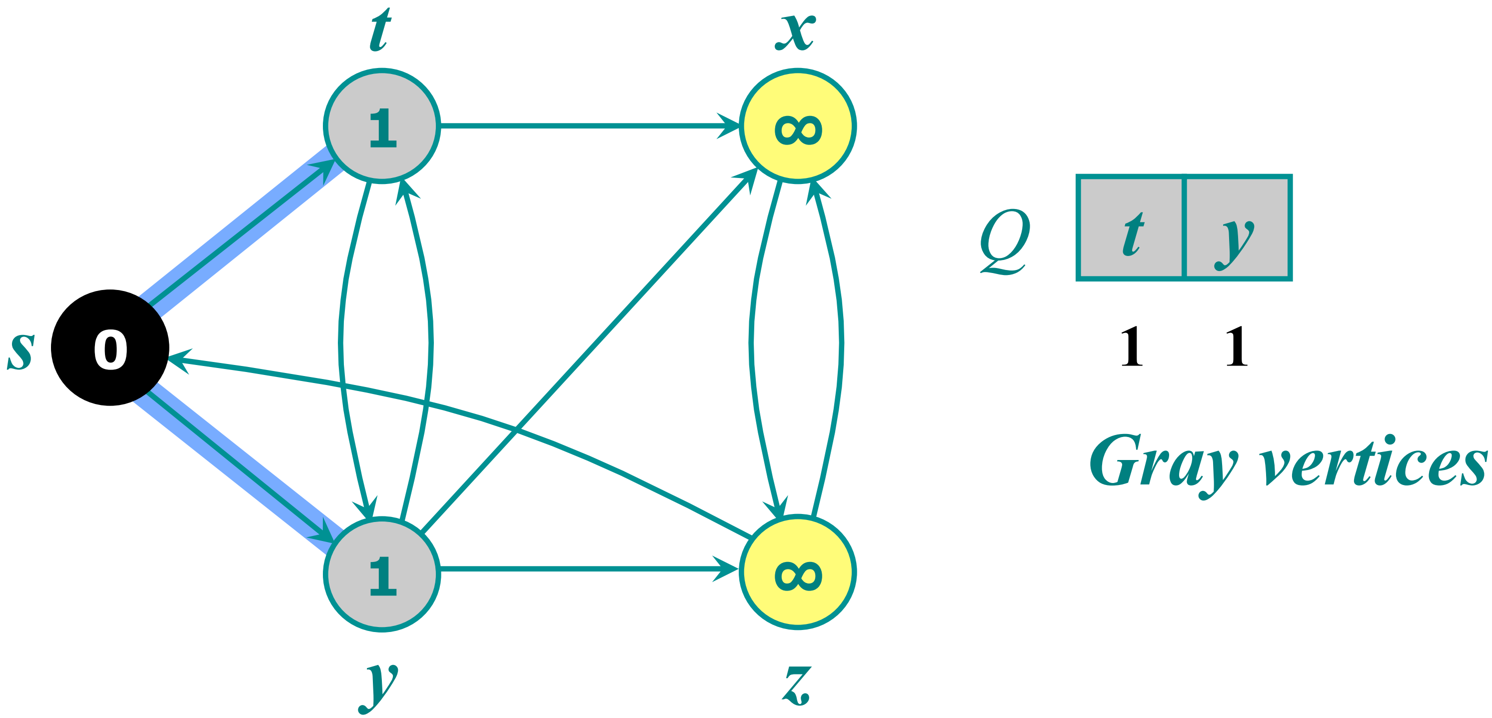


Q s

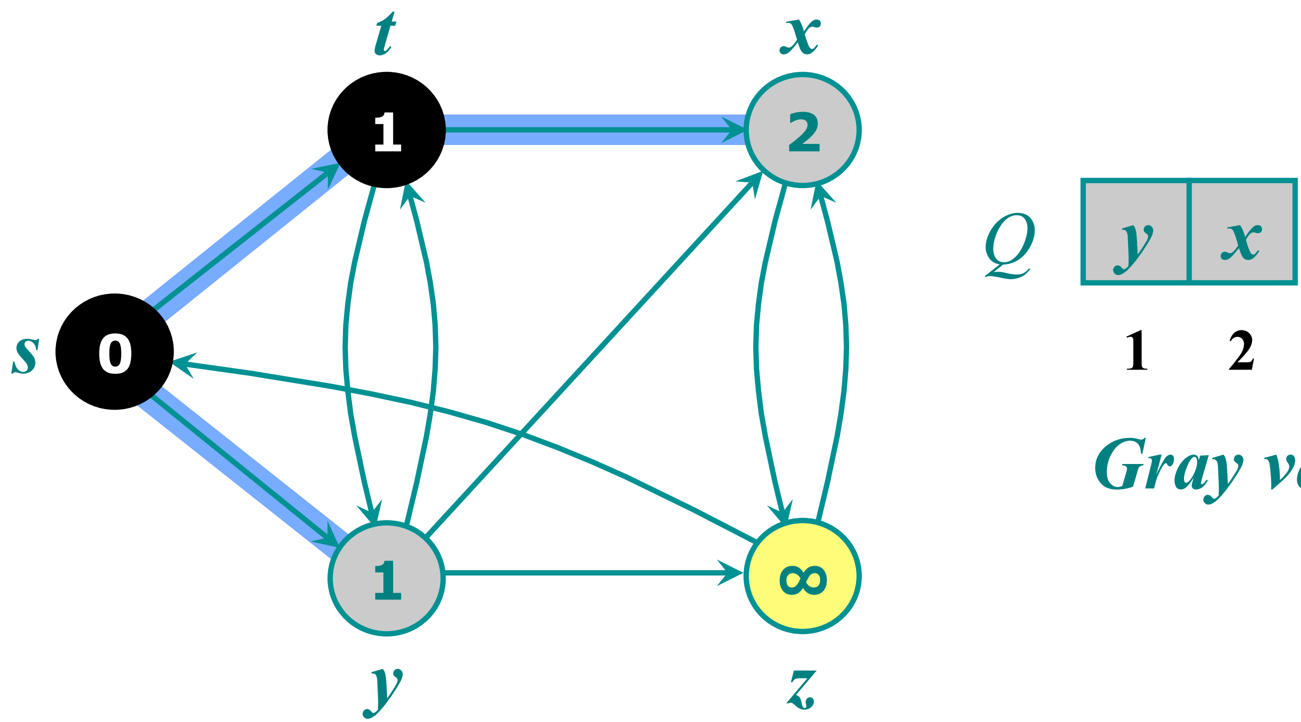
0

Gray vertices

Example of breadth-first search

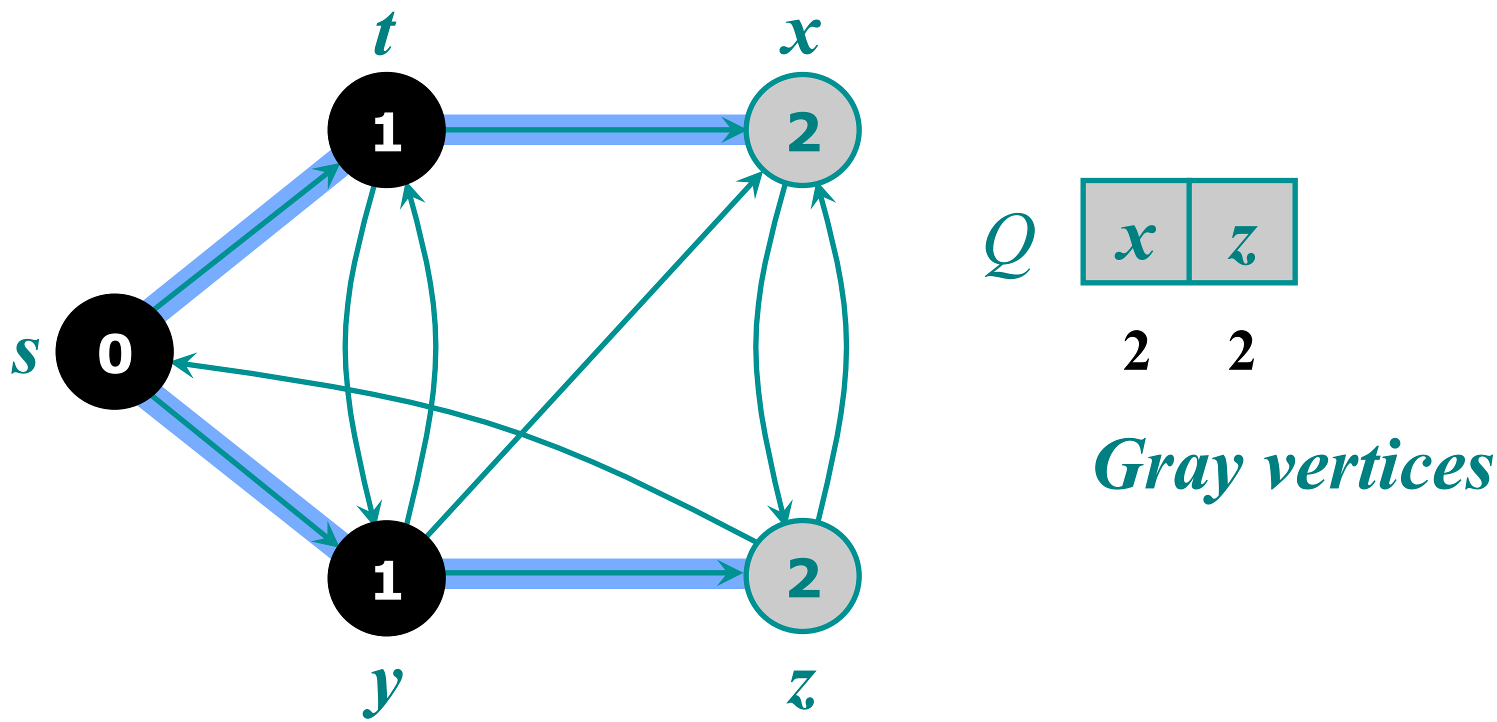


Example of breadth-first search

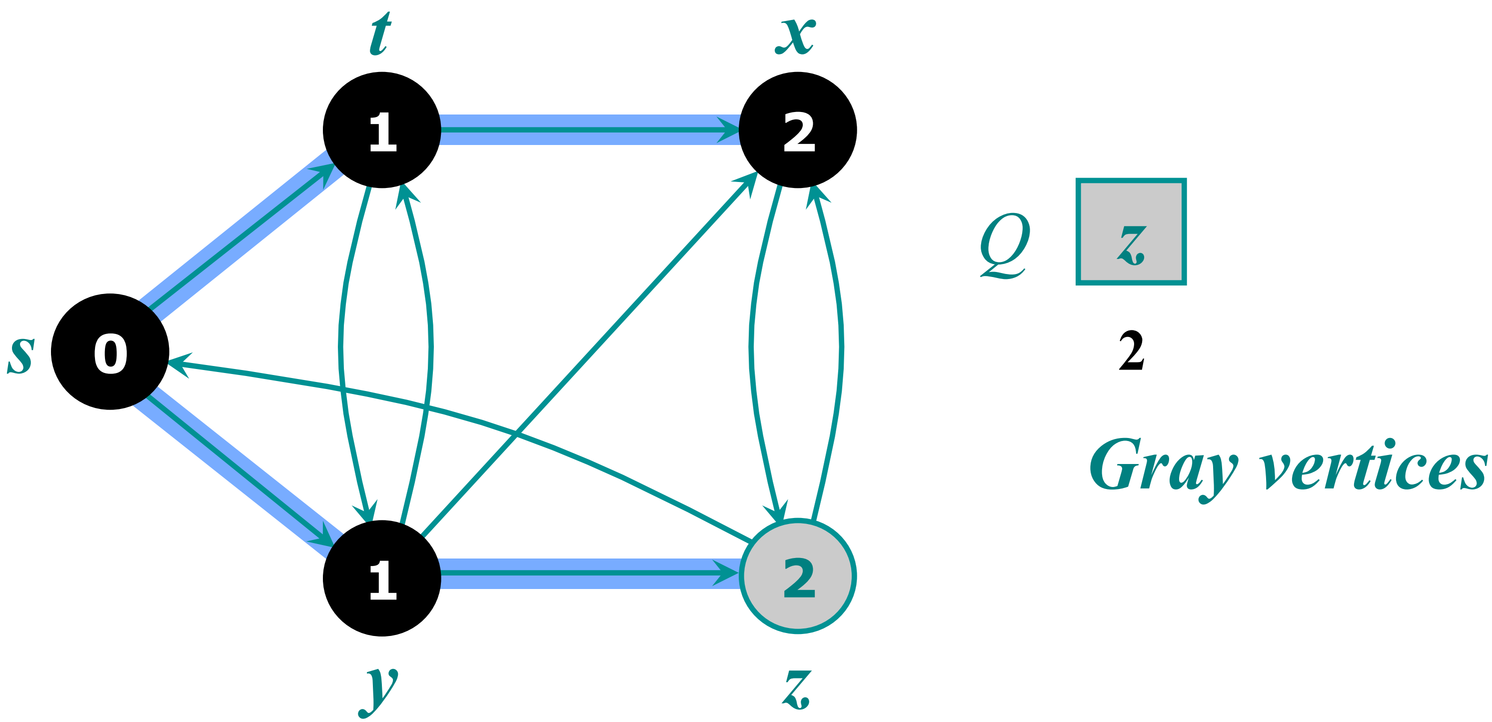


Gray vertices

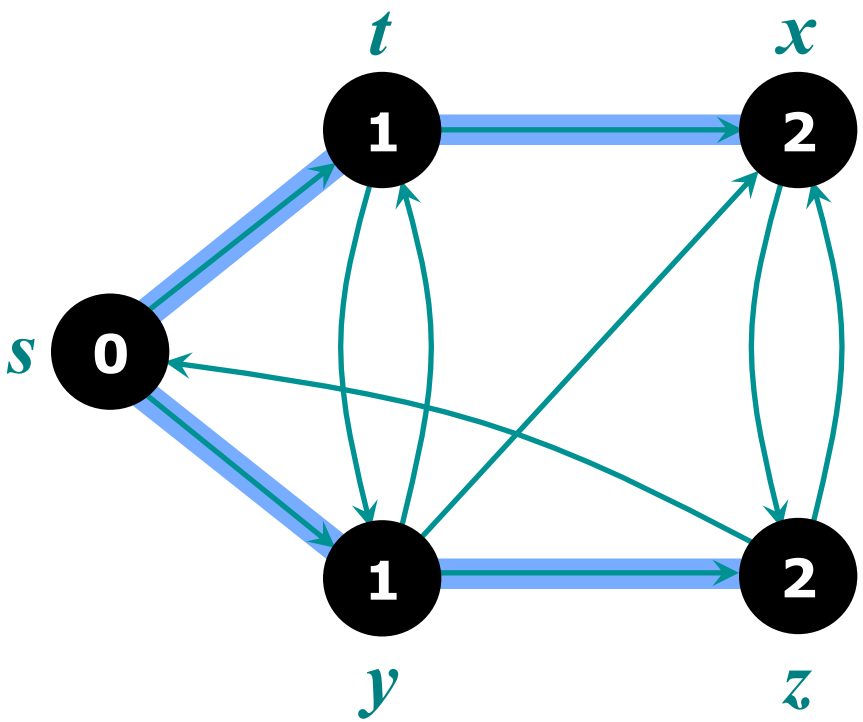
Example of breadth-first search



Example of breadth-first search



Example of breadth-first search



$Q \quad \emptyset$

Gray vertices

Shortest paths in weighted dag

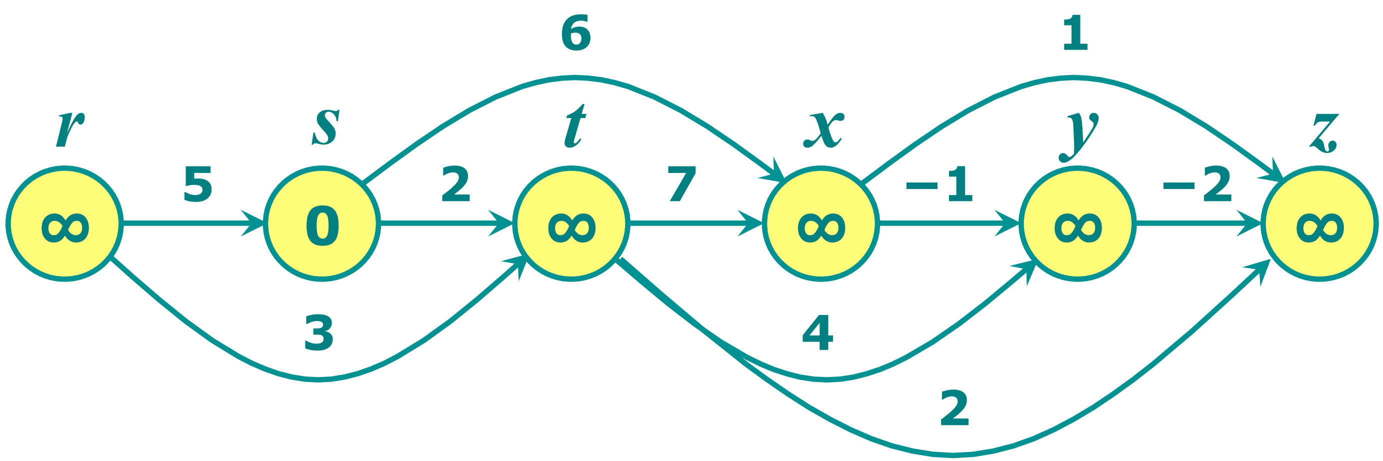
What is dag?

A dag is a directed ***acyclic*** *graph.*

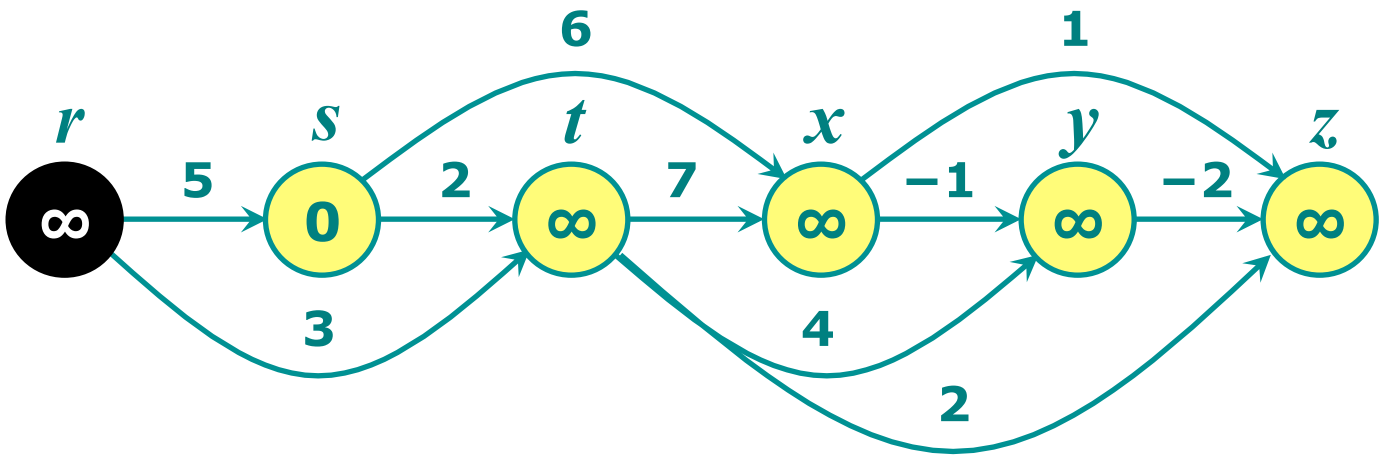
DAG-SHORTEST-PATH(G, w, s)

1. Topologically sort the vertices of G
2. **for** each vertex $v \in V[G]$
3. **do** $d[v] \leftarrow \infty$
4. $\pi(v) \leftarrow \text{NIL}$
5. $d[s] \leftarrow 0$
6. **for** each vertex u, taken in topologically sorted order
7. **do for** each vertex $v \in Adj[u]$
8. **do if** $d[v] > d[u] + w(u, v)$
9. **then** $d[v] \leftarrow d[u] + w(u, v)$
10. $\pi(v) \leftarrow u$

