Part IV

Graph Algorithms

§ Graph Algorithms:

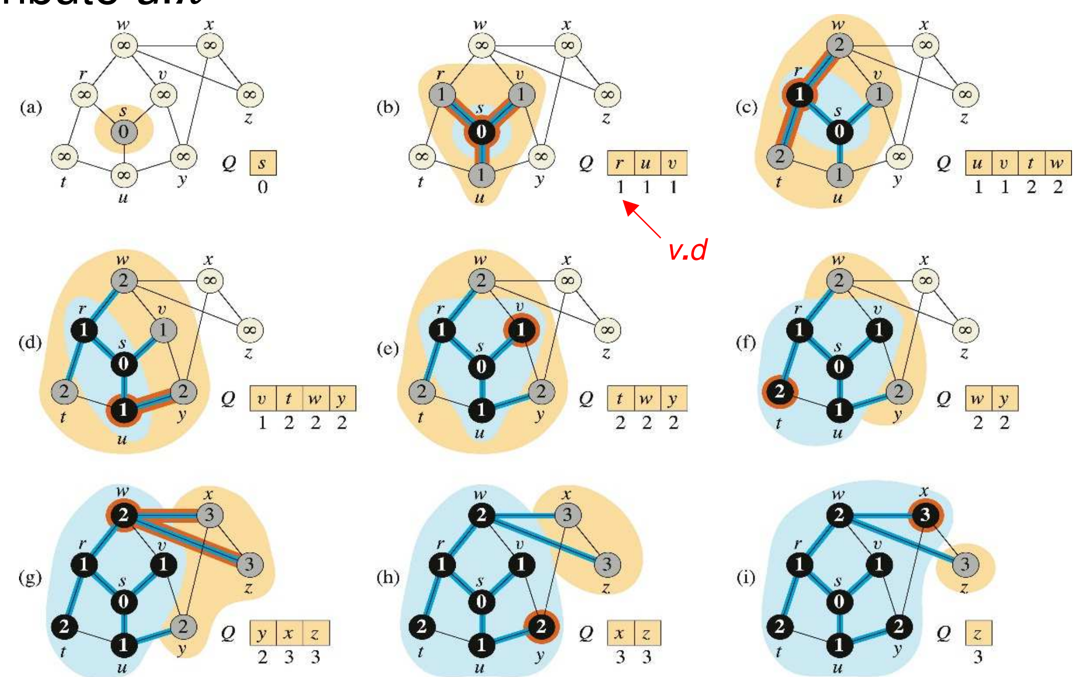
- Elementary Graph Algorithms
- Single-Source Shortest Paths
- All-Pairs Shortest Paths
- Maximum Flow
- Minimum Spanning Trees

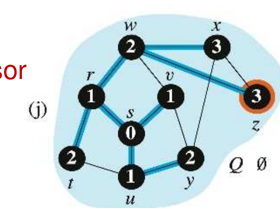
- **Elementary Graph Algorithms**
  - breadth-first search (BFS) over both undirected and directed graphs
  - depth-first search (DFS) over both undirected and directed graphs
  - topological sort over directed graphs
- **Single-Source Shortest Paths**
  - Bellman-Ford algorithm (for general directed graphs, even with negative weights & cycles)
  - Dijkstra's algorithm (for directed graphs without negative weights)
- **All-Pair Shortest Paths**
  - Floyd-Warshall algorithm (for directed graphs, without negative-weight cycles)
- **Maximum Flow**
  - Basic Ford-Fulkerson algorithm (for directed graphs, without negative edge capacity)
  - Edmonds-Karp algorithm (following BFS to find an augmentation path iteratively)

Elementary Graph Algorithms (continued)

- **Breadth-First Search (BFS)**
  - explore edges of graph $G = (V, E)$ to discover every vertex from a source vertex, s
  - color each vertex in <u>white</u>, <u>gray</u>, or <u>black</u> **gray: discovered but its neighbors not fully explored yet; gray vertexes kept in a queue** **black: all its neighbors fully explored**
  - if $(u, v) \in E$ and vertex u is black, vertex v is either gray or black
  - <u>predecessor</u> of vertex u kept in attribute $u.\pi$