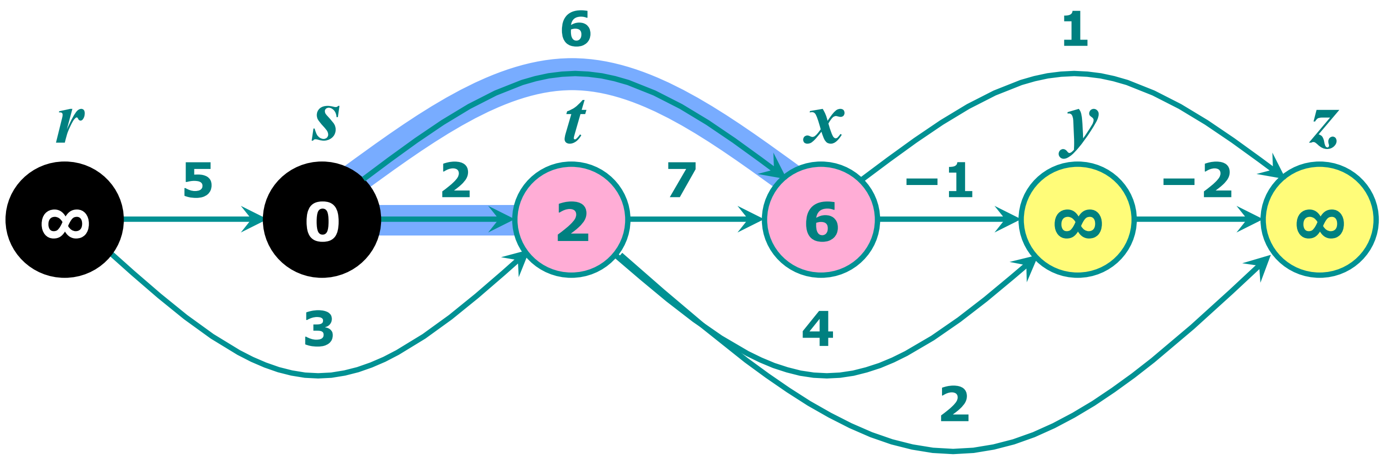
Example of dag shortest paths



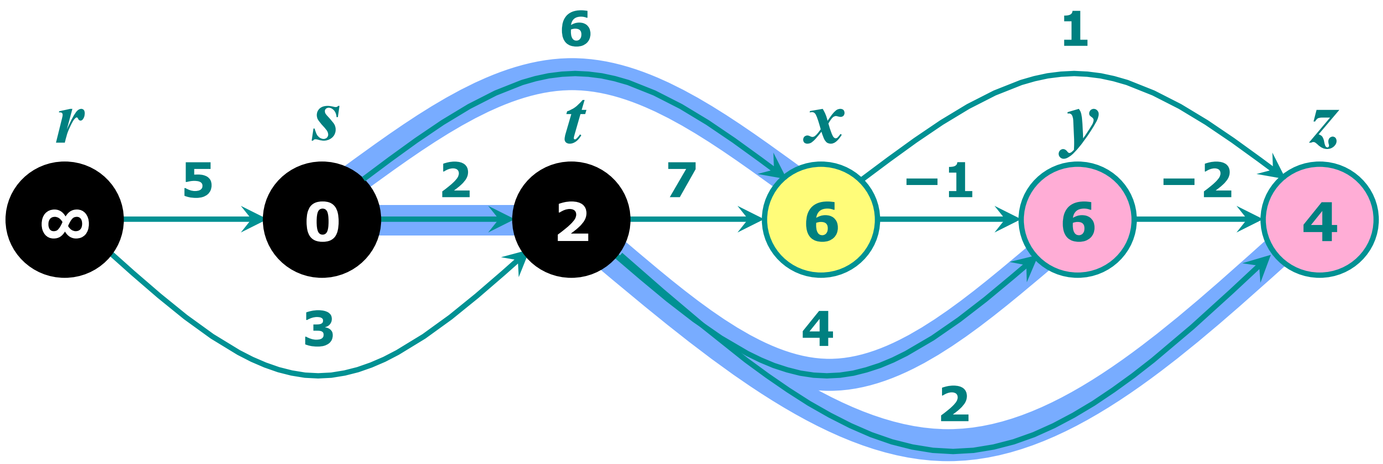
Example of dag shortest paths



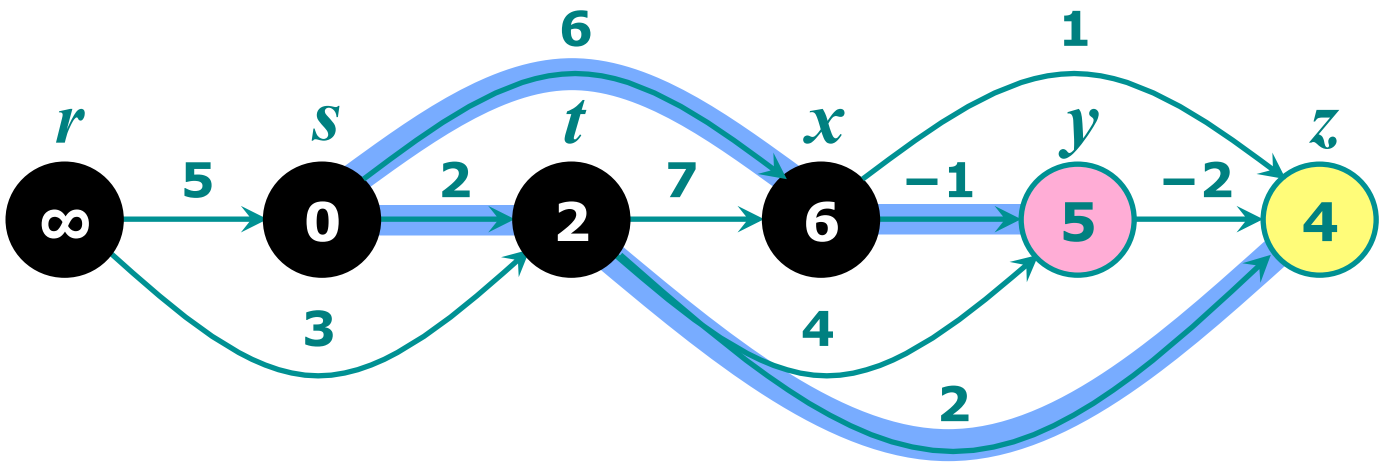
Example of dag shortest paths



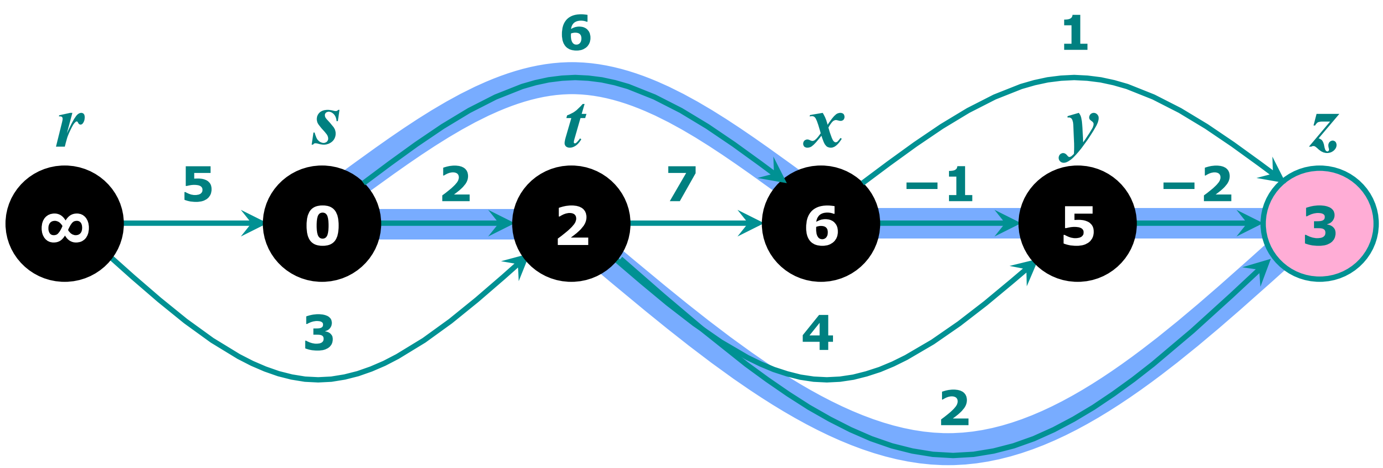
Example of dag shortest paths



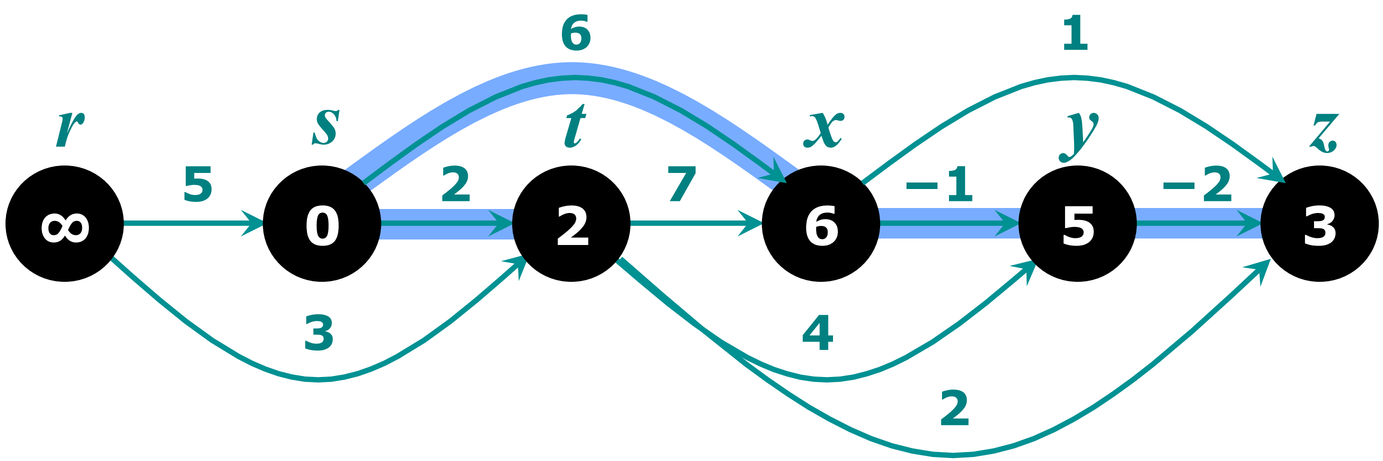
Example of dag shortest paths



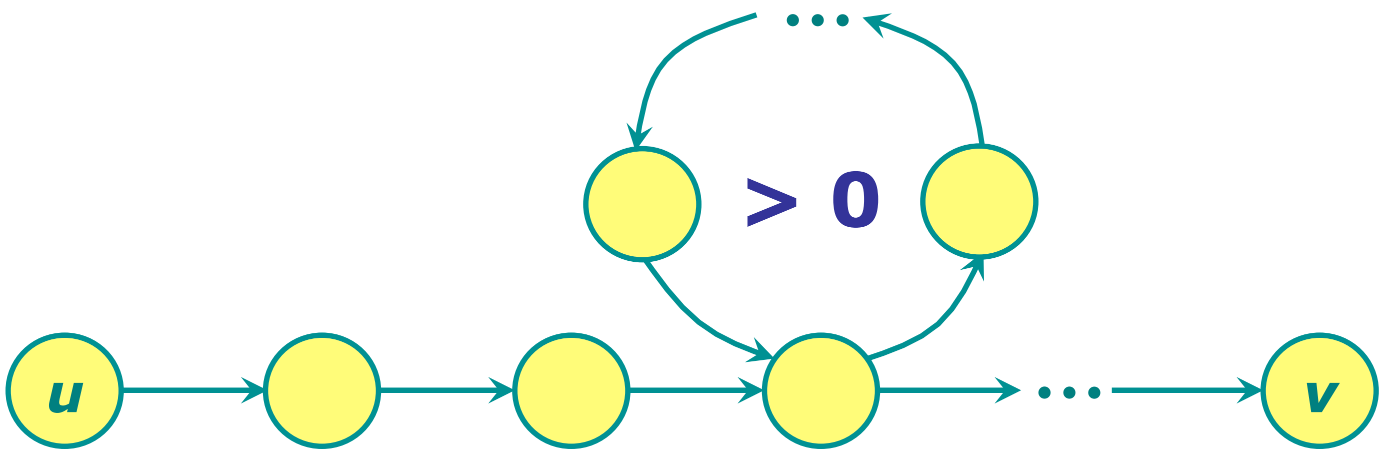
Example of dag shortest paths



Example of dag shortest paths

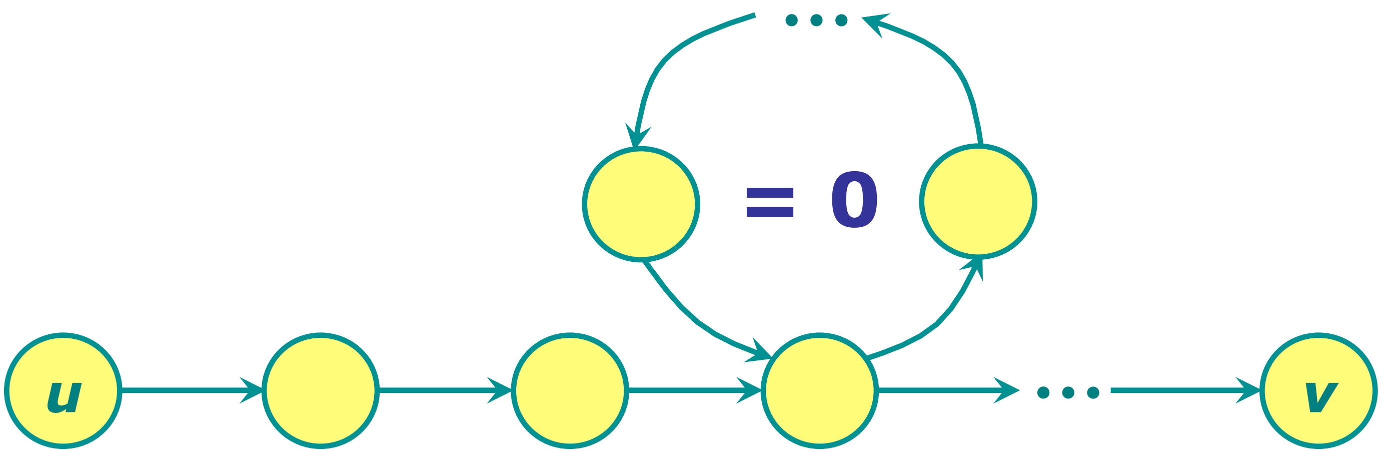


Positive-weight cycle



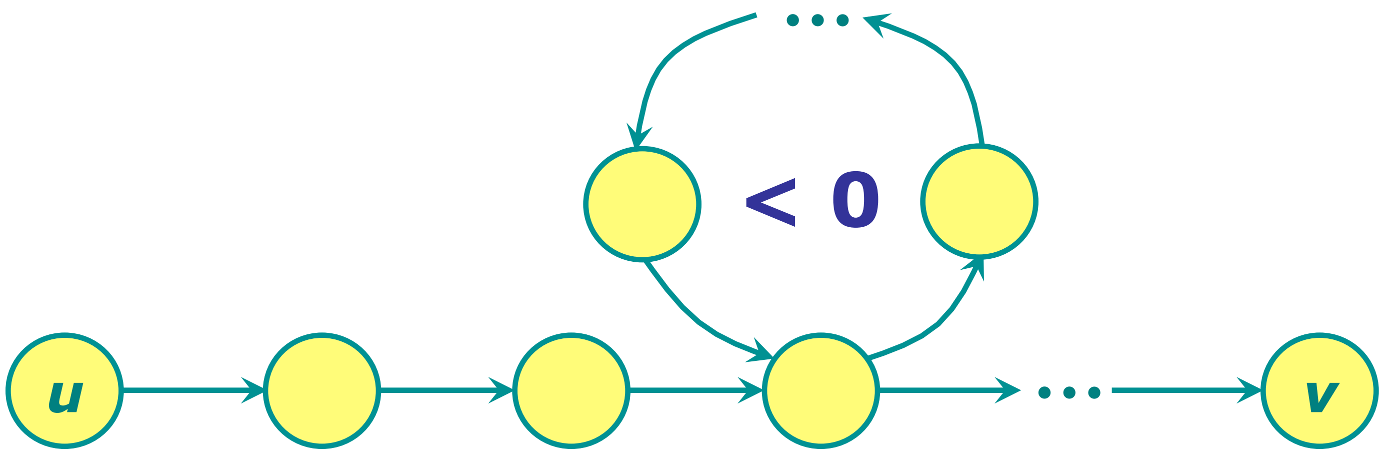
Shortest path cannot contain a positive-weight cycle.

0-weight cycle



0-*weight cycle can be removed from the shortest path.*

Negative-weight cycle

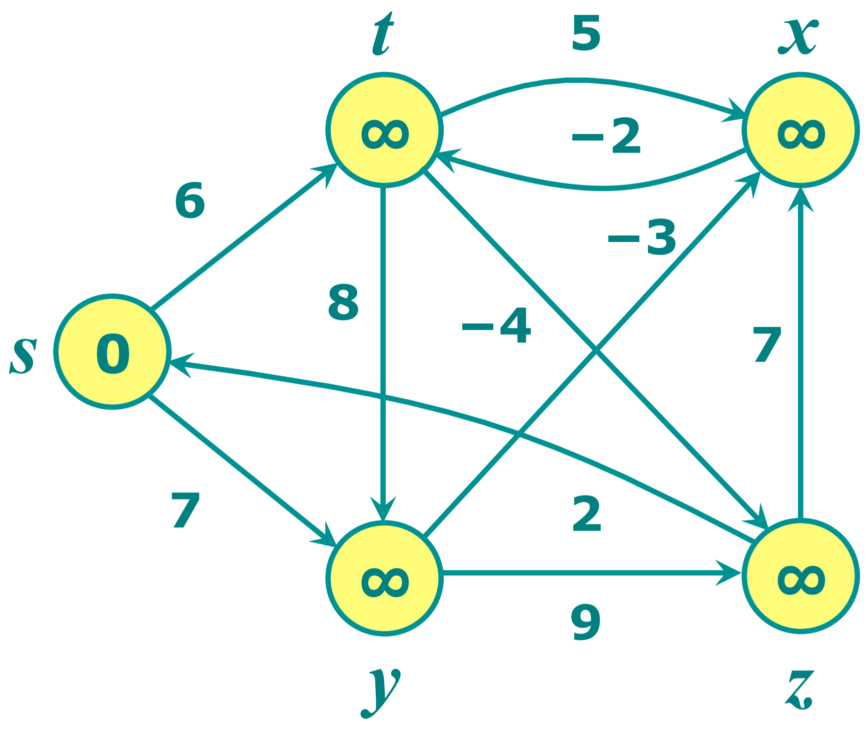


If a graph contains a negative-weight cycle, then some shortest paths may not exist.

Algorithm for negative weight cycle

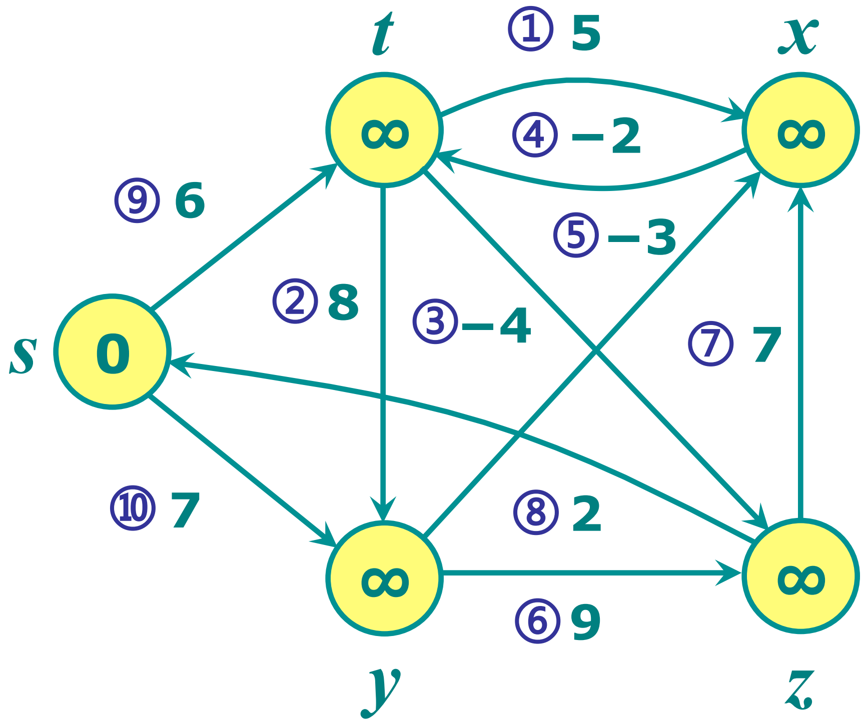
Bellman-Ford algorithm: Finds all shortest-path lengths from a source $s \in V$ to all $v \in V$ or determines that a negative-weight cycle exists.

Example of Bellman-Ford algorithm



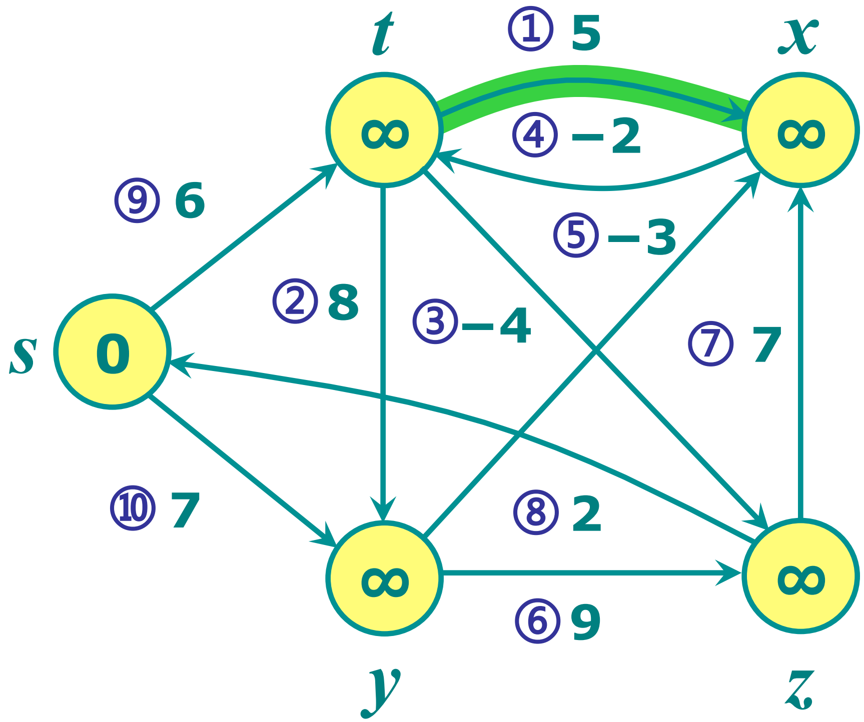
Initialization.

Example of Bellman-Ford algorithm

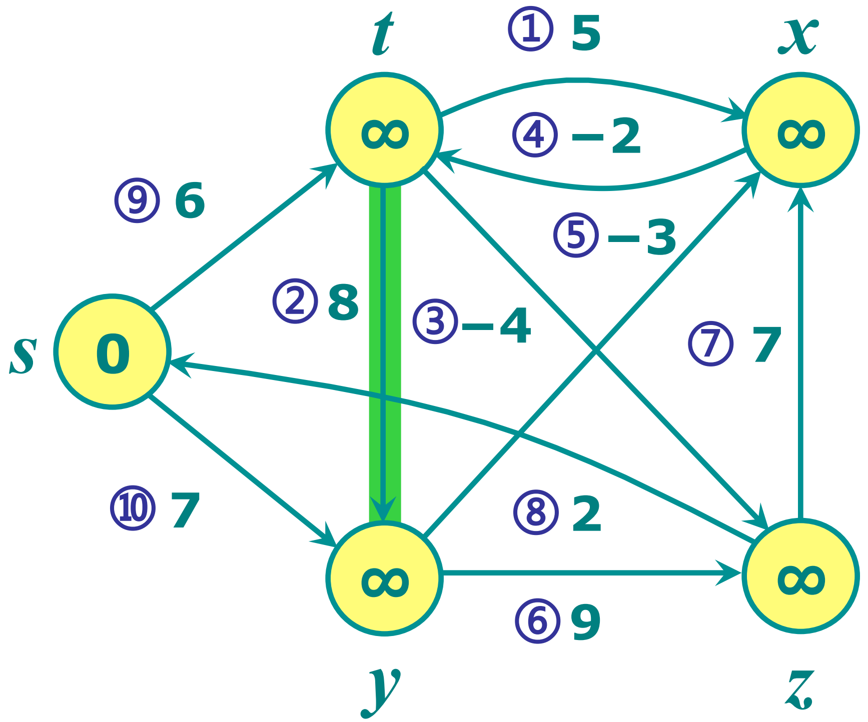


Order of edge relaxation.

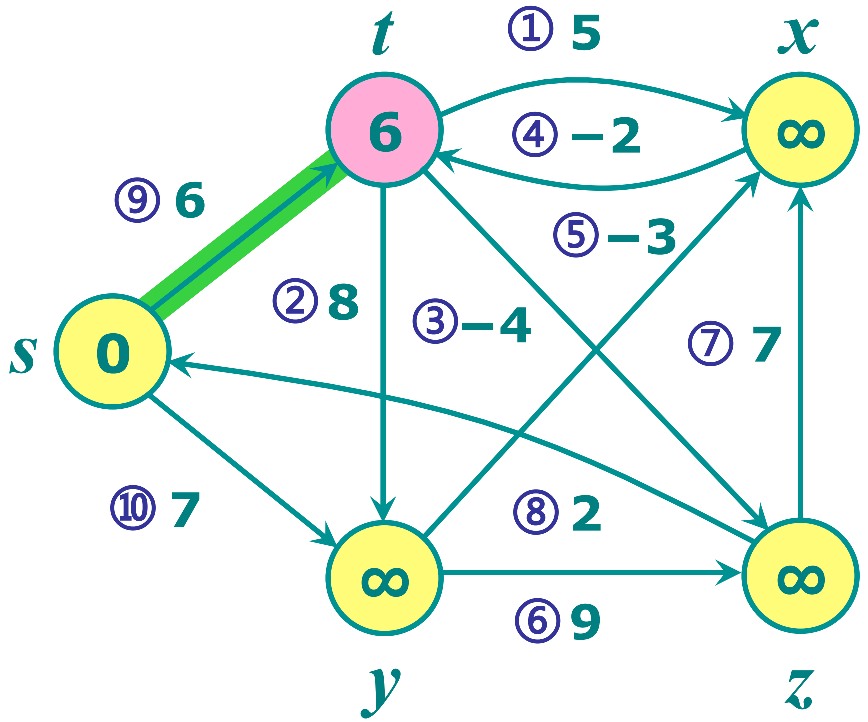
Example of Bellman-Ford algorithm



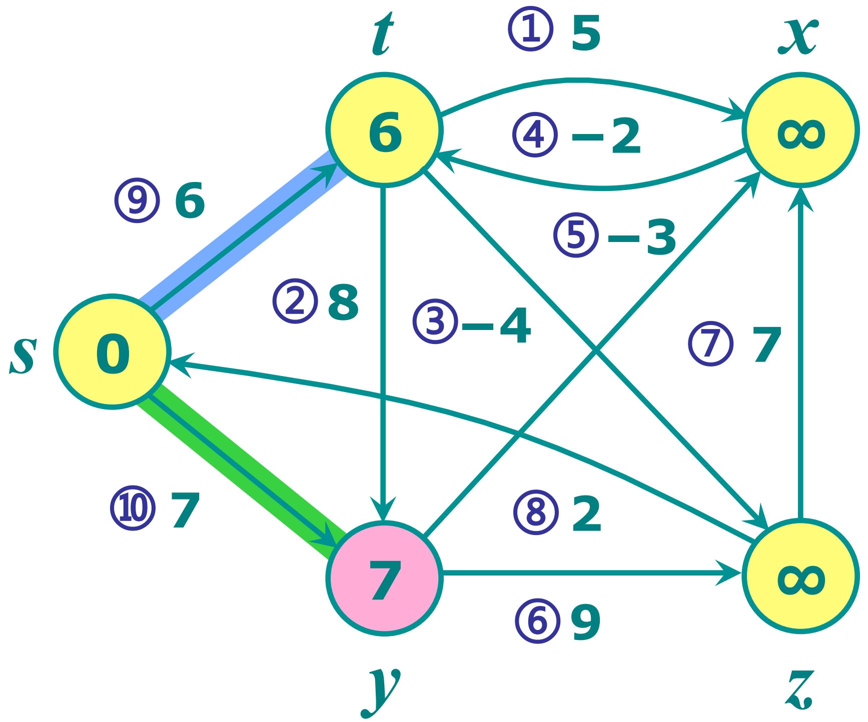
Example of Bellman-Ford algorithm



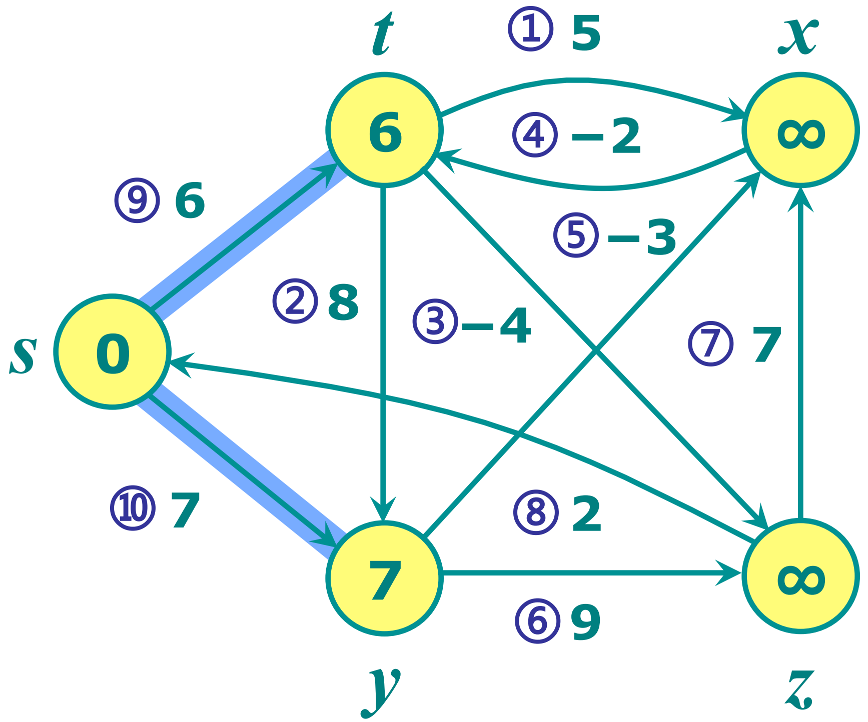
Example of Bellman-Ford algorithm



Example of Bellman-Ford algorithm

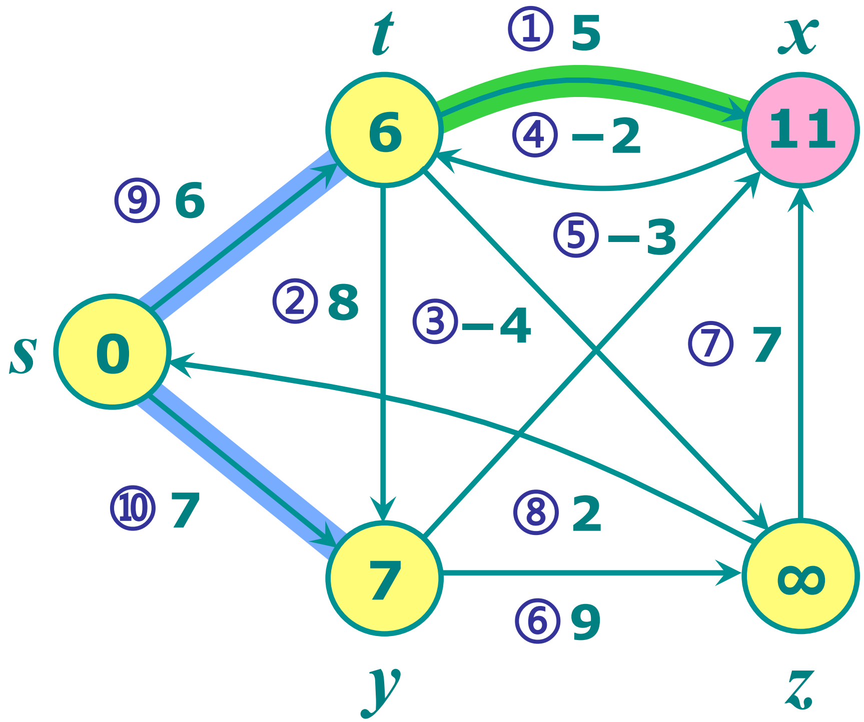


Example of Bellman-Ford algorithm

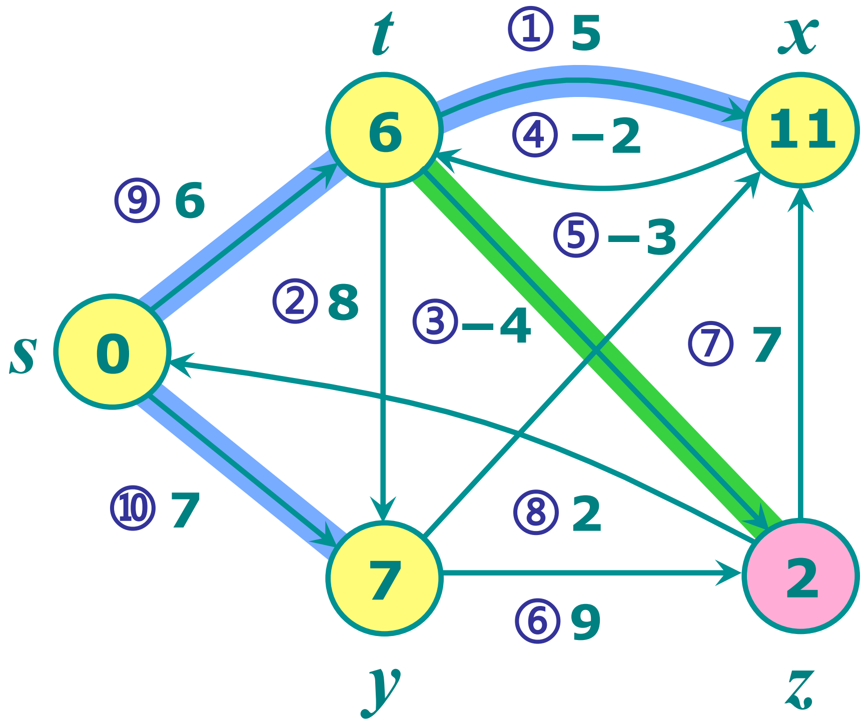


End of pass 1.

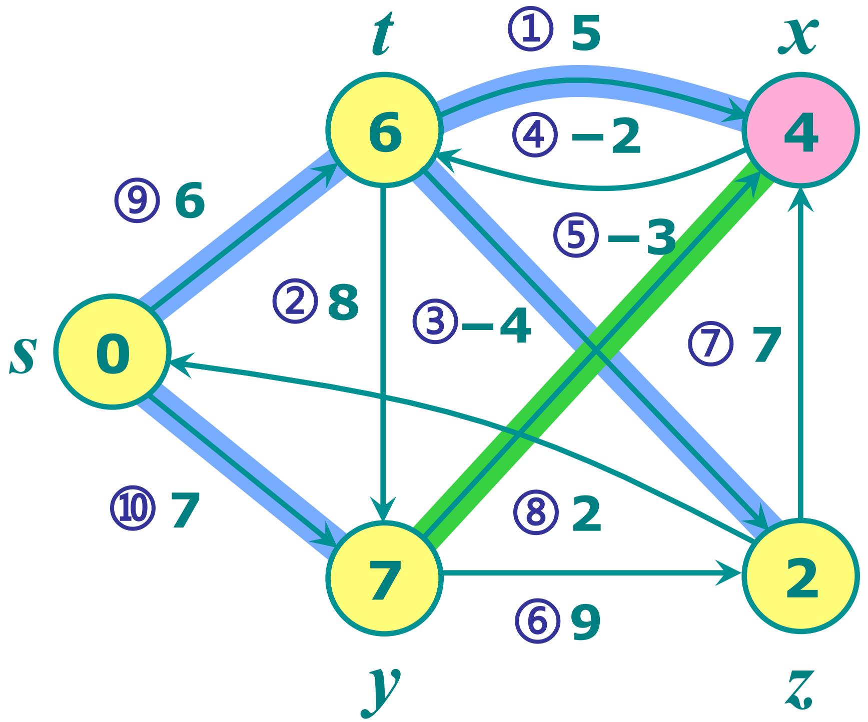
Example of Bellman-Ford algorithm



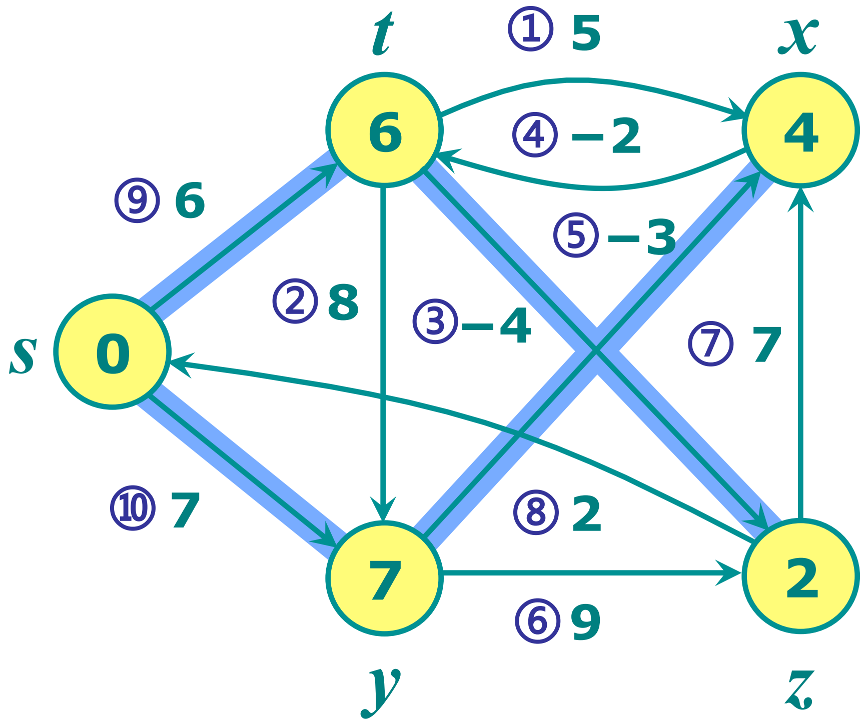
Example of Bellman-Ford algorithm



Example of Bellman-Ford algorithm

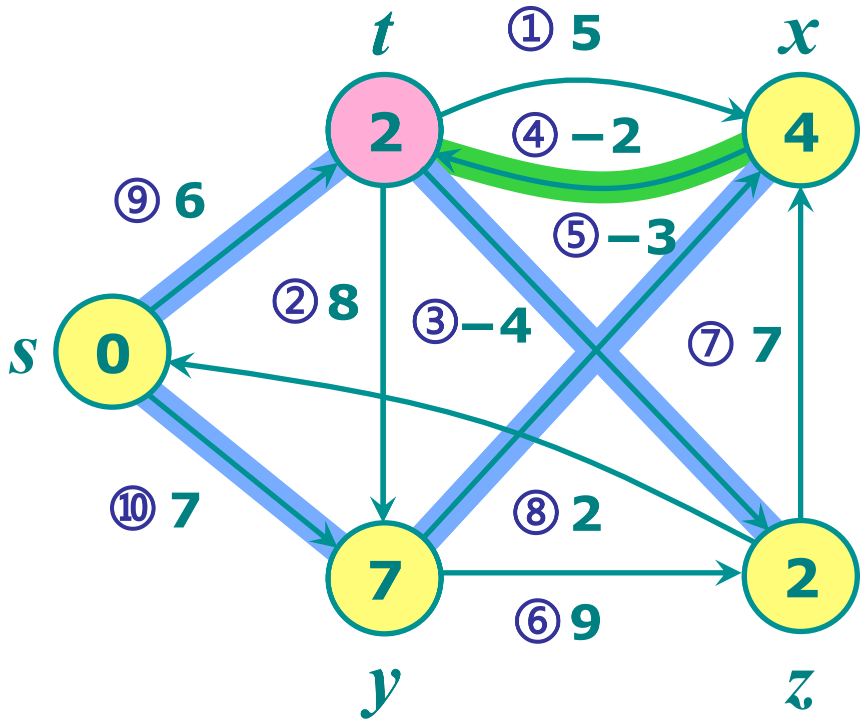


Example of Bellman-Ford algorithm

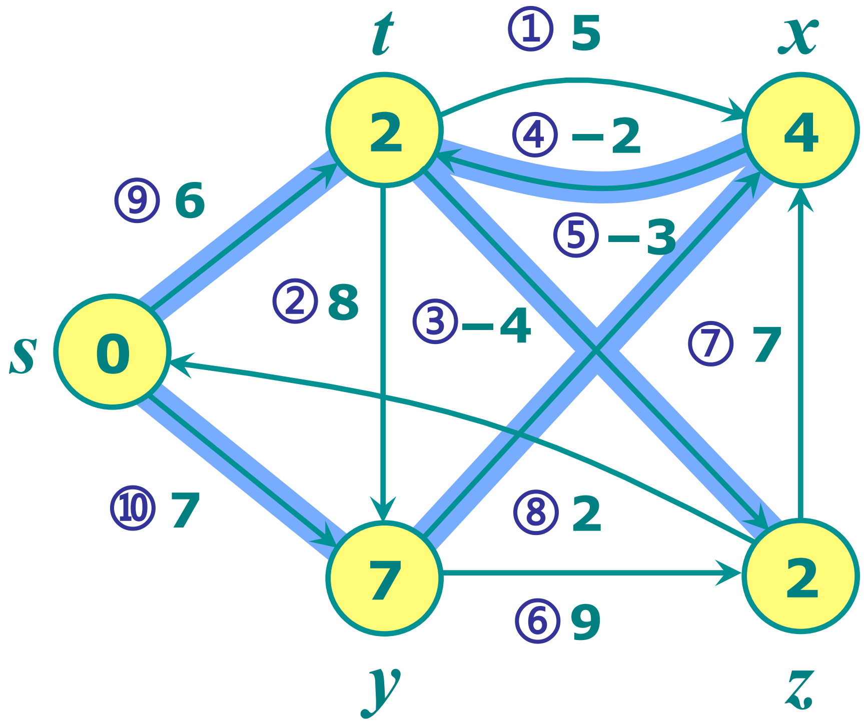


End of pass 2.

Example of Bellman-Ford algorithm

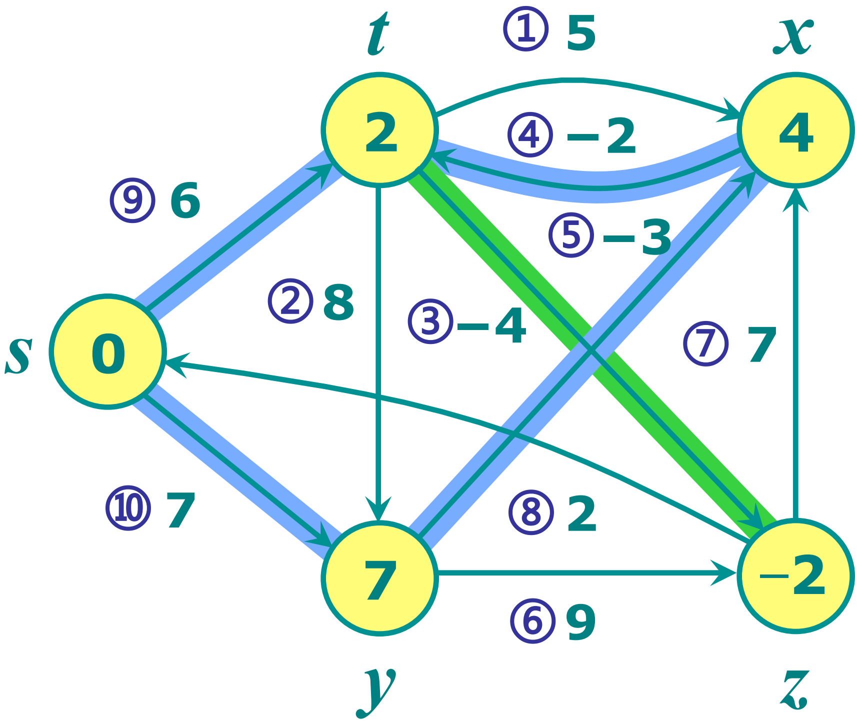


Example of Bellman-Ford algorithm

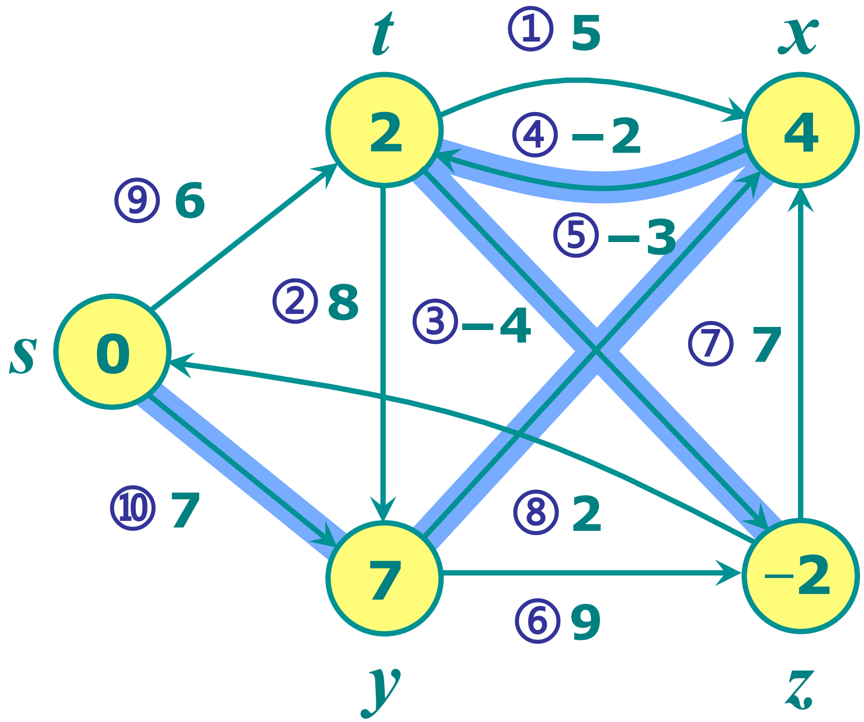


End of pass 3.

Example of Bellman-Ford algorithm

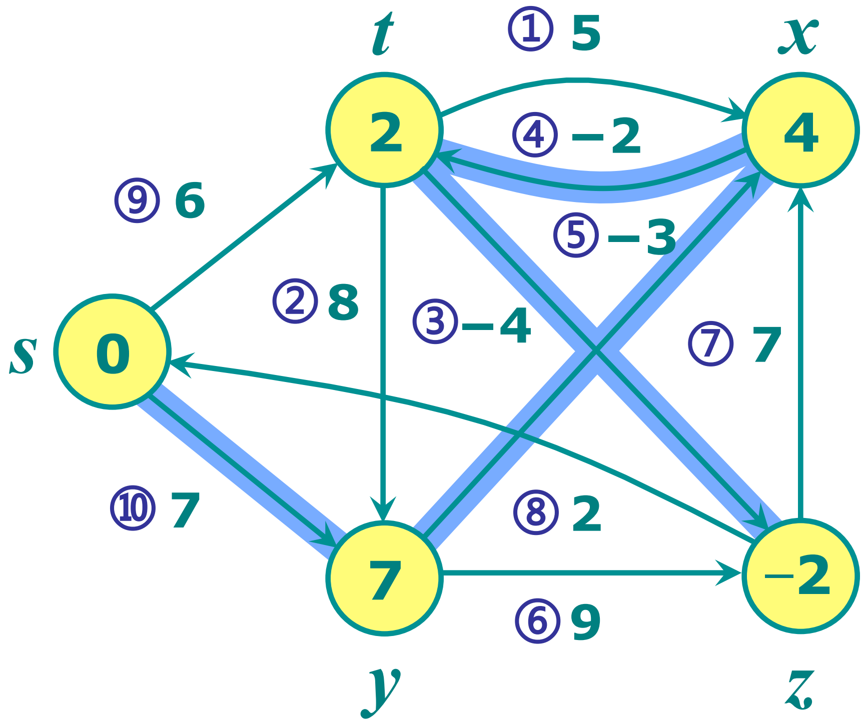


Example of Bellman-Ford algorithm



End of pass 4.