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0                          // distance to s
s.π = NIL
Q = ∅
ENQUEUE(Q, s)                    // elements in Q are in gray
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE      // explore only those
                                 // not discovered yet
            v.color = GRAY
            v.d = u.d + 1
            v.π = u              // u being predecessor
            ENQUEUE(Q, v)        // of v, which is then
                                 // enqueued
    u.color = BLACK              // finish exploring all u's
                                 // neighbors
```





Total time for graphs: $O(V + E)$

Elementary Graph Algorithms (continued)

- **Breadth-First Search (BFS) Correctness**

Theorem 1.

Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from the source s, and upon termination, $v.d = \delta(s, v)$ for all $v \in V$. Moreover, for any vertex $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $v.\pi$ followed by the edge $(v.\pi, v)$.

≡ shortest path from s to $v.\pi$ + edge $(v.\pi, v)$

Note. $\delta(u, v)$ denotes shortest path distance from u to v.

Predecessor subgraph of G, $G_\pi = (V_\pi, E_\pi)$, with

$$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$$

and

$$E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\} .$$

G_π is the breadth-first tree if V_π contains all vertices reachable from s.

Elementary Graph Algorithms (continued)

- **Depth-First Search (DFS)**

From each search step (at a gray node) to meet:
+ a white neighbor → uncovered one, a "tree" edge added
+ a gray neighbor (with just *v.d* given) → a "back" edge
+ a black neighbor (with both *v.d* & *v.f* given) → "F" or "C" edge; from its *v.d* & my *v.d* to determine F/C

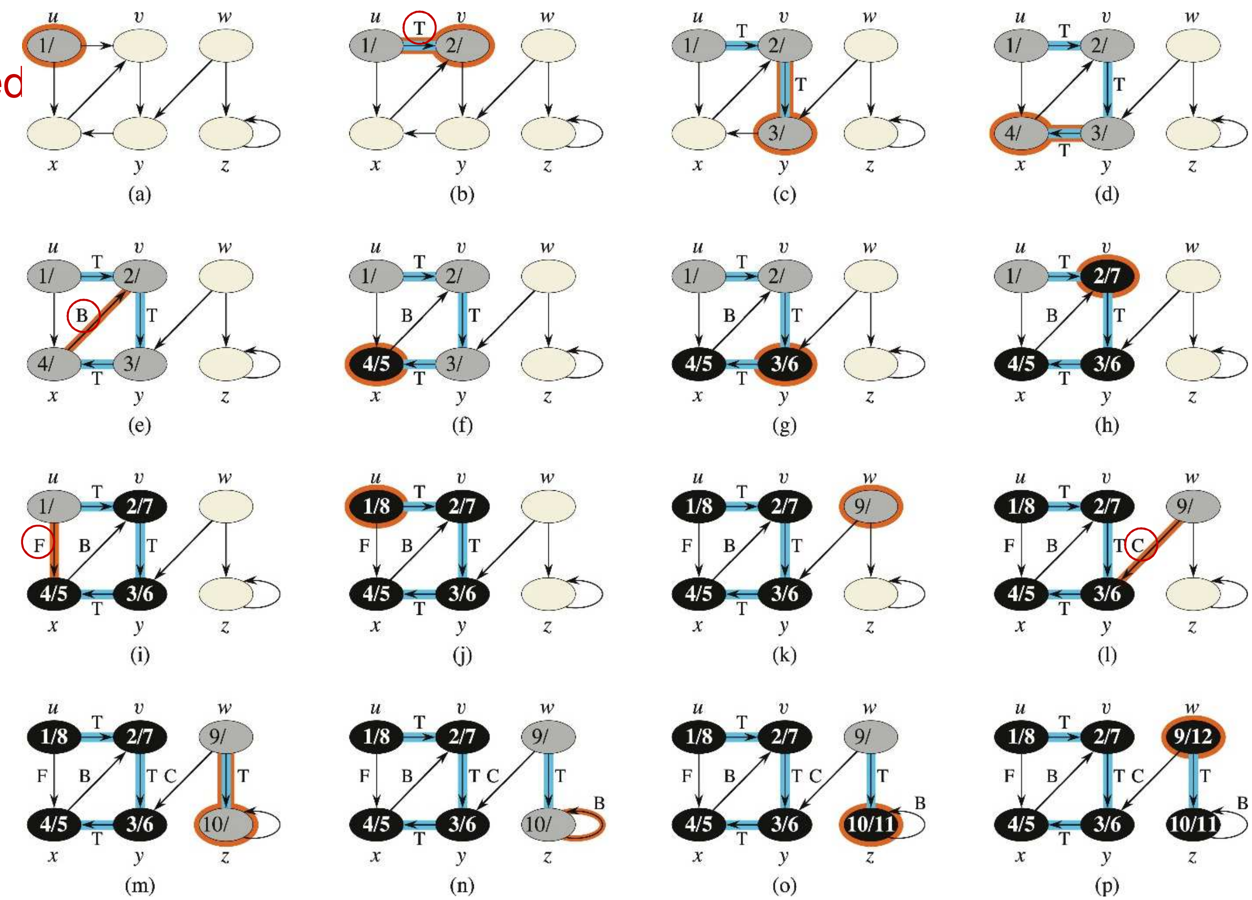
  – color each vertex in graph $G = (V, E)$ in white, gray, or black: initial in white, then in gray upon discovery, and finally in black once done (i.e., all its adjacencies examined)

  – two timestamps in each vertex *v*: *v.d* for discovery time and *v.f* for finish time, with $v.f \leq 2|V|$

Total time for directed graphs: $O(2V + E)$

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1
u.d = time                  // discovery time
u.color = GRAY
for each v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time                  // finish time
```



Elementary Graph Algorithms (continued)