Example of Bellman-Ford algorithm



End

Bellman-Ford algorithm

BELLMAN-FORD(G, w, s)

for each vertex $v \in V[G]$

  do $d[v] \leftarrow \infty$

    $\pi(v) \leftarrow$ NIL

$d[s] \leftarrow 0$

for $i \leftarrow 1$ **to** $|V[G]| - 1$

  do for each edge $(u, v) \in E[G]$

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

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

        $\pi(v) \leftarrow u$

for each edge $(u, v) \in E[G]$

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

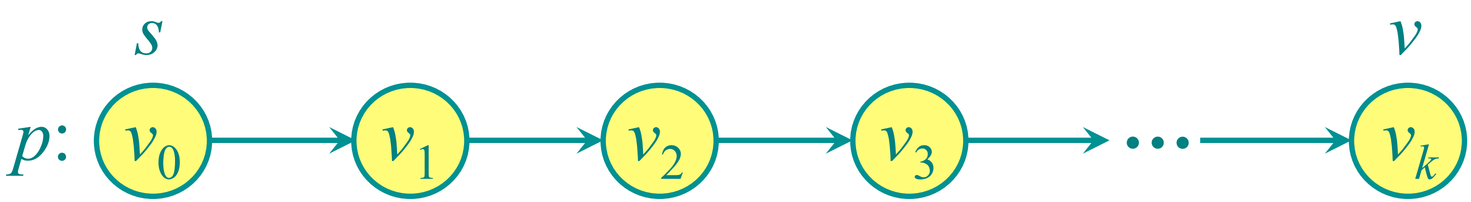
    then return FALSE

return TURE

Correctness of Bellman-Ford algorithm

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

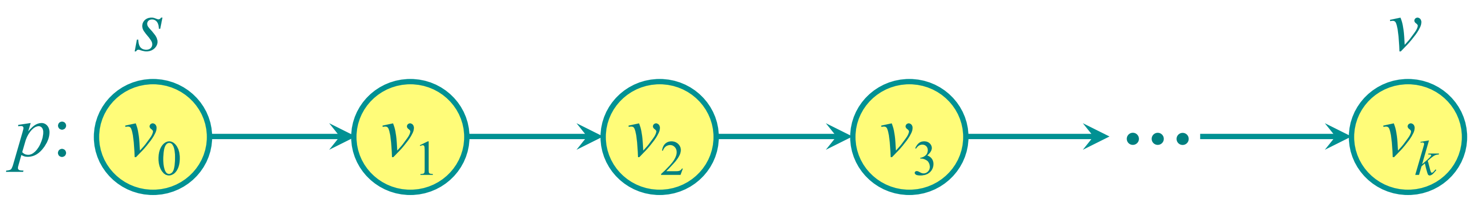
Proof. Let $v \in V$ be any vertex, and consider a shortest path p from s to v with the minimum number of edges.



Since p is a shortest path, we have

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

Correctness of Bellman-Ford algorithm



Initially, $d[v_0] = 0 = \delta(s, v_0)$,

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

$\vdots$

- 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 $\leq |V| - 1$ edge

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

Correctness of Bellman-Ford algorithm

Conversely, suppose that graph G contains a negative-weight cycle that is reachable from the source s; let this cycle be $c = v_0 \to v_1 \to \ldots \to v_k$, where $v_0 = v_k$. Then,

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$$

If the Bellman-Ford algorithm returns TRUE, then,

$d[v_i] \le d[v_{i-1}] + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$, and

$$\sum_{i=1}^{k} d[v_i] \le \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i))$$

$$= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Correctness of Bellman-Ford algorithm

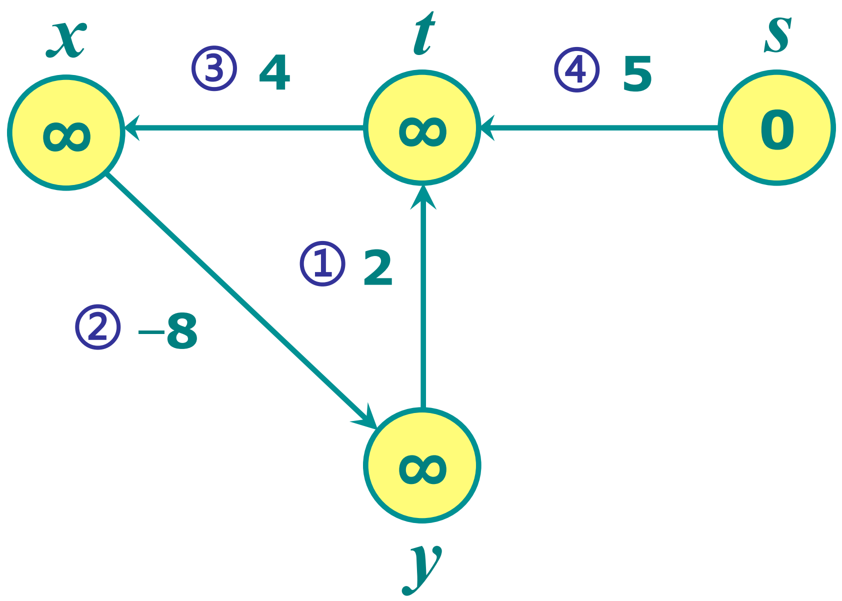
$$\sum_{i=1}^{k} d[v_i] \le \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Since $v_0 = v_k$, and so,

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$$, thus,

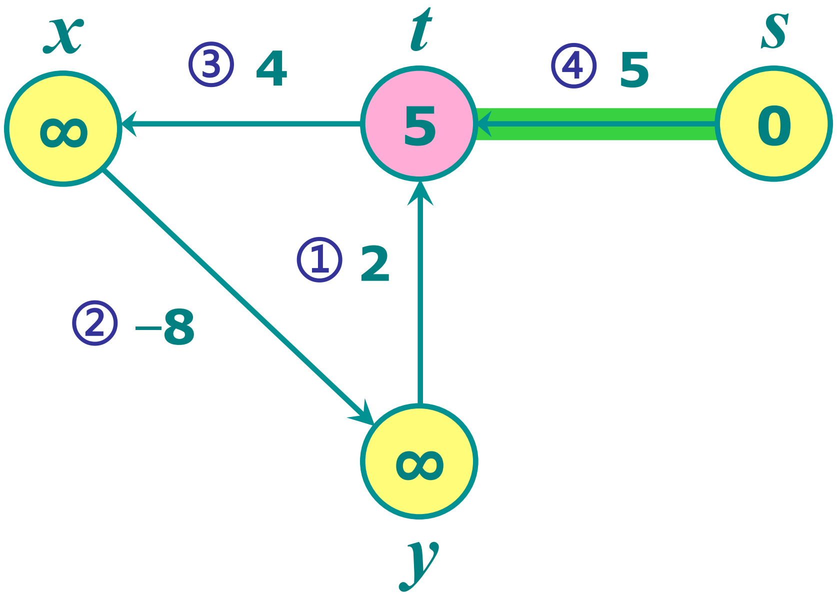
$0 \le \sum_{i=1}^{k} w(v_{i-1}, v_i)$ contradicts with $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$ □

Example of negative-weight cycle

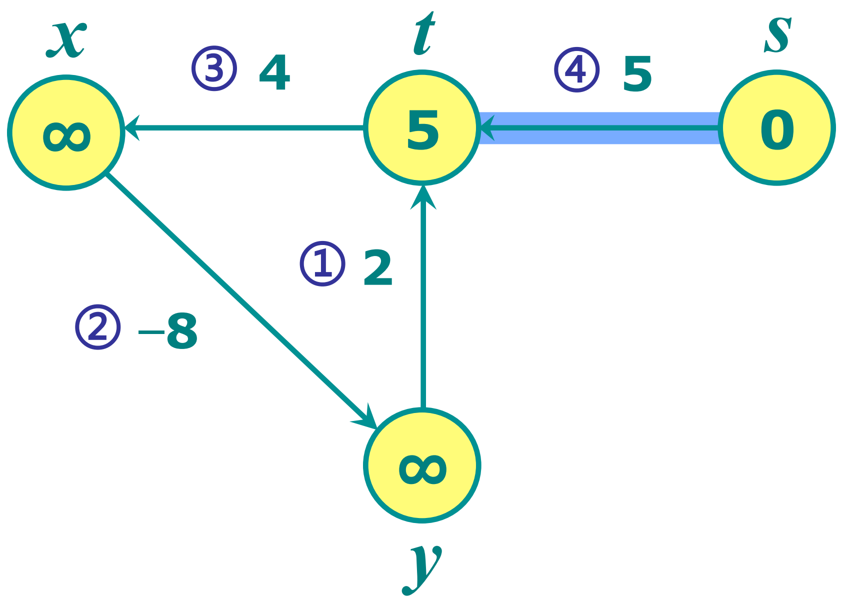


Initialization and order of edge relaxation.

Example of negative-weight cycle

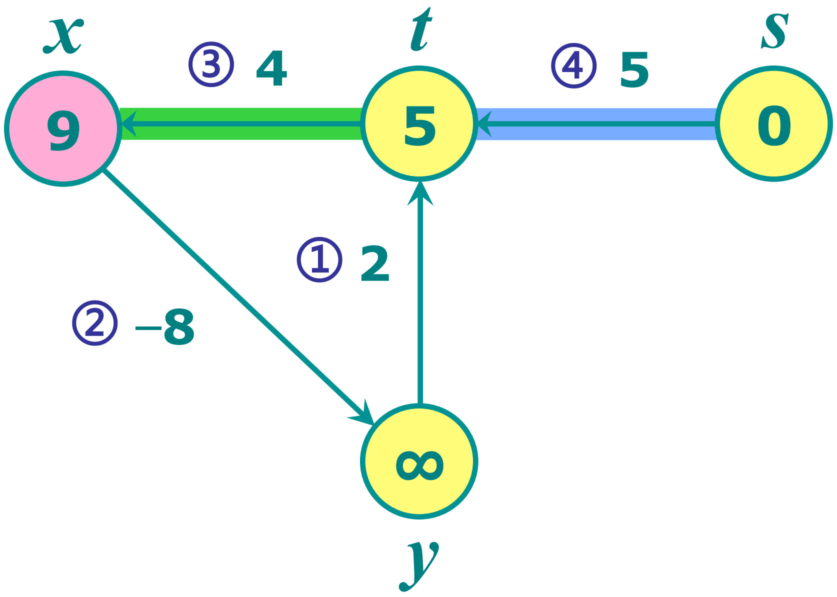


Example of negative-weight cycle

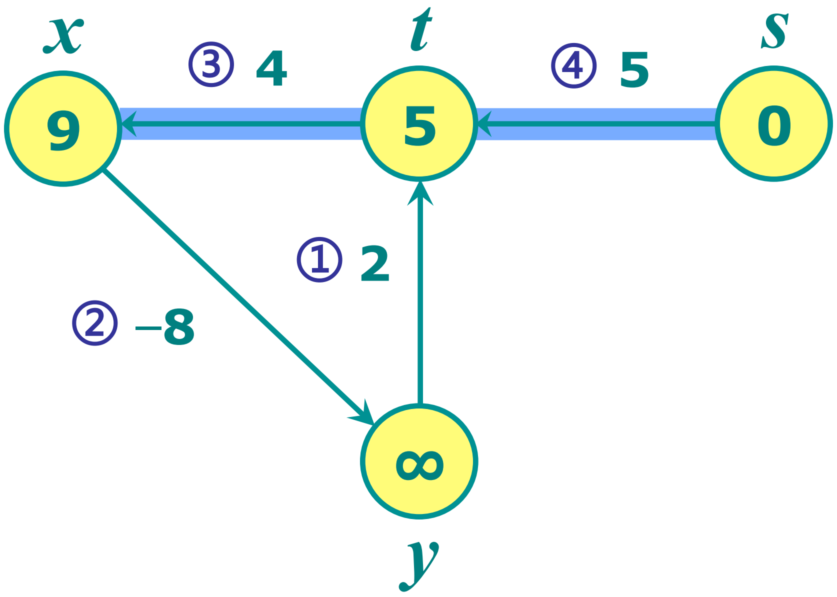


End of pass 1.

Example of negative-weight cycle

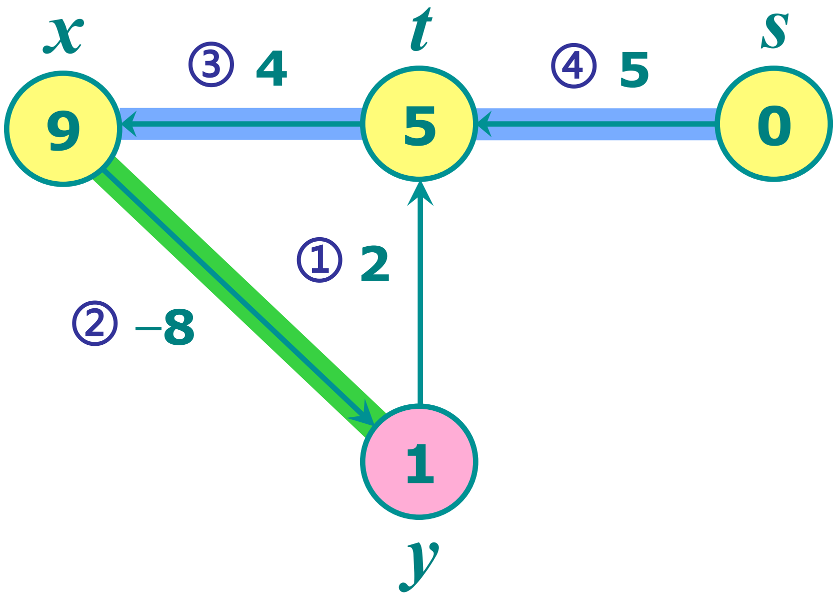


Example of negative-weight cycle

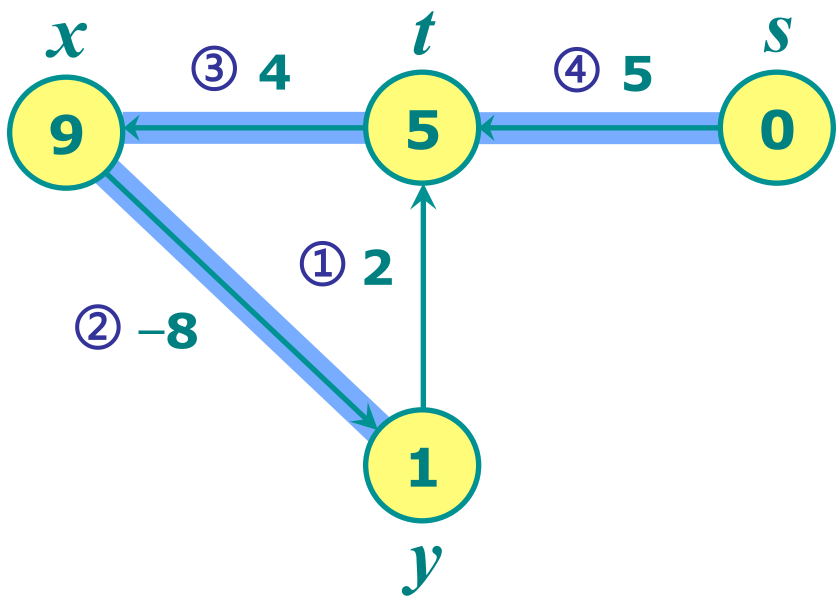


End of pass 2.

Example of negative-weight cycle

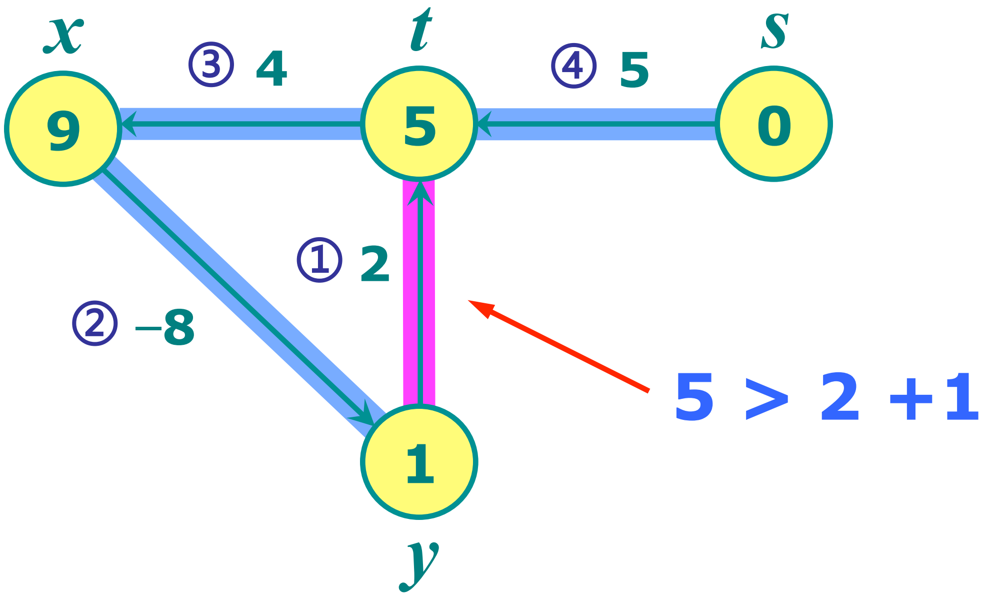


Example of negative-weight cycle



End of pass 3.

Example of negative-weight cycle



End.

Single-source shortest-paths algorithms

Algorithms	**Unweighted**	**Positive**	**Negative**	**Cycle**	**Time**
Breadth-first search	○	×	×	○	$O(V+E)$
Dag shortest paths	○	○	○	×	$O(V+E)$
Dijkstra	○	○	×	○	$O(E\lg V)$
Bellman-Ford	○	○	○	○	$O(VE)$

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 = $O(V^2E)$.
- Dense graph (n^2 edges) $\Rightarrow \Theta(n^4)$ time in the worst case.

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

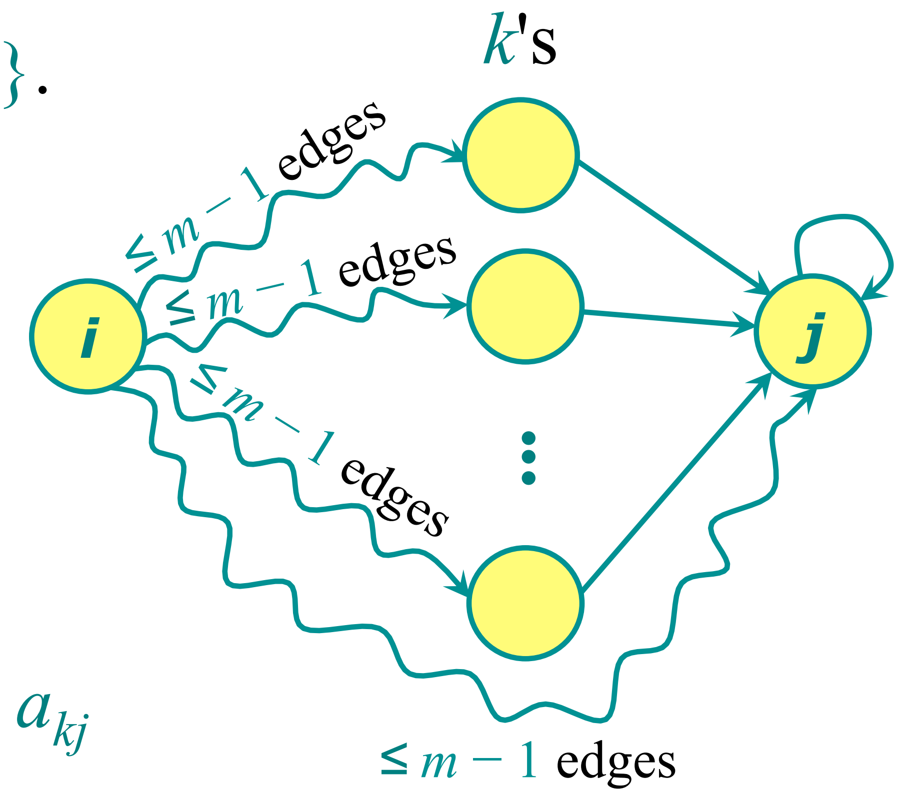
$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j, \\ \infty & \text{if } i \neq j. \end{cases}$$

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

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

Proof of claim

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



Relaxation!

for $k \leftarrow 1$ **to** n

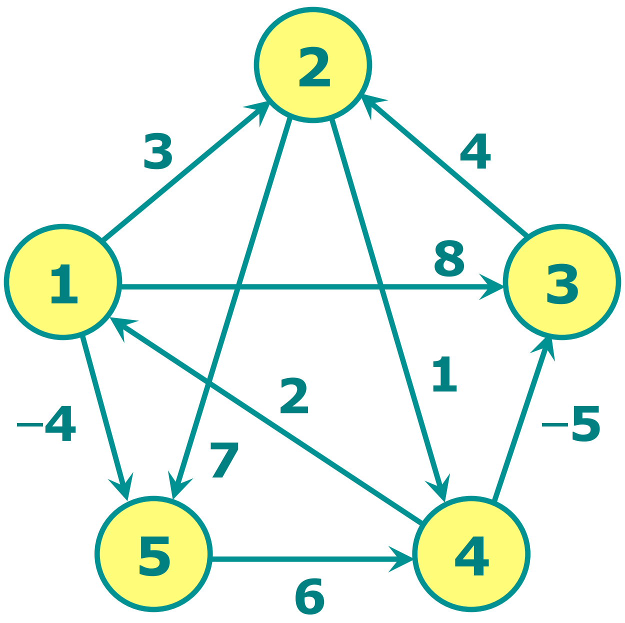
do if $d_{ij} > d_{ik} + a_{kj}$

then $d_{ij} \leftarrow d_{ik} + a_{kj}$

Note: No negative-weight cycles implies

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

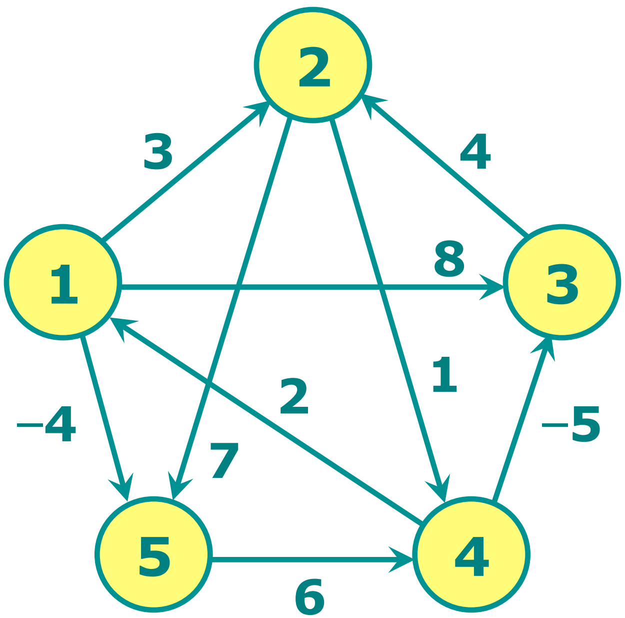
All-pairs shortest paths example



$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$\Pi^{(1)} = \begin{pmatrix} \varnothing & 1 & 1 & \varnothing & 1 \\ \varnothing & \varnothing & \varnothing & 2 & 2 \\ \varnothing & 3 & \varnothing & \varnothing & \varnothing \\ 4 & \varnothing & 4 & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & 5 & \varnothing \end{pmatrix}$$