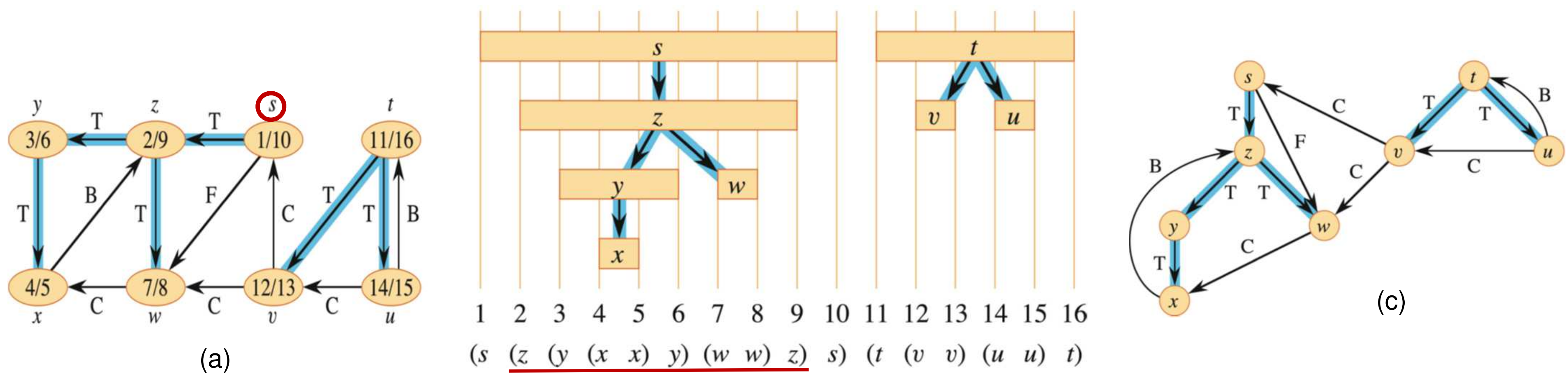
- **Properties of Depth-First Search (DFS)**
  - predecessor subgraph of G, $G_\pi = (V_\pi, E_\pi)$, forms a forest of trees (with one tree for a component)
  - discovery and finish times have parenthesis structure

like a stack: node discovered **first** finishes its exploration **last**

Theorem 2.

In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

- the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,
- the interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$, and u is a descendant of v in a depth-first tree, or
- the interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$, and v is a descendant of u in a depth-first tree.



(a) (b) (c)

consider DFS for the above with edges all undirected

Elementary Graph Algorithms (continued)

- **Edge Classification under Depth-First Search (DFS)**
  - four edge types in the forest of trees, $G_\pi = (V_\pi, E_\pi)$, for a directed graph G:

1. ***Tree edges*** are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v). This means that v is white upon discovery.
2. ***Back edges*** are those non-tree edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.
3. ***Forward edges*** are those nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
4. ***Cross edges*** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

For an undirected graph, only the first two edge types exist in the tree created by DFS, as follows:

Theorem 3.

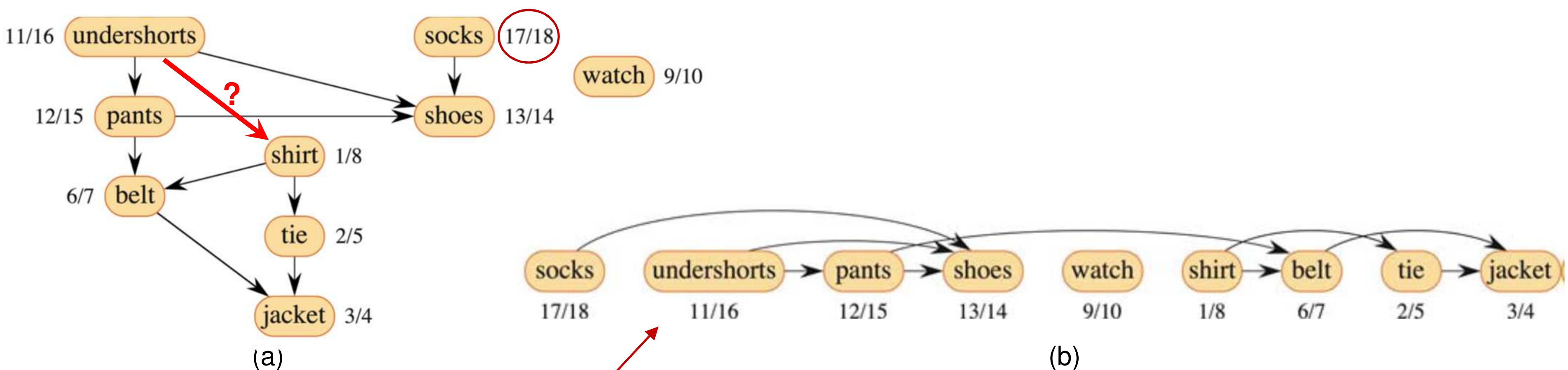
Under depth-first search in an undirected graph G, every edge of G is either a tree edge or a back edge. (See the example given in last slide for explanation.)

Elementary Graph Algorithms (continued)

- **Topological Sort under Depth-First Search (DFS) over Direct Graphs**
  - a direct acyclic graph (dag) can indicate precedence among events
  - topologically sorted vertices of a dag obtained by DFS

because under DFS, a node discovered **first** completes its discovery **last**

Example:



Nodes line up in the reverse order of their finish times after DFS so that all arrows pointing rightward.

Theorem 4.

Depth-first search produces a topological sort of the vertices of a directed graph *G*.

Single-Source Shortest Paths

- **Definition**
  - a weighted, directed graph $G = (V, E)$, with weight $w(p)$ of path P being sum of its edge weights
  - shortest-path weight $\delta(u, v)$ from u to v as follows:

$$\delta(u,v) = \begin{cases} \min\left\{w(p) : u \overset{p}{\leadsto} v\right\} & \text{if there exists a path } u \leadsto v \text{ ,} \\ \infty & \text{otherwise .} \end{cases}$$

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

  – from given source vertex to each vertex $v \in V$

Optimal substructure of shortest path

Lemma

Any subpath of a shortest path is a shortest path.

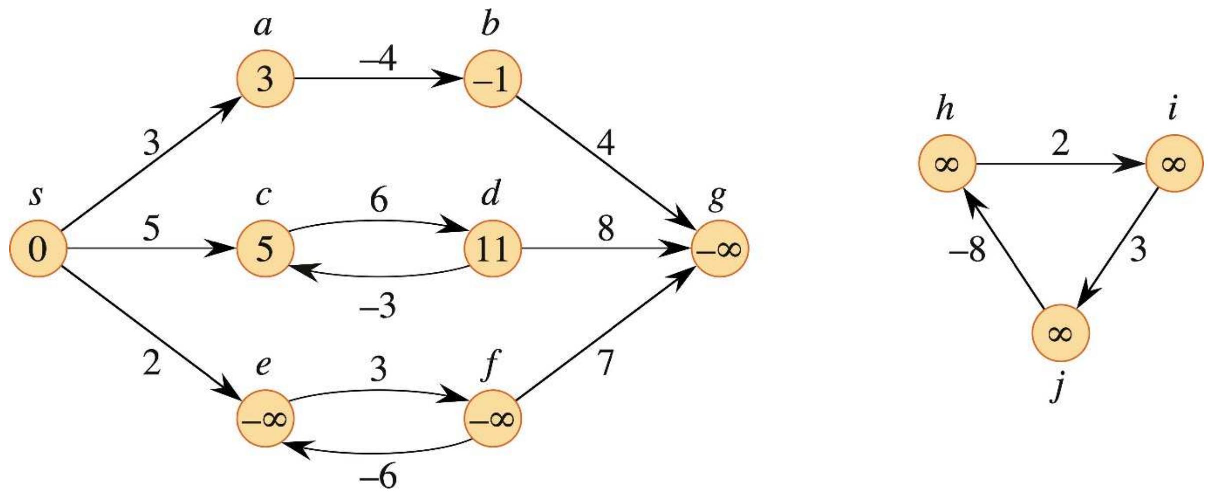
– various algorithms discussed, including
  - ○ Bellman-Ford algorithm (based on relaxation over all edges of a **general graph** iteratively) for **one** source
  - ○ Dijkstra's algorithm (a greedy method for all edge weights ≥ 0) to find shortest paths from **one** source
  - ○ Floyd-Warshall algorithm (based on dynamic programming) to find **all** shortest path pairs (All-Pair SPs)

Single-Source Shortest Paths (continued)

Negative-weight edges

– a negative-weight cycle on a path from s to $v \rightarrow \delta(s, v) = -\infty$

– negative-weight cycle formed by vertices e & f, yielding $\delta(s, e) = \delta(s, f) = \delta(s, g) = -\infty$



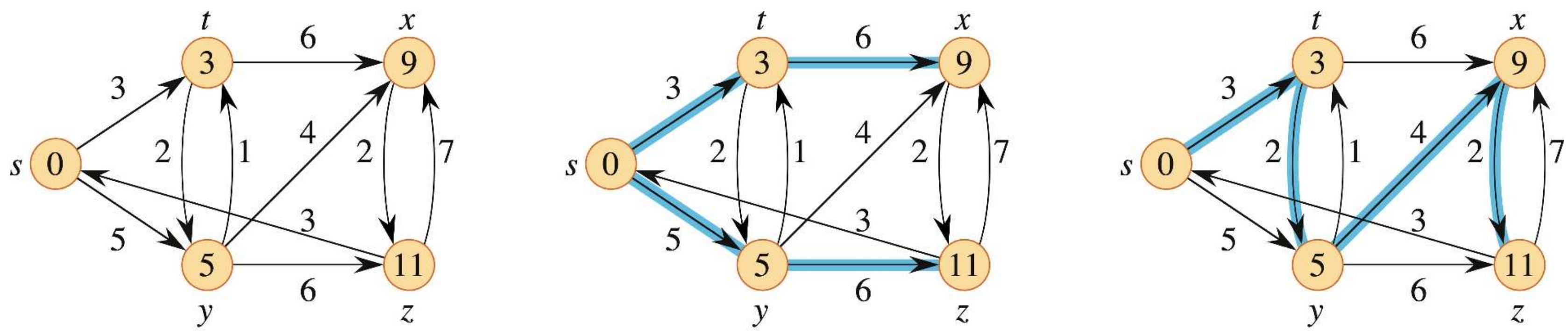
Shortest path representation

– following breadth-first search to construct *predecessor subgraph* $G_\pi = (V_\pi, E_\pi)$ for every $v \in V$