All-pairs shortest paths example



$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

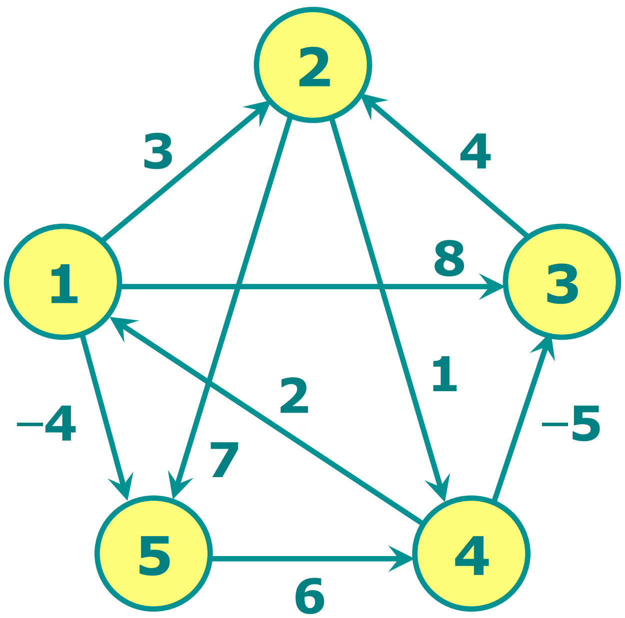
$$d_{ij}{}^{(m)} = \min_k \{d_{ik}{}^{(m-1)} + a_{kj}\}$$

$$d_{14}{}^{(2)} = \min_k \{d_{ik}{}^{(1)} + a_{kj}\}.$$

$$d_{14}{}^{(2)} = \min\{(d_{11}{}^{(1)} + a_{14}), (d_{12}{}^{(1)} + a_{24}), (d_{13}{}^{(1)} + a_{34}), (d_{14}{}^{(1)} + a_{44}), (d_{15}{}^{(1)} + a_{54})\}.$$

$$d_{14}{}^{(2)} = \min\{(0 + \infty), (3 + 1), (8 + \infty), (\infty + 0), (-4 + 6)\} = 2$$

All-pairs shortest paths example

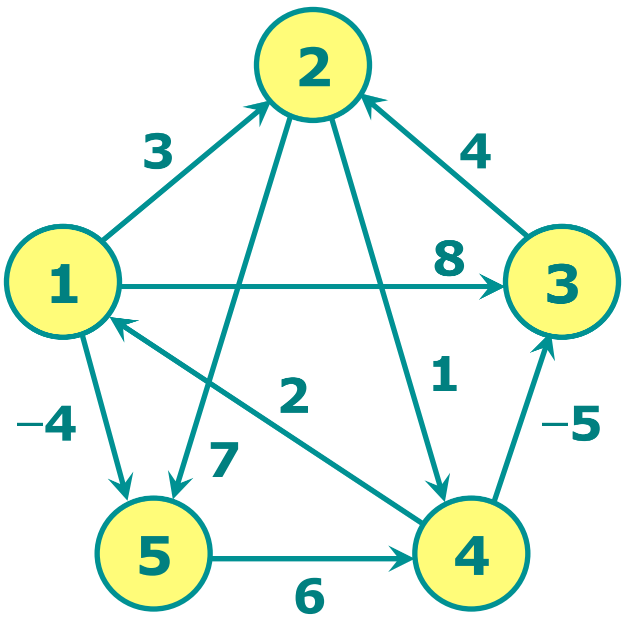


Length 2

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$\Pi^{(2)} = \begin{pmatrix} \varnothing & 1 & 1 & 5 & 1 \\ 4 & \varnothing & 4 & 2 & 2 \\ \varnothing & 3 & \varnothing & 2 & 2 \\ 4 & 3 & 4 & \varnothing & 1 \\ 4 & \varnothing & 4 & 5 & \varnothing \end{pmatrix}$$

All-pairs shortest paths example

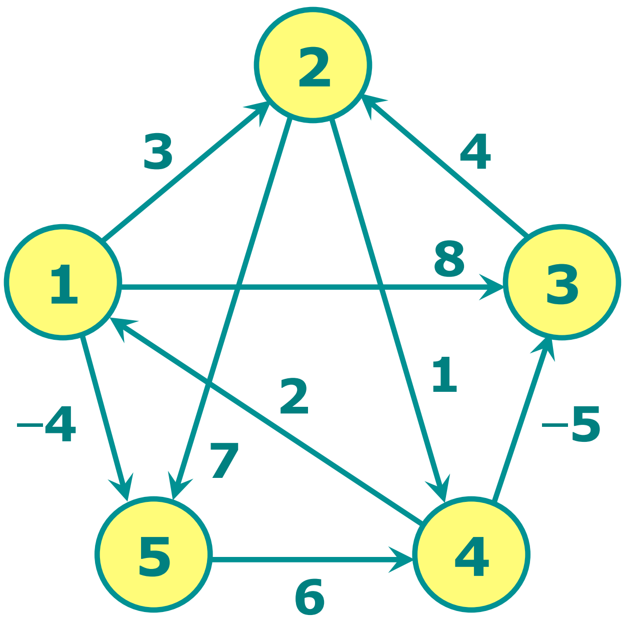


Length 3

$$D^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$\Pi^{(3)} = \begin{pmatrix} \varnothing & 1 & 4 & 5 & 1 \\ 4 & \varnothing & 4 & 2 & 1 \\ 4 & 3 & \varnothing & 2 & 2 \\ 4 & 3 & 4 & \varnothing & 1 \\ 4 & 3 & 4 & 5 & \varnothing \end{pmatrix}$$

All-pairs shortest paths example

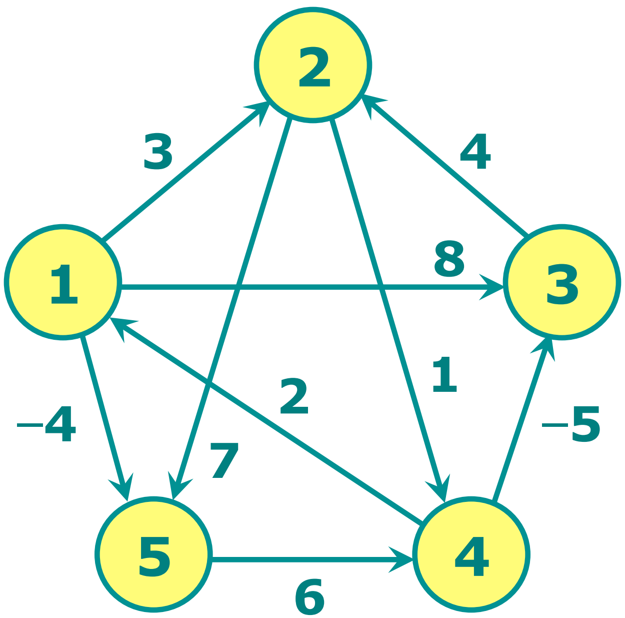


Length 4

$$D^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$\Pi^{(4)} = \begin{pmatrix} \varnothing & 3 & 4 & 5 & 1 \\ 4 & \varnothing & 4 & 2 & 1 \\ 4 & 3 & \varnothing & 2 & 1 \\ 4 & 3 & 4 & \varnothing & 1 \\ 4 & 3 & 4 & 5 & \varnothing \end{pmatrix}$$

All-pairs shortest paths example



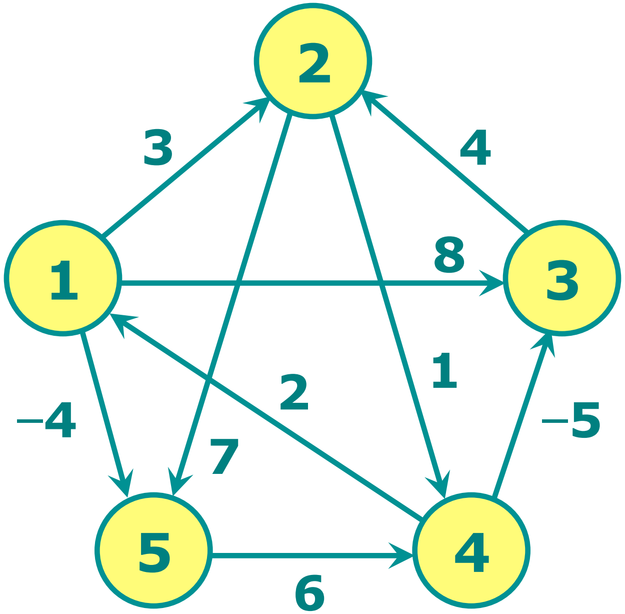
$$D^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$Path_{12} = 1 \to 5 \to 4 \to 3 \to 2$$

$$\Pi^{(4)} = \begin{pmatrix} \varnothing & 3 & 4 & 5 & 1 \\ 4 & \varnothing & 4 & 2 & 1 \\ 4 & 3 & \varnothing & 2 & 1 \\ 4 & 3 & 4 & \varnothing & 1 \\ 4 & 3 & 4 & 5 & \varnothing \end{pmatrix}$$

$$d_{12}^{(4)} = -4 + 6 + -5 + 4$$
$$= 1$$

All-pairs shortest paths example



$$D^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$\Pi^{(4)} = \begin{pmatrix} \varnothing & 3 & 4 & 5 & 1 \\ 4 & \varnothing & 4 & 2 & 1 \\ 4 & 3 & \varnothing & 2 & 1 \\ 4 & 3 & 4 & \varnothing & 1 \\ 4 & 3 & 4 & 5 & \varnothing \end{pmatrix}$$

$$Path_{35} = 3 \rightarrow 2 \rightarrow 4 \rightarrow 1 \rightarrow 5$$

$$\begin{aligned} d_{35}^{(4)} &= 4 + 1 + 2 + -4 \\ &= 3 \end{aligned}$$

Matrix multiplication

All-pairs shortest paths for graph $G = (V, E)$.

- Dynamic programming: compute $D^{(|V| - 1)}$.
- $d_{ij}^{(m)} = \min_k \{d_{ik}^{(m-1)} + a_{kj}\}$.

Observe that if we make the substitutions

$$
\begin{aligned}
D^{(m-1)} &\rightarrow a, \\
D^{(0)} &\rightarrow b, \\
D^{(m)} &\rightarrow \mathrm{c}, \\
\min &\rightarrow +, \\
+ &\rightarrow \cdot \,.
\end{aligned}
$$

Problem change to compute $C = A \cdot B$, where C, A, and B are $n \times n$ matrices:

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

Matrix multiplication

Consequently, we can compute

$$D^{(1)} = D^{(0)} \cdot D^{(0)}$$
$$D^{(2)} = D^{(1)} \cdot D^{(0)}$$
$$\vdots$$
$$D^{(n-1)} = D^{(n-2)} \cdot D^{(0)}$$

Yielding $D^{(n-1)} = \delta(i, j)$. Time $= \Theta(n \cdot n^3) = (n^4)$. No better than running Bellman-Ford once from each vertex.

Powering a number

Problem: Compute a^n, where $n \in \mathbb{N}$

Naive algorithm: $\Theta(n)$.

Divide-and-conquer algorithm:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even;} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd.} \end{cases}$$

$$T(n) = T(n/2) + \Theta(1) \;\Rightarrow\; T(n) = \Theta(lgn)$$

Improved matrix multiplication

Repeated squaring: $D^{(2k)} = D^{(k)} \cdot D^{(k)}$.

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

Note: $D^{(n-1)} = D^{(n)} = D^{(n+1)} = \cdots$.

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

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

Floyd-Warshall algorithm

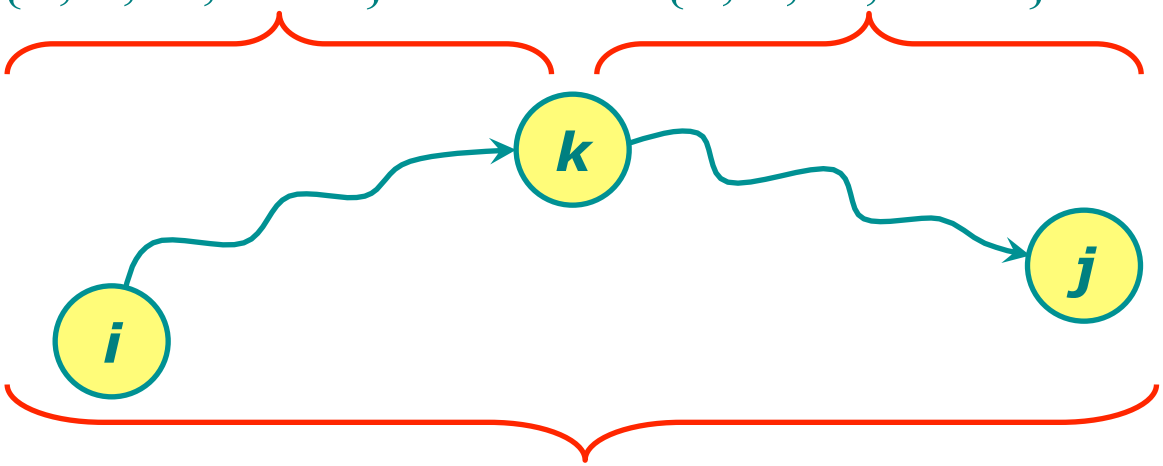
Also dynamic programming, but faster!

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

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

Floyd-Warshall algorithm

all intermediate vertices in $\{1, 2, ..., k-1\}$ all intermediate vertices in $\{1, 2, ..., k-1\}$

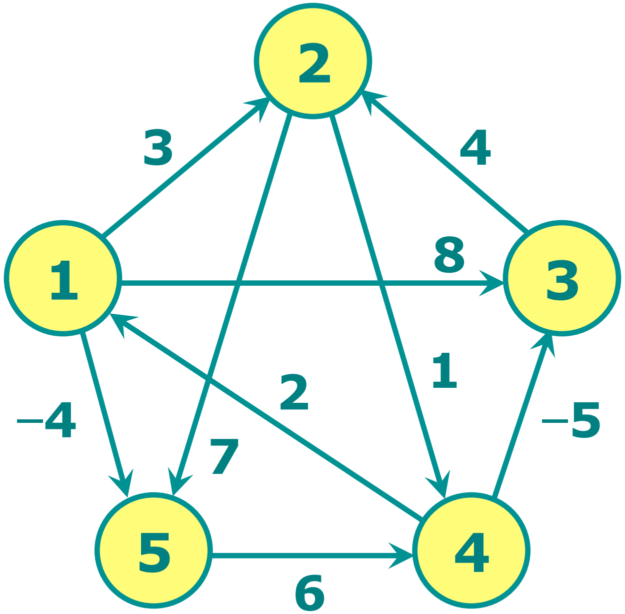


p: all intermediate vertices in $\{1, 2, ..., k\}$

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1. \end{cases}$$

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

Example of Floyd-Warshall algorithm

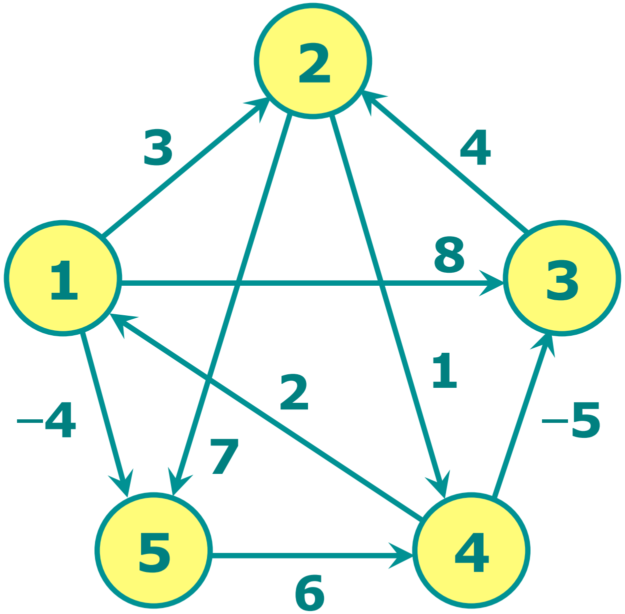


Initialization

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$\Pi^{(0)} = \begin{pmatrix} \varnothing & 1 & 1 & \varnothing & 1 \\ \varnothing & \varnothing & \varnothing & 2 & 2 \\ \varnothing & 3 & \varnothing & \varnothing & \varnothing \\ 4 & \varnothing & 4 & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & 5 & \varnothing \end{pmatrix}$$

Example of Floyd-Warshall algorithm

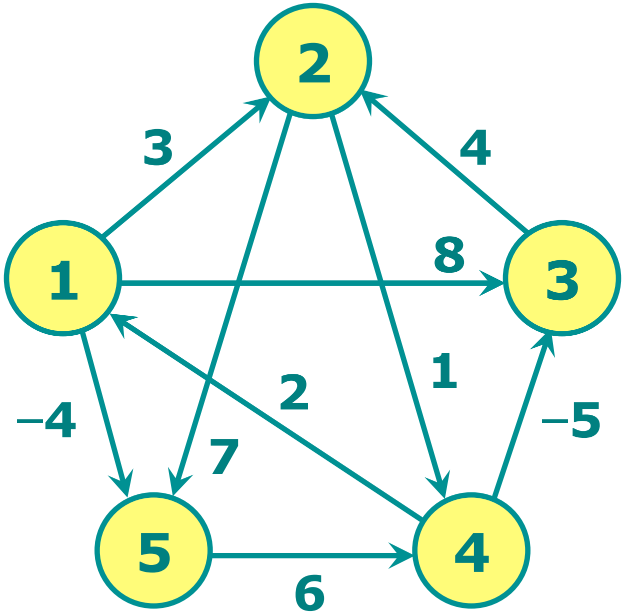


Node 1

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$\Pi^{(1)} = \begin{pmatrix} \varnothing & 1 & 1 & \varnothing & 1 \\ \varnothing & \varnothing & \varnothing & 2 & 2 \\ \varnothing & 3 & \varnothing & \varnothing & \varnothing \\ 4 & 1 & 4 & \varnothing & 1 \\ \varnothing & \varnothing & \varnothing & 5 & \varnothing \end{pmatrix}$$

Example of Floyd-Warshall algorithm

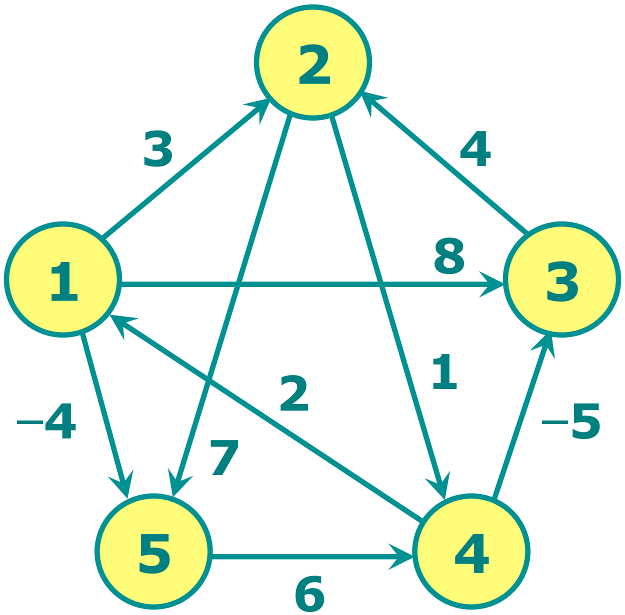


Node 2

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$\Pi^{(2)} = \begin{pmatrix} \varnothing & 1 & 1 & 2 & 1 \\ \varnothing & \varnothing & \varnothing & 2 & 2 \\ \varnothing & 3 & \varnothing & 2 & 2 \\ 4 & 1 & 4 & \varnothing & 1 \\ \varnothing & \varnothing & \varnothing & 5 & \varnothing \end{pmatrix}$$

Example of Floyd-Warshall algorithm

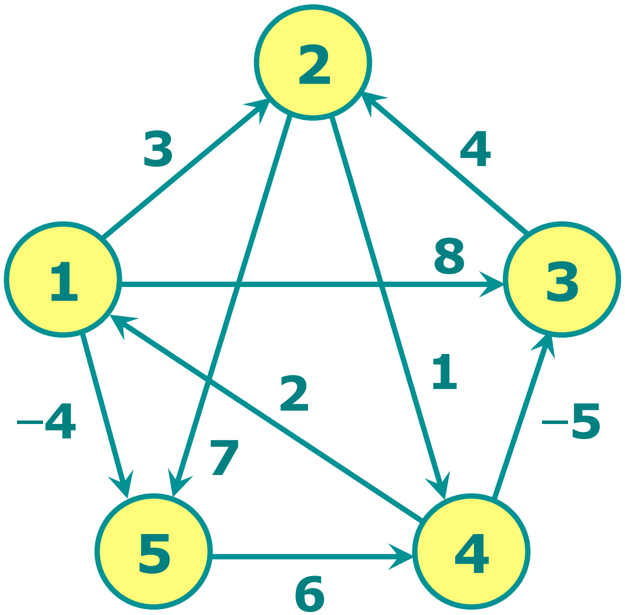


Node 3

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$\Pi^{(3)} = \begin{pmatrix} \varnothing & 1 & 1 & 2 & 1 \\ \varnothing & \varnothing & \varnothing & 2 & 2 \\ \varnothing & 3 & \varnothing & 2 & 2 \\ 4 & 3 & 4 & \varnothing & 1 \\ \varnothing & \varnothing & \varnothing & 5 & \varnothing \end{pmatrix}$$

Example of Floyd-Warshall algorithm



Node 4

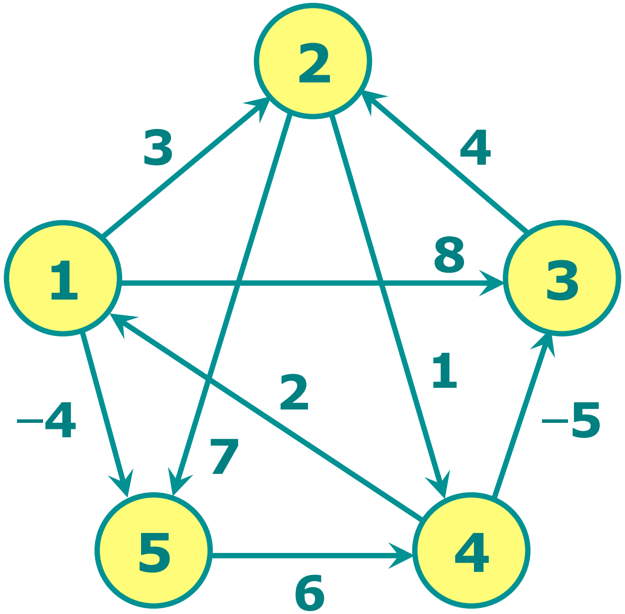
$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$\Pi^{(4)} = \begin{pmatrix} \varnothing & 1 & 4 & 2 & 1 \\ 4 & \varnothing & 4 & 2 & 4 \\ 4 & 3 & \varnothing & 2 & 4 \\ 4 & 3 & 4 & \varnothing & 1 \\ 4 & 4 & 4 & 5 & \varnothing \end{pmatrix}$$

$$Path_{13} = 1 \rightarrow 2 \rightarrow 4 \rightarrow 3$$

$$d_{13}^{(4)} = 3 + 1 + -5 = -1$$

Example of Floyd-Warshall algorithm



Node 5

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$\Pi^{(5)} = \begin{pmatrix} \varnothing & 5 & 4 & 5 & 1 \\ 4 & \varnothing & 4 & 2 & 4 \\ 4 & 3 & \varnothing & 2 & 4 \\ 4 & 3 & 4 & \varnothing & 1 \\ 4 & 4 & 4 & 5 & \varnothing \end{pmatrix}$$

Floyd-Warshall algorithm

FLOYD-WARSHALL(W)

1. $n \leftarrow rows[W]$
2. $d_0 \leftarrow W$
3. **for** $k \leftarrow 1$ **to** n
4. **do for** $i \leftarrow 1$ **to** n
5. **do for** $j \leftarrow 1$ **to** n
6. **do** $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
7. **return** $D^{(n)}$

Running time is $\Phi(n^3)$

Graph reweighting

Is it any possible to change all ***negative-weight*** edges to ***nonnegative***?