– at termination, G_π is a "shortest path tree," i.e., rooted tree from s via a shortest path to every vertex reachable from s

Single-Source Shortest Paths (continued)

- **Examples of Shortest Path Trees**



(a) Weighted, direct graph

(b) Shortest path tree rooted at *s*

(c) Another shortest path tree

<u>**Shortest path via relaxation**</u>

– relaxing edge (u, v): check if distance to v is improved by going through u (over the edge)

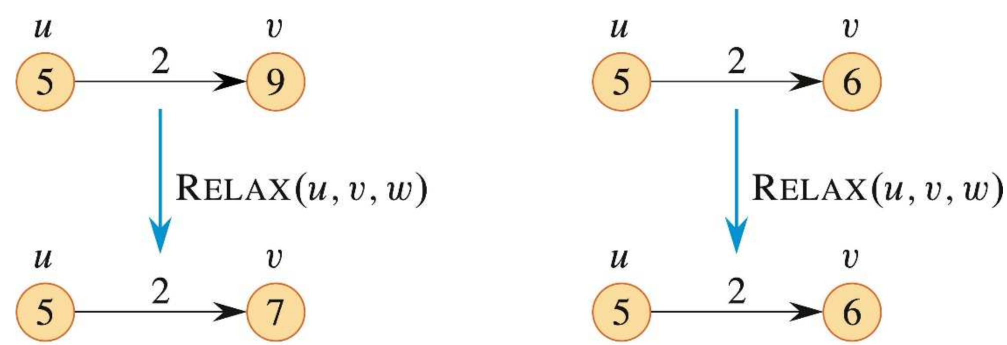
– if so, updating $v.d$ and $v.\pi$

<u>RELAX(u, v, w)</u>

if $v.d > u.d + w(u, v)$

  $v.d = u.d + w(u, v)$

  $v.\pi = u$

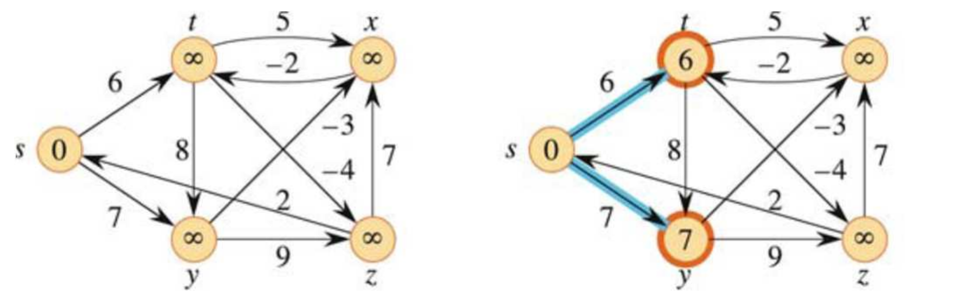


(a) (b)

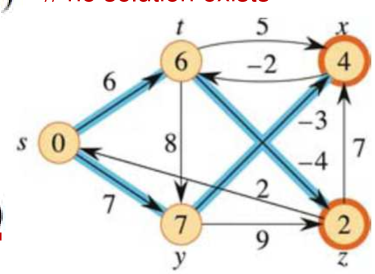
Single-Source Shortest Paths (continued)

- **Bellman-Ford Algorithm**
  - solve <u>general</u> shortest path problems (where edge weights can be negative) for <u>directed graphs</u>
  - if a negative-weighted cycle reachable from the source, no solution existing
  - otherwise, producing shortest paths to **<u>all</u>** reachable vertices and their weights from the source

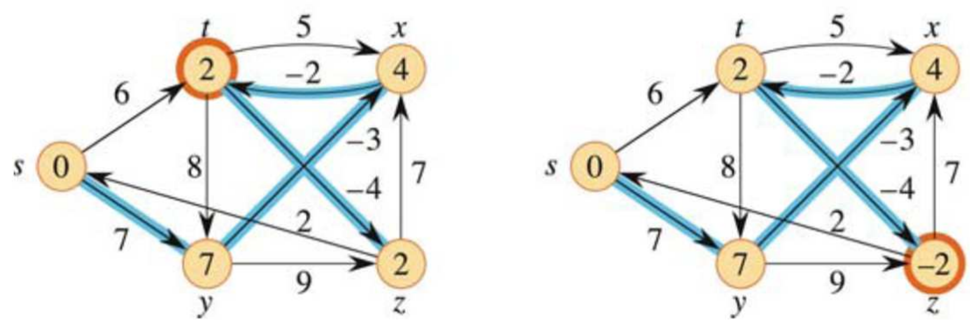
```
BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for i = 1 to |G.V| - 1          // iterate |G.V|-1 times to ensure
                                  // that relaxation effects are through
      for each edge (u, v) ∈ G.E  // relax each edge in order
          RELAX(u, v, w)
  for each edge (u, v) ∈ G.E
      if v.d > u.d + w(u, v)      // no solution exists
          return FALSE
  return TRUE
```



(a) (b)



(c)



(d) (e)

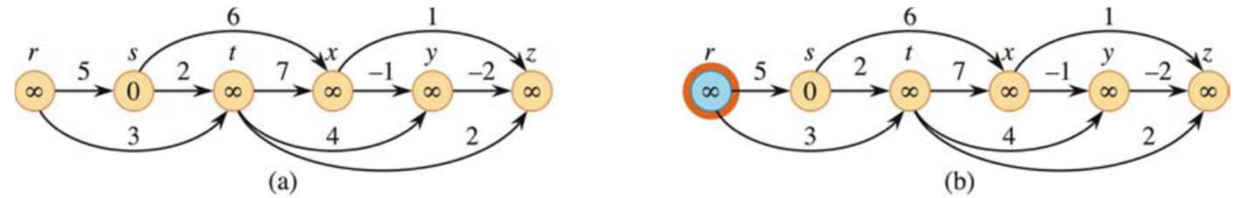
```
INIT-SINGLE-SOURCE(G, s)
  for each v ∈ G.V
      v.d = ∞
      v.π = NIL
  s.d = 0
```

+ Upon completion, the Bellman-Ford algorithm gives the predecessor subgraph G_π to denote a shortest-path tree rooted at *s*. It is applicable to graphs with **negative-weighted edges, cycles, negative-weighted cycles**.

+ The nested **for** loops <u>relax all edges $|V| - 1$ times</u>, yielding time complexity of $\Theta(VE)$.

Single-Source Shortest Paths (continued)

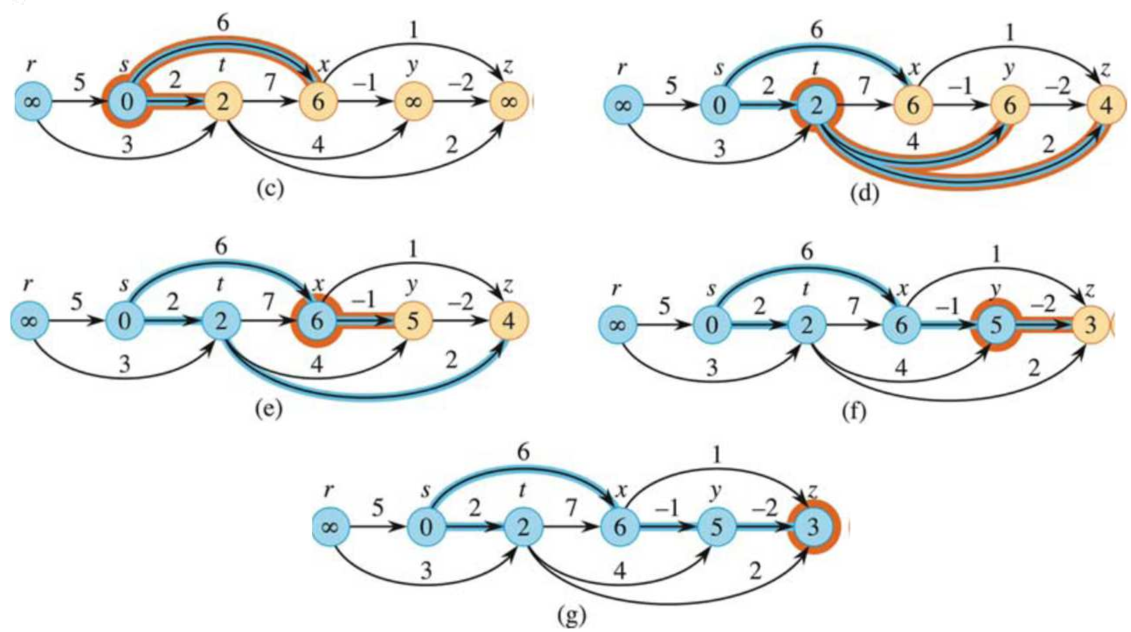
- **Single-Source Shortest Paths in Directed Acyclic Graph (DAG)**
  - without cycles in DAG, $G = (V, E)$, one can sort its vertices via topological sort (using DFS)
  - time complexity reduced to $\Theta(V + E)$, as relaxation effects propagate rightward only
  - good for graphs **without cycles** (but possibly with negative-weighted edges)



(a) (b)