Then, we can use ***Dijkstra's algorithm*** to compute the shortest path for all edges.

Graph reweighting properties.

- For all pairs of vertices $u, v \in V$, a path p is a shortest path from u to v using weight function w if and only if p is also a shortest path from u to v using weight function $\hat{w}$ after reweighting.
- For all edges (u, v), the new weight $\hat{w}(u,v)$ is nonnegative.

Graph reweighting

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

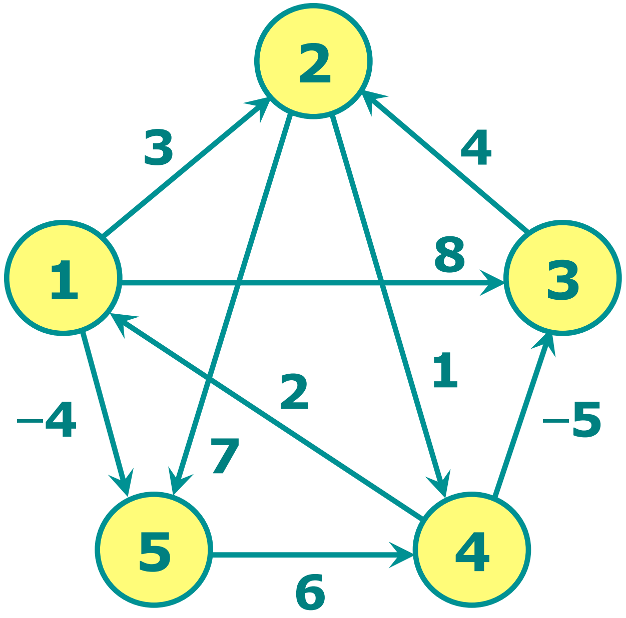
Proof. Let $p = v_1 \to v_2 \to \ldots \to v_k$ be a path in G. We have

$$\begin{aligned}
\hat{w}(p) &= \sum_{i=1}^{k} \hat{w}(v_{i-1}, v_i) \\
&= \sum_{i=1}^{k} \left(w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i) \right) \\
&= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) \\
&= w(p) + h(v_0) - h(v_k)
\end{aligned}$$

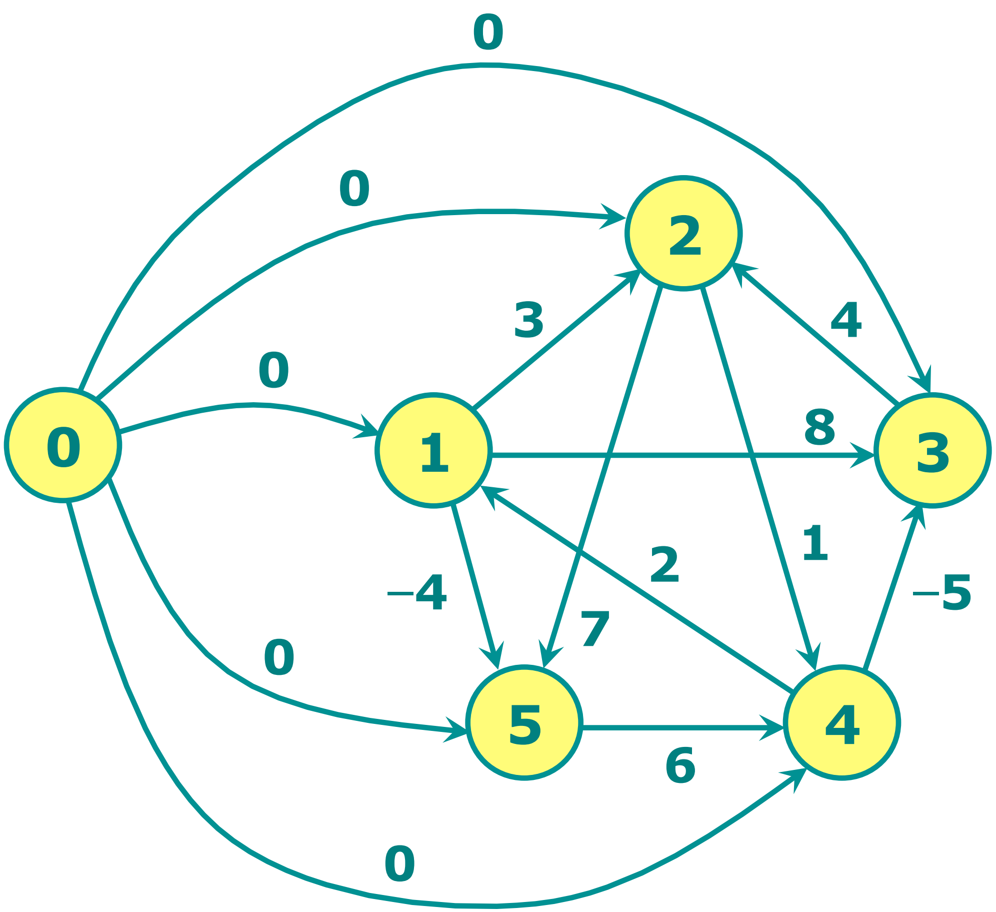
Same amount!

$\square$

Graph reweighting

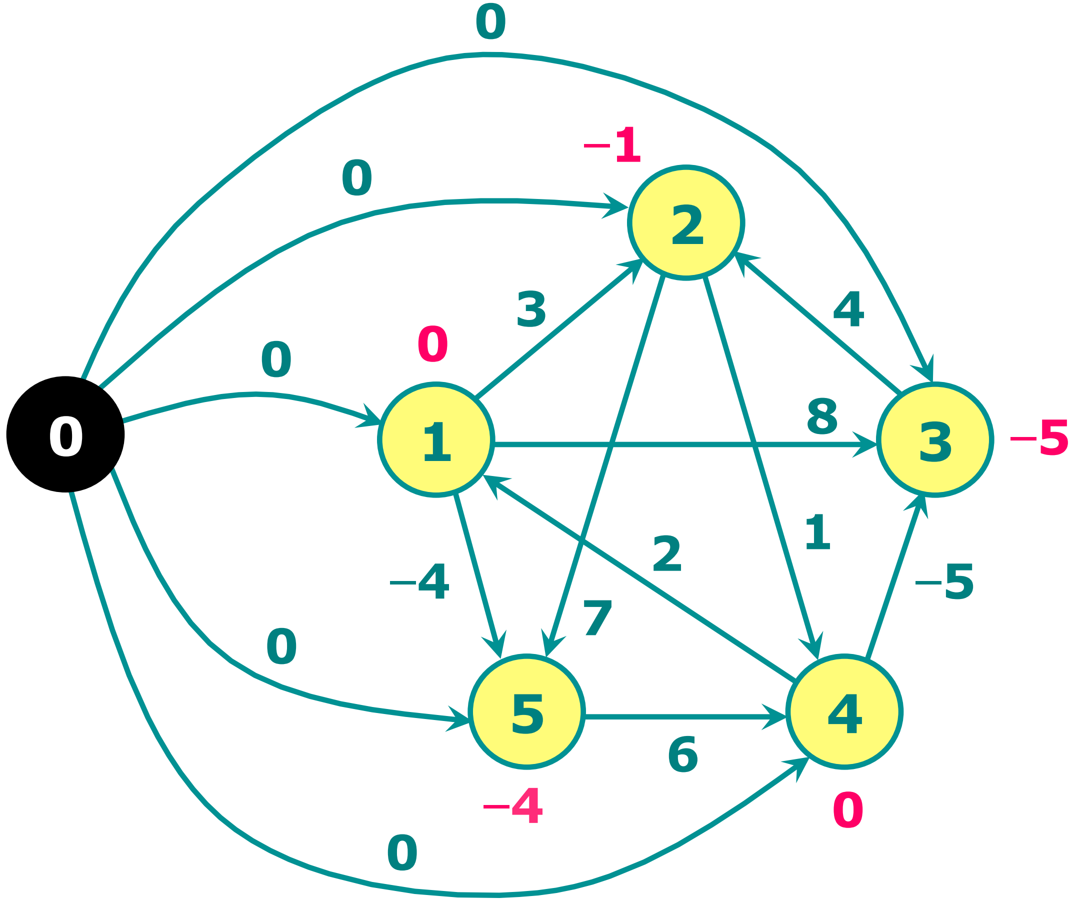


Graph reweighting



Run Bellman-Ford algorithm for vertex v_0

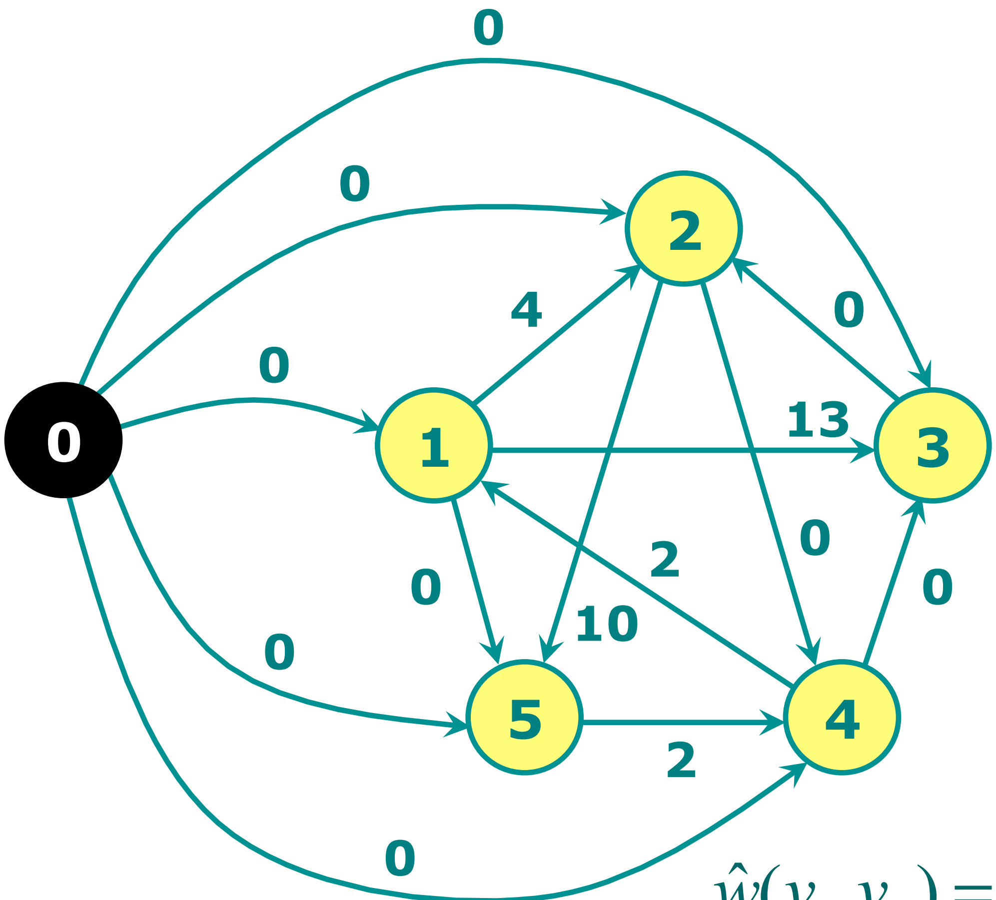
Graph reweighting



$$h(v_k) = \delta(v_0, v_k)$$

Run Bellman-Ford algorithm for vertex v_0

Graph reweighting



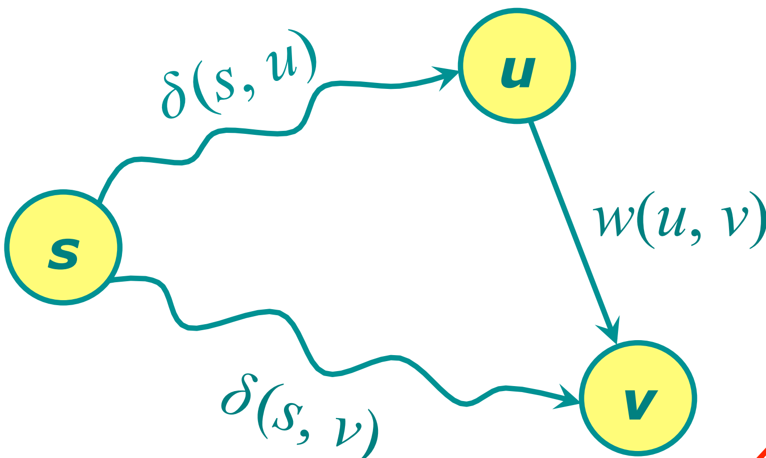
$$h(v_k) = \delta(v_0, v_k)$$

$$\hat{w}(v_i, v_j) = w(v_i, v_j) + h(v_i) - h(v_j)$$

Run Dijkstra's algorithm for each v_k

Graph reweighting

Why is $\hat{w}(u,v)$ nonnegative?



Triangle inequality

$$\delta(s,v) \le \delta(s,u) + w(u,v)$$
$$w(u,v) + \delta(s,u) - \delta(s,v) \ge 0$$
$$h(v) = \delta(s,v),\ h(u) = \delta(s,u)$$

$$\Rightarrow$$

$$\hat{w}(u,v) = w(u,v) + h(u) - h(v) \ge 0$$

Johnson's algorithm

JOHNSON(G)

1. Compute G', where $V[G'] = V[G] \cup \{s\}$,
   $E[G'] = E[G] \cup \{(s, v): v \in V[G]\}$, and
   $w(s, v) = 0$ for all $v \in V[G]$
2. **if** BELLMAN-FORD(G', w, s) = FALSE
3. **then** print "the input graph contains a negative-weight cycle"
4. **else for** each vertex $v \in V[G']$
5. **do** set $h(v)$ to the value of $\delta(s, v)$
6. **for** each edge $(u, v) \in E[G']$
7. **do** $\hat{w}(u,v) \leftarrow w(u,v) + h(u) - h(v)$
8. **for** each vertex $u \in V[G]$
9. **do** run DIJKSTRA$(G, \hat{w}, u)$ to compute $\hat{\delta}(u,v)$
   for all $v \in V[G]$

Johnson's algorithm

for each vertex $v \in V[G]$

    do $d_{uv} \leftarrow \hat{\delta}(u,v) + h(v) - h(u)$

return D

Analysis of Johnson's algorithm

1. Run 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 for each vertex $u \in V[G]$.
- Time $= O(VE\lg V)$.

3. For each $(u, v) \in E[G]$, compute $\delta(u, v)$.
- Time $= O(E)$.

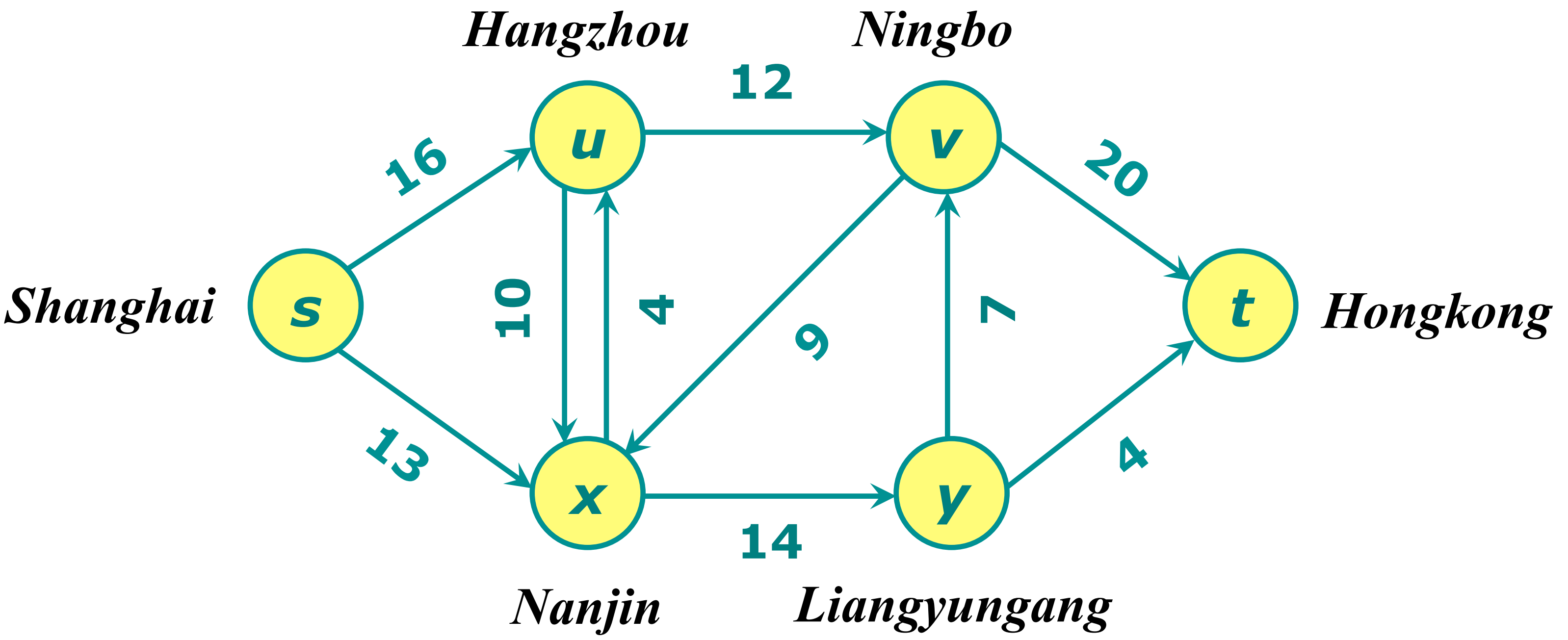
Total time $= O(VE\lg V)$.

Johnson's algorithm is particularly suitable for sparse graph.

All-pairs shortest-paths algorithms

Algorithms	**Time**	**Data structure**
Brute-force (run Bellman-Ford once from each vertex)	$O(V^2E)$	Adjacency-list or adjacency-matrix
Dynamic programming	$O(V^4)$	Adjacency-matrix only
Improved dynamic Programming	$O(V^3 lgV)$	Adjacency-matrix only
Floyd-Warshall algorithm	$O(V^3)$	Adjacency-matrix only
Johnson algorithm	$O(VElgV)$	Adjacency-list or adjacency-matrix

Flow networks



Flow networks

Definition.
A ***flow network*** is a directed graph $G = (V, E)$ with two distinguished vertices: a ***source*** s and a ***sink*** t. Each edge $(u, v) \in E$ has a nonnegative ***capacity*** $c(u, v)$. If $(u, v) \notin E$, then $c(u, v) = 0$.

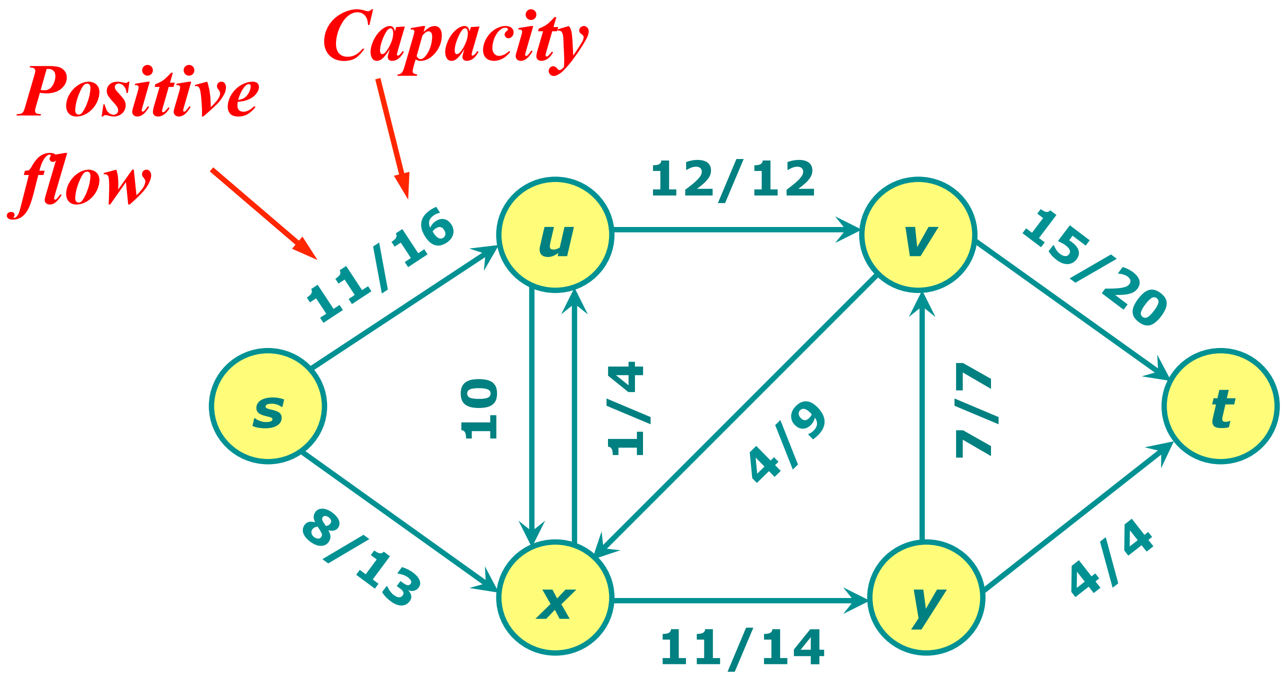
Flow networks

Definition.