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

```
topologically sort the vertices   // time: O(2V + E)
INIT-SINGLE-SOURCE(G, s)
for each vertex u, taken in topologically sorted order
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```



(c) (d) (e) (f) (g)

INIT-SINGLE-SOURCE(G, s)

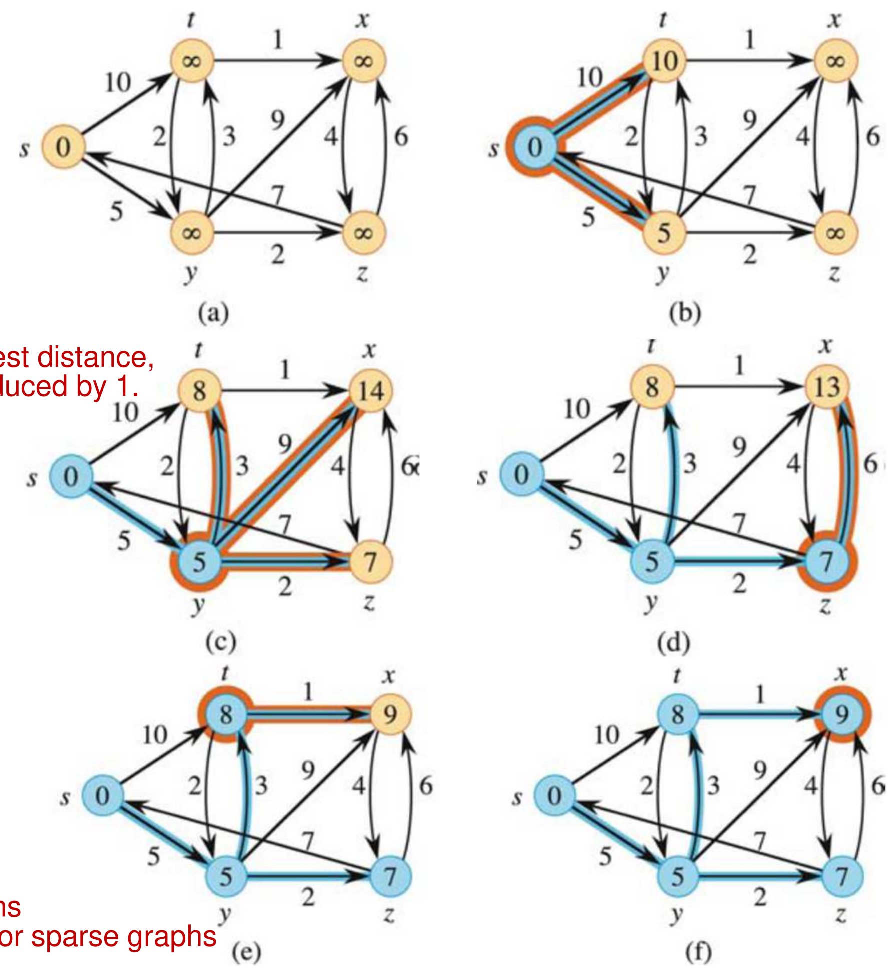
```
for each v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

Single-Source Shortest Paths (continued)

- **Dijkstra's Algorithm**
  - for weighted, directed graph, $G = (V, E)$, with **edge weights all non-negative**, i.e., $w(u, v) \geq 0$
  - maintaining a set S of vertices whose final shortest path weights from the source determined
  - selecting repeatedly the next vertex $u \in (V - S)$ with the minimum shortest-path estimate

```
DIJKSTRA(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  S = ∅
  Q = G.V          // i.e., insert all vertices into Q
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)    // choose the vertex with smallest distance,
                            // a greedy algorithm; Q is reduced by 1.
      S = S ∪ {u}
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
```

```
INIT-SINGLE-SOURCE(G, s)
  for each v ∈ G.V
      v.d = ∞
      v.π = NIL
  s.d = 0
```



Time complexity:

+ if Q is implemented in an array, we have complexity: $O(V^2 + E)$ as each EXTRACT-MIN takes $O(V)$.
+ if Q is implemented in a binary min-heap, where EXTRACT-MIN and DECREASE-KEY take $O(\lg V)$ each and there are up to E decrease operations in total, we have $O((V + E) \cdot \lg V)$; better for sparse graphs

Heap building takes $O(V)$.

All-Pairs Shortest Paths

§ Problem Overview

– for weighted, directed graph, $G = (V, E)$, with edge weight function of $\boldsymbol{w}$: $E \rightarrow R$, i.e., $W = (w_{ij})$
– $|V| = n$ vertices numbered 1, 2, 3, …, n
– create $n \times n$ matrix $D = (d_{ij})$ of shortest-path distances, with $d_{ij} = \delta(i, j)$ for all vertices i and j
– via Bellman-Ford algorithm to get complexity: $O(V^2 \bullet E)$ – as it invokes once per vertex, reaching $O(V^4)$ for a dense graph whose E equals $\Theta(V^2)$
– if no negative-weighted edges exist, Dijkstra's algorithm yields complexity of $O((V^2 + E) \bullet V)$ with a linear array, or of $O((V + E) \bullet \lg V \bullet V)$ with a binary heap

Alternatively, a more efficient algorithm exists.