A ***positive flow*** on G is a function $f\colon V \times V \to \mathbb{R}$ satisfying the following:

- ***Capacity constraint***: For all $u, v \in V$, we require $0 \le f(u, v) \le c(u, v)$.
- ***Skew symmetry***: For all $u, v \in V$, we require $f(u, v) = -f(v, u)$
- ***Flow conservation***: For all $u \in V - \{s, t\}$, we require $\sum_{v \in V} f(u,v) = 0$

A flow on a network



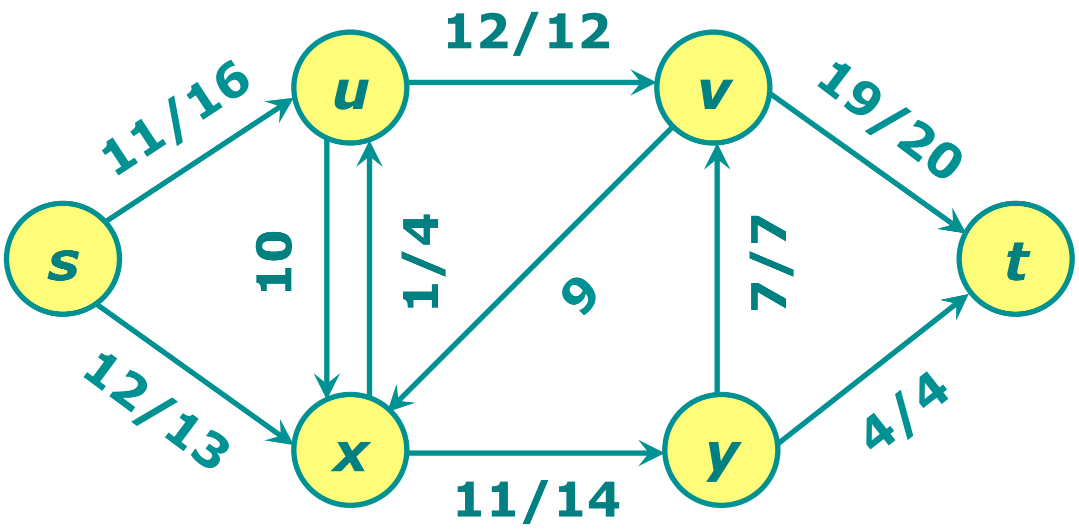
***Flow conservation*:**

- Flow into x is $8 + 4 = 12$.
- Flow out of x is $1 + 11 = 12$.

The ***value*** of this flow is $11 + 8 = 19$.

Maximum-flow problem

Maximum-flow problem.
Given a flow network G, find a flow of maximum value on G.



The ***maximum flow*** is 23.

Cuts

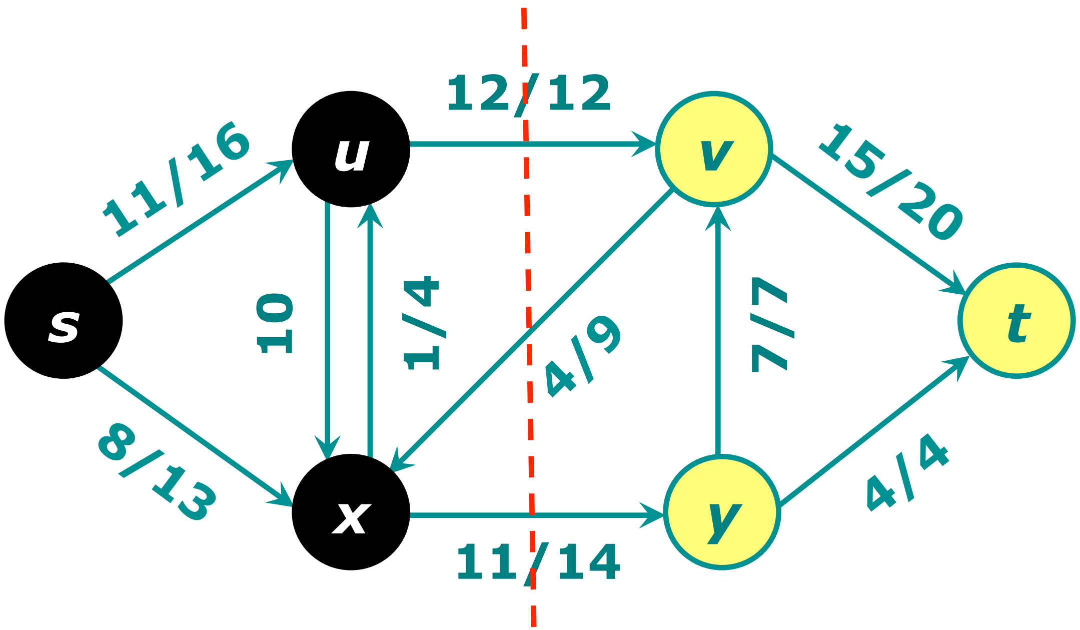
Definition.

A ***cut*** (S, T) of a flow network $G = (V, E)$ is a partition of V such that $s \in S$ and $t \in T$. If f is a flow on G, then the ***flow across the cut*** is $f(S, T)$.

Maximum flow in a network is bounded by the capacity of ***minimum cut*** of the network.

Why?

Cuts of flow networks



The net flow across this cut is

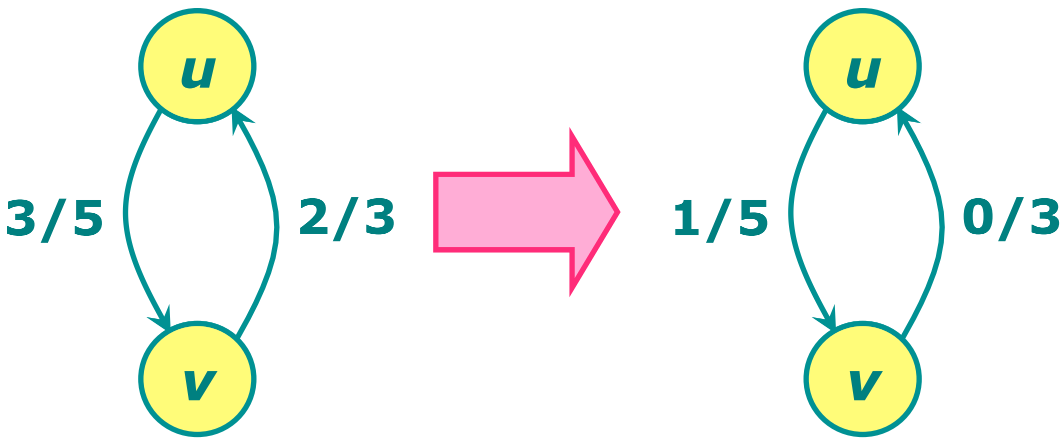
$f(u, v) + f(x, v) + f(x, y) = 12 + (-4) + 11 = 19.$

and its capacity is

$c(u, v) + c(x, y) = 12 + 14 = 26.$

Flow cancellation

Without loss of generality, positive flow goes either from u to v, or from v to u, but not both.



Net flow from u to v in both cases is 1.

The capacity constraint and flow conservation are preserved by this transformation.

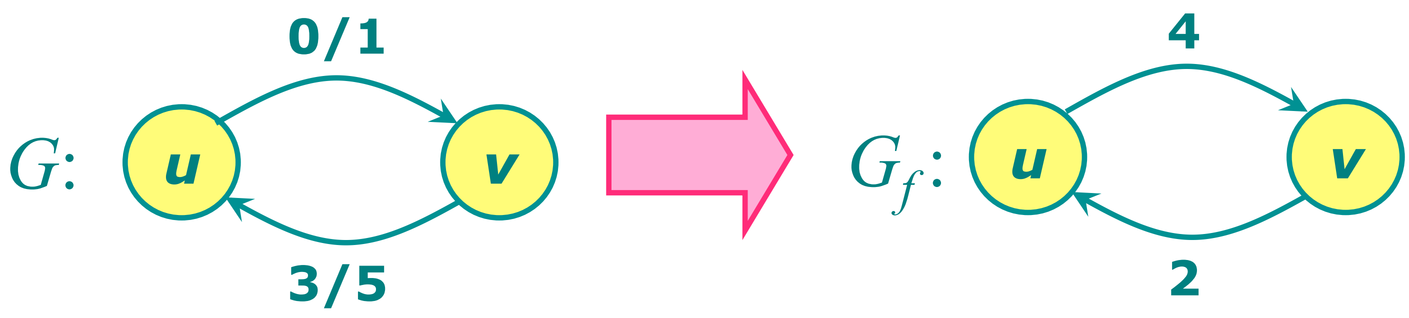
Residual network

Definition.

Let f be a flow on $G = (V, E)$. The ***residual network*** $G_f(V, E_f)$ is the graph with strictly positive ***residual capacities***

$$c_f(u, v) = c(u, v) - f(u, v) > 0.$$

Edges in E_f admit more flow.



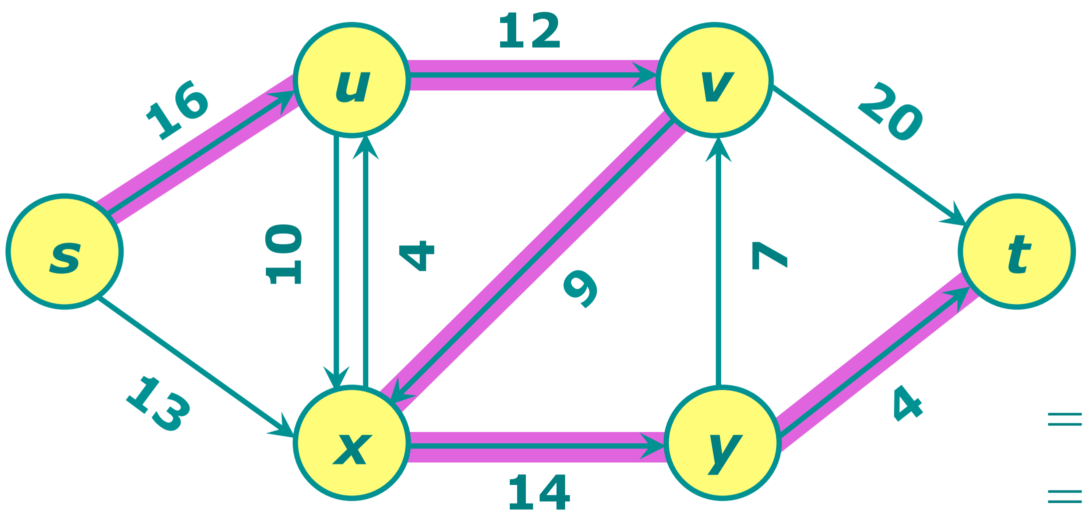
Augmenting paths

Definition.

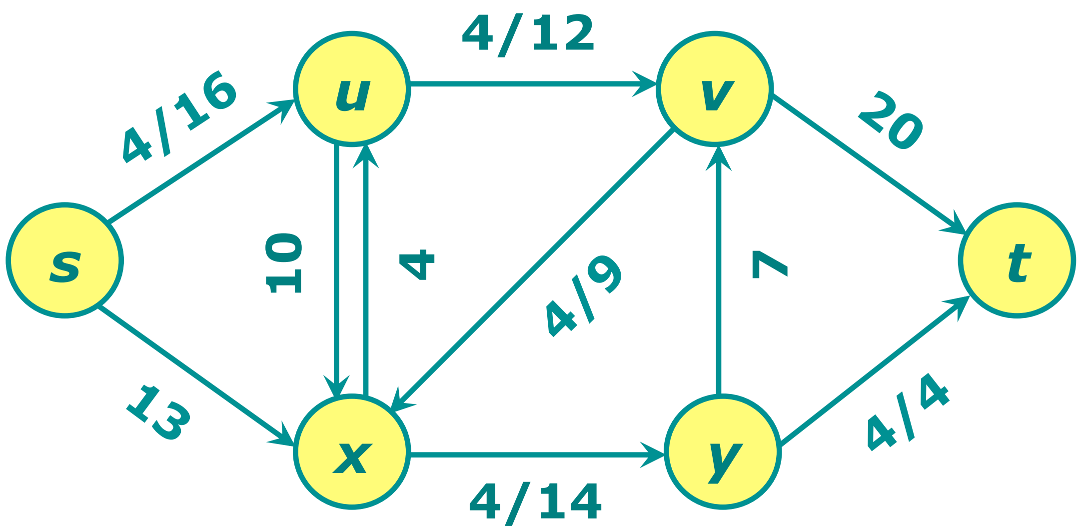
Any path from s to t in G_f is an augmenting path in G with respect to f. The flow value can be increased along an ***augmenting path*** p by

$$c_f(p) = \min_{(u,v \in p)} \{c_f(u,v)\}$$

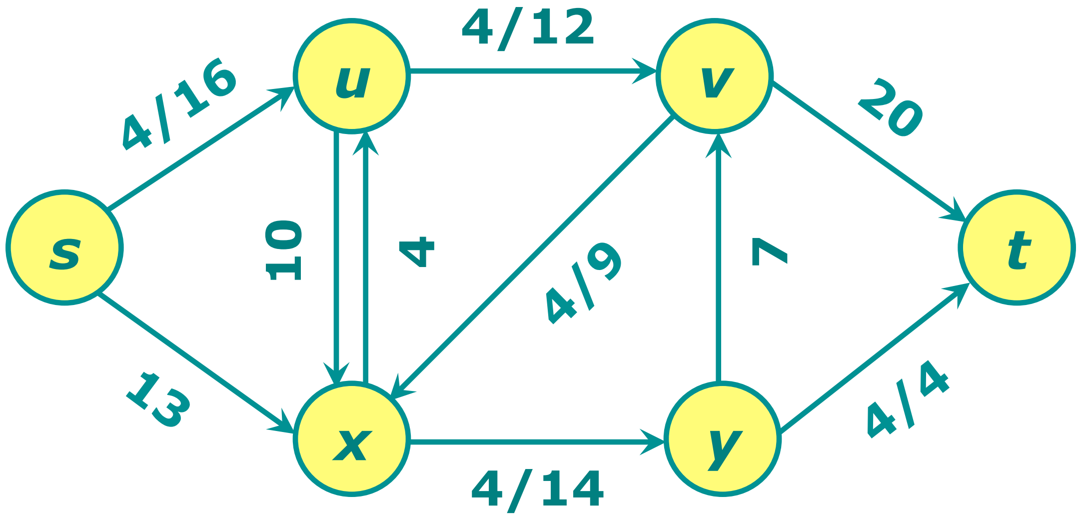
Example of maximum-flow problem

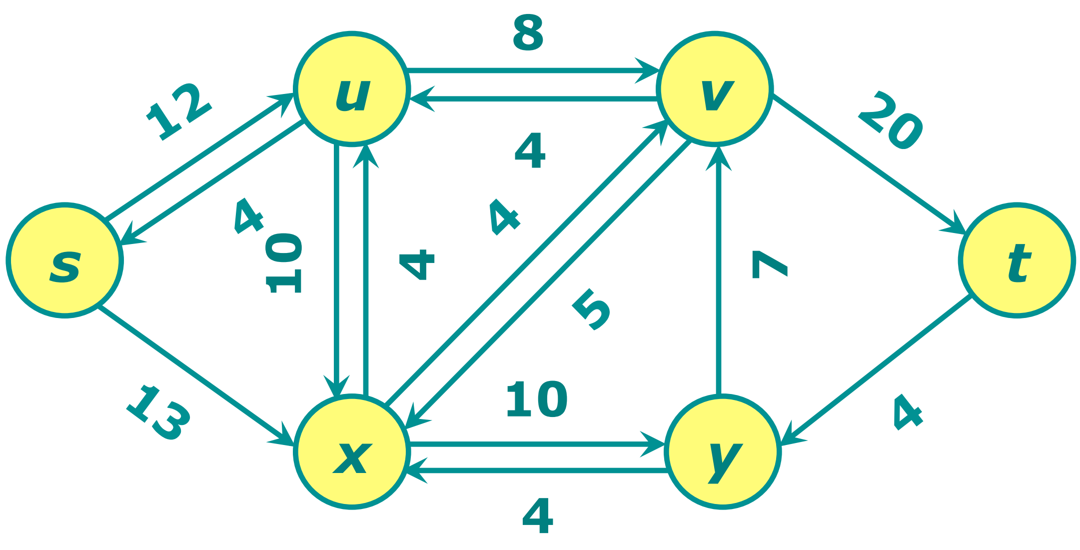


$c_f(p)$
$= \min(16, 12, 9, 14, 4)$
$= 4$

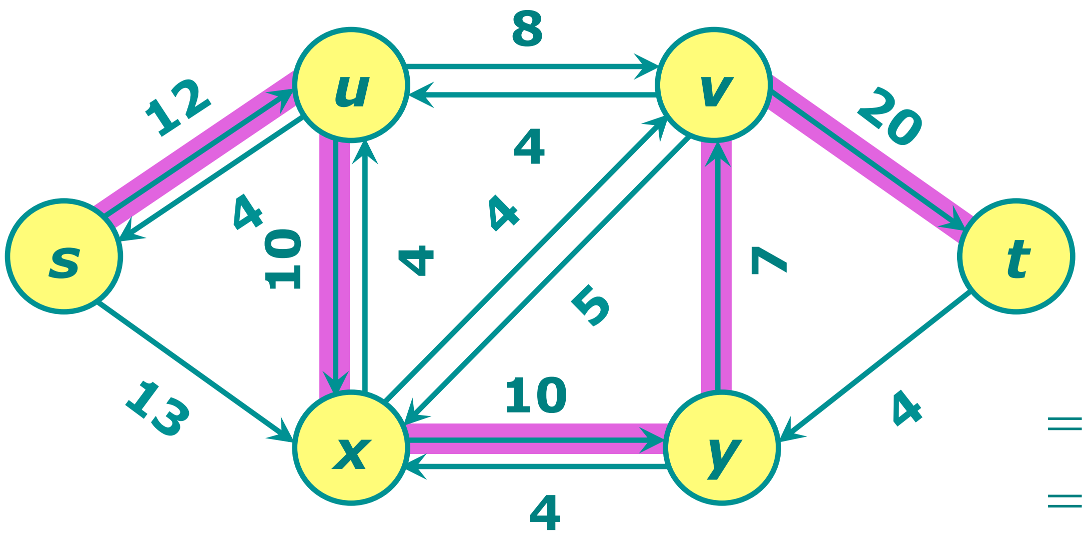


Example of maximum-flow problem



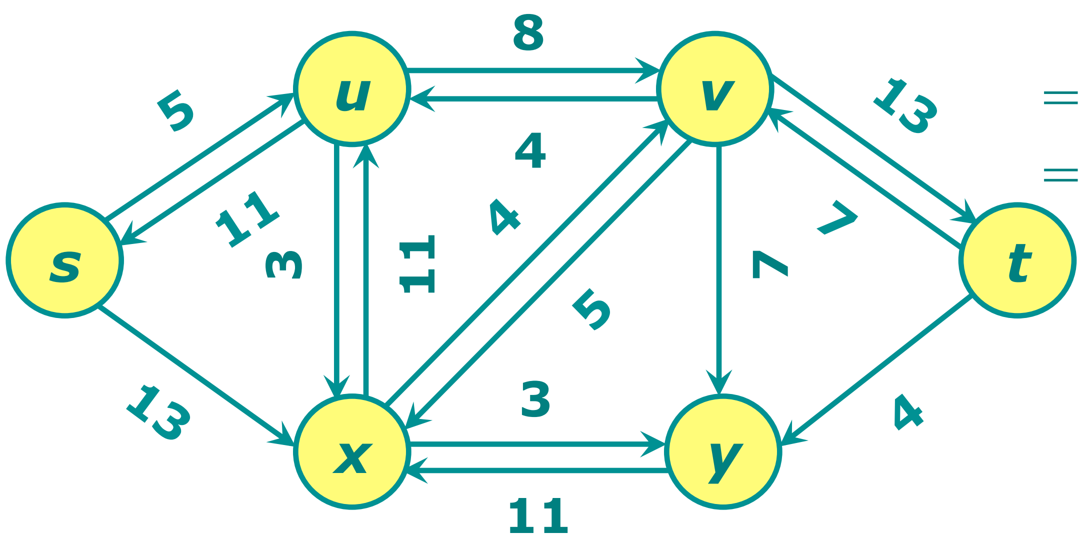


Example of maximum-flow problem

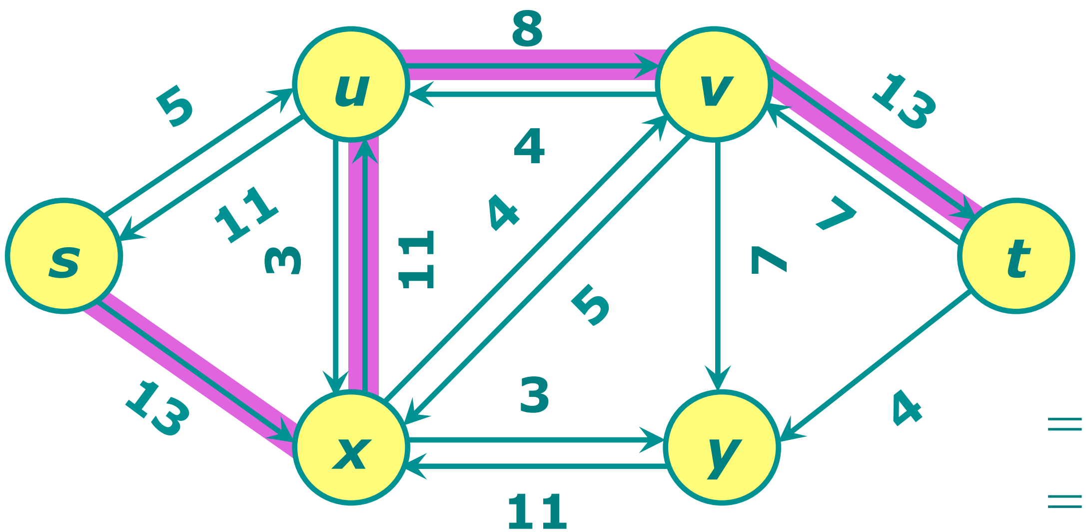


$c_f(p)$
$= \min(12, 10, 10, 7, 20)$
$= 7$

Total
$= 4 + 7$
$= 11$

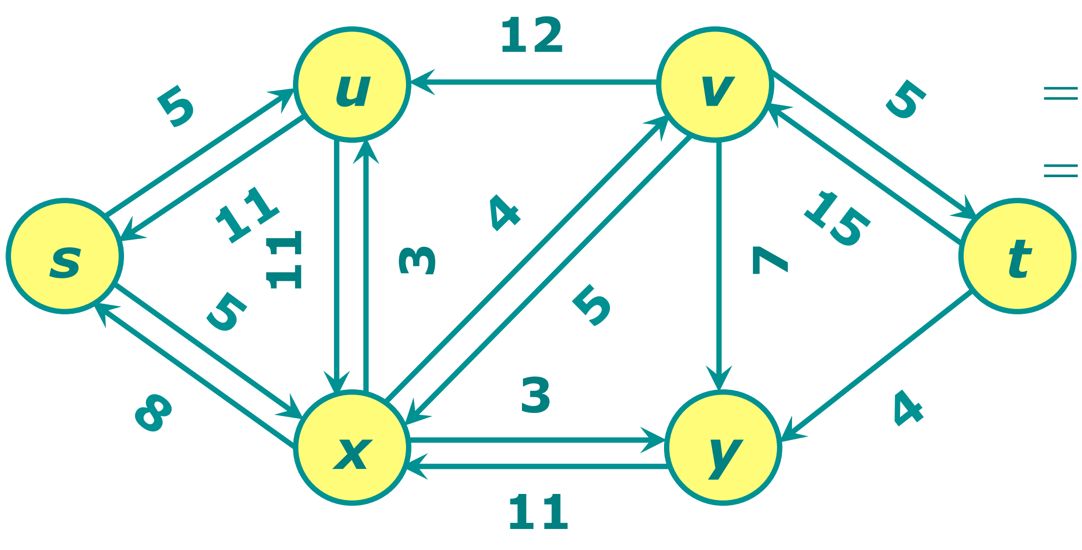


Example of maximum-flow problem

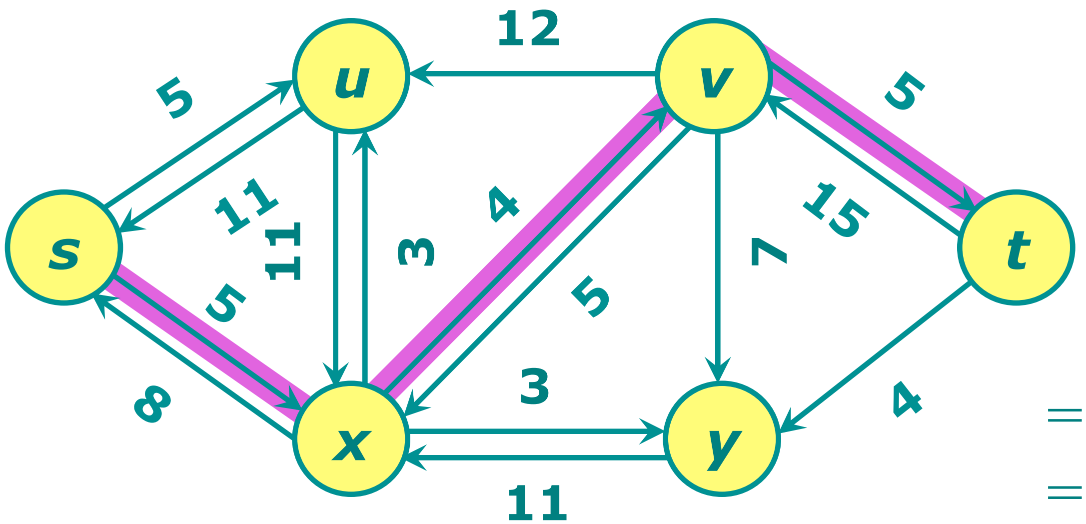


$c_f(p)$
$= \min(13, 11, 8, 13)$
$= 8$

Total
$= 4 + 7 + 8$
$= 19$

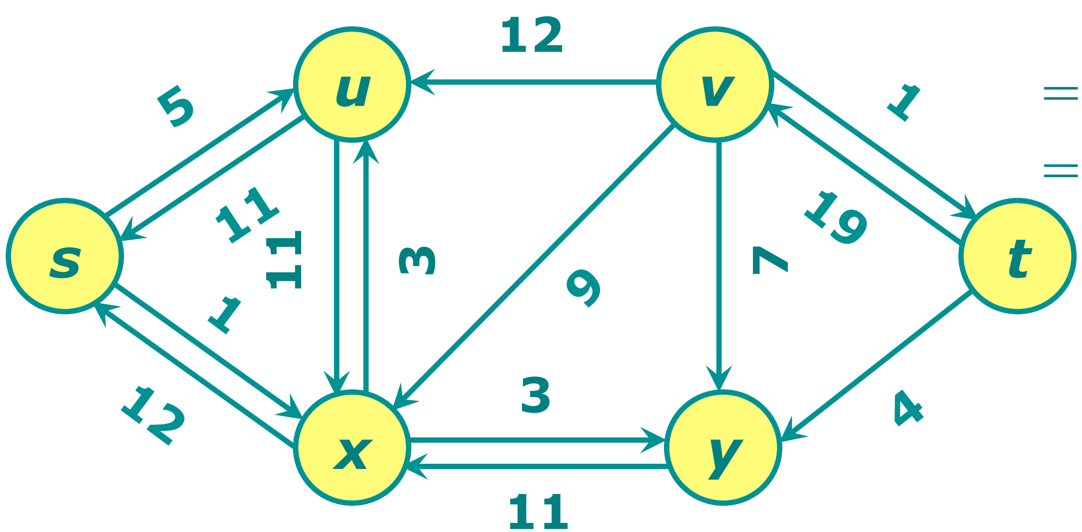


Example of maximum-flow problem

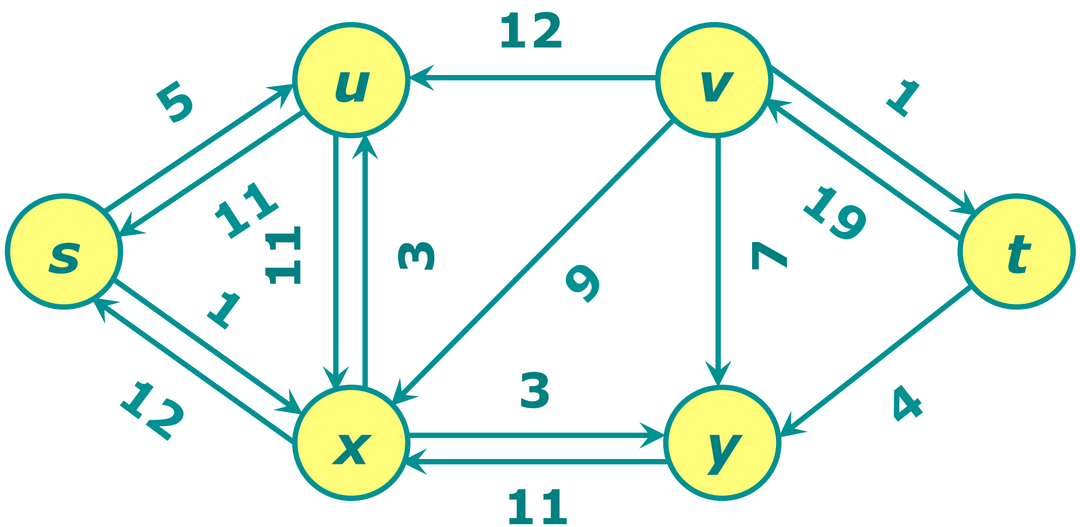


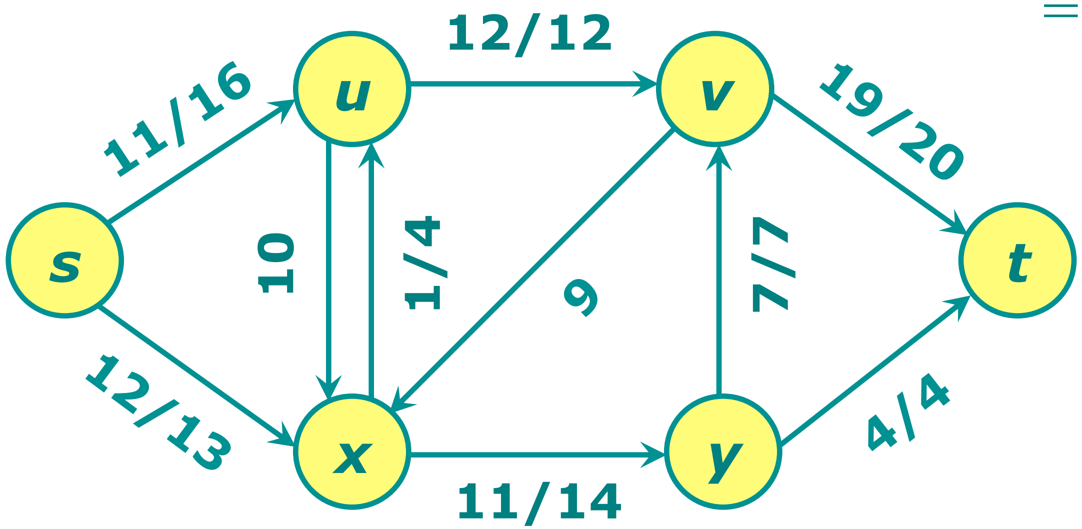
$c_f(p)$
$= \min(5, 4, 5)$
$= 4$

Total
$= 4 + 7 + 8 + 4$
$= 23$



Example of maximum-flow problem





The maximum flow = 23

Ford-Fulkerson algorithm

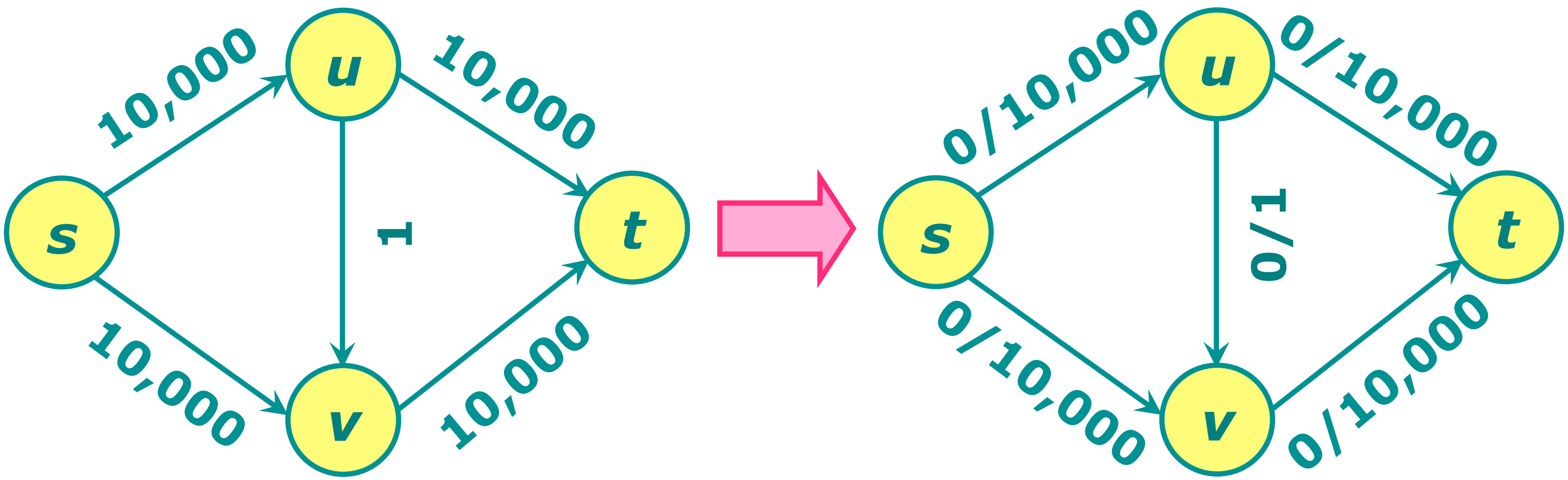
FORD-FULKERSON(G, s, t)

1. **for** each edge $(u, v) \in E[G]$
2. **do** $f[u, v] \leftarrow 0$
3. $f[v, u] \leftarrow 0$
4. **while** there exists a path p from s to t in the residual network G_f
5. **do** $c_f(p) \leftarrow \min\{c_f(u, v) : (u, v) \text{ is in } p\}$
6. **for** each edge (u, v) in p
7. **do** $f[u, v] \leftarrow f[u, v] + c_f(p)$
8. $f[v, u] \leftarrow -f[v, u]$

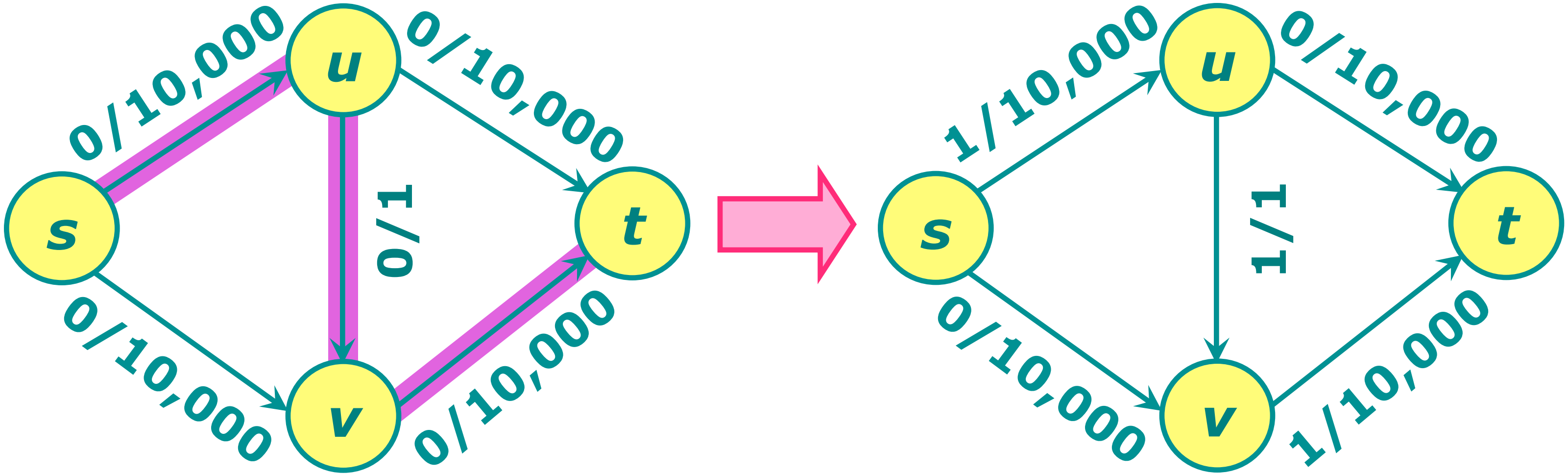
Analysis of Ford-Fulkerson algorithm

- Find a path in a residual netork is $O(V+E)$ if we use either depth-first search or breadth-first search.
- f^* denote the maximum flow found by the algorithm.
- Running time of Ford-Fulkerson algorithm is $O(E|f^*|)$.

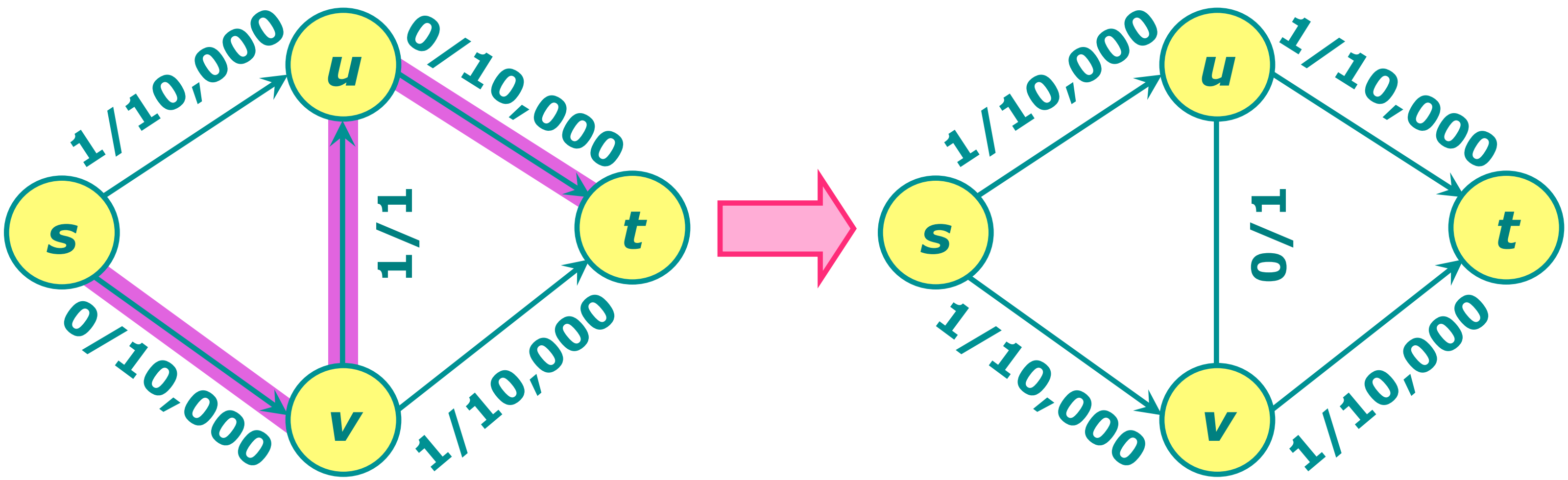
Ford-Fulkerson max-flow algorithm



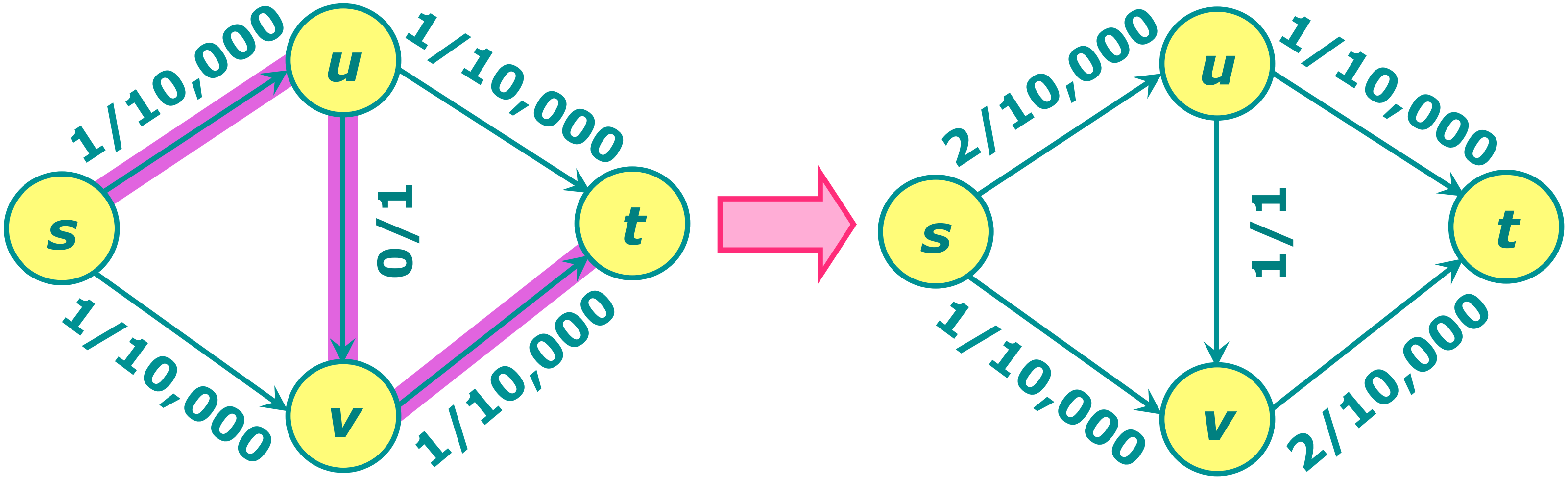
Ford-Fulkerson max-flow algorithm



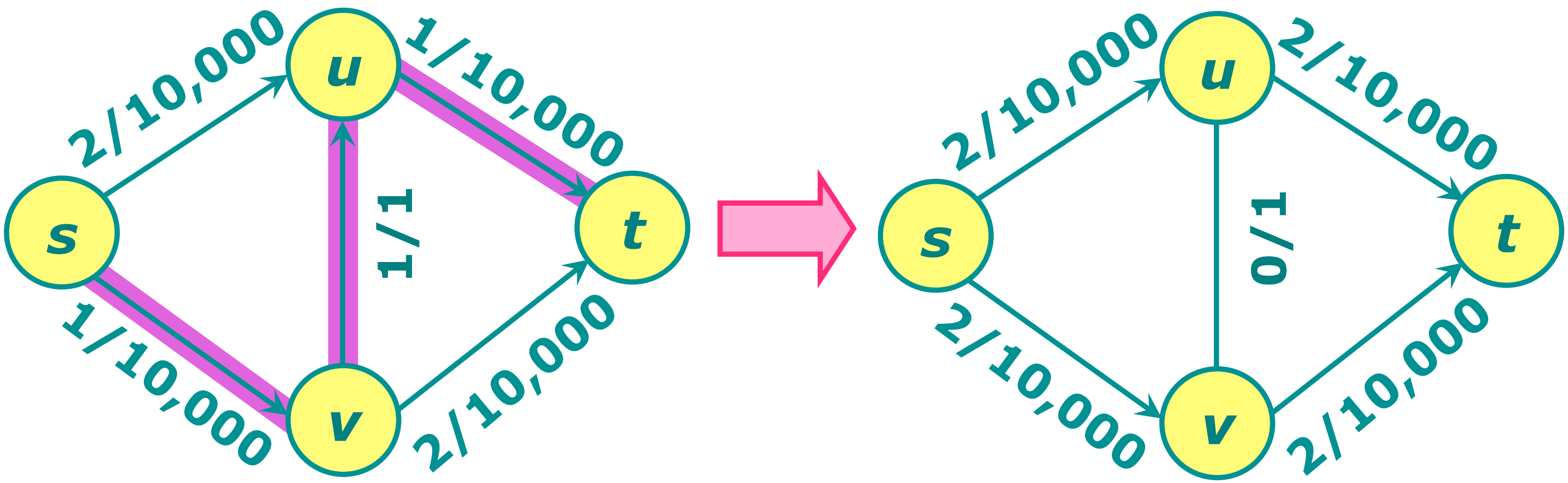
Ford-Fulkerson max-flow algorithm



Ford-Fulkerson max-flow algorithm



Ford-Fulkerson max-flow algorithm



Running time is 10,000.

Edmonds-Karp algorithm

Edmonds and Karp noticed that many people's implementations of Ford-Fulkerson augment along a ***breadth-first augmenting path***: a shortest path in G_f from s to t where each edge has weight 1. These implementations would always run relatively fast.

Since a breadth-first augmenting path can be found in $O(V + E)$ time, their analysis, which provided the first polynomial-time bound on maximum flow, focuses on bounding the number of flow augmentations.

Edmonds-Karp algorithm's running time is $O(VE^2)$.

Single-source shortest-paths algorithms

Algorithms	**Unweighted**	**Positive**	**Negative**	**Cycle**	**Time**
Breadth-first search					
Dag Shortest Paths					
Dijkstra					
Bellman-Ford					

Single-source shortest-paths algorithms

Algorithms	**Unweighted**	**Positive**	**Negative**	**Cycle**	**Time**
Breadth-first search	○	×	×	○	$O(V+E)$
Dag Shortest Paths	○	○	○	×	$O(V+E)$
Dijkstra	○	○	×	○	$O(E \lg V)$
Bellman-Ford	○	○	○	○	$O(VE)$

All-pairs shortest-paths algorithms

Algorithms	**Time**	**Data structure**
Brute-force (run Bellman-Ford once from each vertex)		
Dynamic programming		
Improved dynamic Programming		
Floyd-Warshall algorithm		
Johnson algorithm		

All-pairs shortest-paths algorithms

Algorithms	**Time**	**Data structure**
Brute-force (run Bellman-Ford once from each vertex)	$O(V^2E)$	Adjacency-list or adjacency-matrix
Dynamic programming	$O(V^4)$	Adjacency-matrix only
Improved dynamic Programming	$O(V^3 lg V)$	Adjacency-matrix only
Floyd-Warshall algorithm	$O(V^3)$	Adjacency-matrix only
Johnson algorithm	$O(VE lg V)$	Adjacency-list or adjacency-matrix

Any question?

Xiaoqing Zheng
Fudan University