For $W = (w_{ij})$ for a graph with n nodes, labelled 1 to n:

$$\underline{w_{ij}} = \begin{cases} 0 & \text{if } i = j \ , \\ \text{weight of } (i, j) & \text{if } i \neq j \, , (i, j) \in E \ , \\ \infty & \text{if } i \neq j \, , (i, j) \notin E \ . \end{cases}$$

All-Pairs Shortest Paths (continued)

§ Recursive Solution

Let $l_{ij}^{(m)}$ = weight of shortest path $i \rightsquigarrow j$ that contains $\leq m$ edges.

- $m = 0$

  $\Rightarrow$ there is a shortest path $i \rightsquigarrow j$ with $\leq m$ edges if and only if $i = j$ and distance = 0

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

- $m \geq 1$

  $$\Rightarrow l_{ij}^{(m)} = \min\left(l_{ij}^{(m-1)}, \min_{1 \leq k \leq n} \{l_{ik}^{(m-1)} + w_{kj}\}\right)$$ (k ranges over all possible predecessors of j)

  $$= \min_{1 \leq k \leq n} \{l_{ik}^{(m-1)} + w_{kj}\}$$ (since $w_{jj} = 0$ for all j).

  as $l_{ij}^{(m-1)} = l_{ij}^{(m-1)} + w_{jj}$, which is one element in the 2nd "min" operation.

  with $l_{ij}^{(1)} = w_{ij}$

All simple shortest paths contain $\leq n - 1$ edges

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

All-Pairs Shortest Paths (continued)

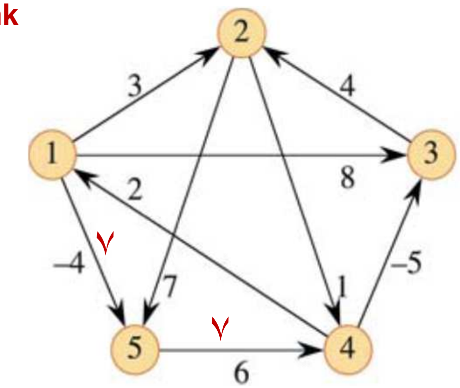
- **Compute Solution Bottom Up** (for a graph without negative-weight cycles)

Compute $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$, where an element of $L^{(k)}$ is denoted by $l_{ij}^{(k)}$

Start with $L^{(1)} = W$, since $l_{ij}^{(1)} = w_{ij}$.

Go from $L^{(m-1)}$ to $L^{(m)}$:

```
EXTEND(L, W, n)                    // extend each path by one link
  let L' = (l'_ij) be a new n × n matrix
  for i = 1 to n
      for j = 1 to n
          l'_ij = ∞
          for k = 1 to n
              l'_ij = min(l'_ij, l_ik + w_kj)
  return L'
```



For n = 5 in this case, we have $L^{(m)} = L^4$, for all $m \geq 4$ (due to "min").

Slow all-pairs shortest paths (APSP)

```
SLOW-APSP(W, n)
  L(1) = W
  for m = 2 to n − 1
      let L(m) be a new n × n matrix
      L(m) = EXTEND(L(m−1), W, n)
  return L(n−1)
```

obtained by individually adding Row 1 and Column 4 of $L^{(1)}$ to get the minimal value

$L^{(1)}_{14}$ $L^{(2)}_{14}$

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

$L^{(2)}_{53}$

$L^{(3)}_{13}$ $L^{(3)}_{25}$ $L^{(4)}_{12}$ $L^{(4)}_{35}$

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

$L^{(3)}_{52}$

All-Pairs Shortest Paths (continued)

- **Improving Running Time**
  - graph without negative-weight cycles, $L^{(m)} = L^{(n\text{-}1)}$, for all $m \geq n$-1
  - repeated squaring in $\lg(n-1)$ iterations to have: $2^{\lceil \lg(n-1) \rceil} \geq n$-1

FASTER-APSP(W, n) // all-pairs shortest paths

```
L^(1) = W
m = 1
while m < n − 1
    let L^(2m) be a new n × n matrix
    L^(2m) = EXTEND(L^(m), L^(m), n)
    m = 2m
return L^(m)
```

Time complexity: $\Theta(n^3 \lg n)$, since EXTEND takes $\Theta(n^3)$.

All-Pairs Shortest Paths (continued)

developed independently in 1962 by Robert Floyd and Stephen Warshall

§ Floyd-Warshall Algorithm (instead of adding **one link** per path iteratively, this one **adds one vertex** at a time)

- graph possibly with negative weight edges, but without negative-weight cycles
- via dynamic programming, to get complexity of $\Theta(n^3)$.
- iteratively adding one vertex at a time to compute shortest-path weights bottom up
- given minimum-weight path p (from v_i to v_j) with its intermediate nodes all ϵ $\{v_1, v_2, v_3, \dots, v_k\}$, where p may or may not contain v_k (added in latest iteration)

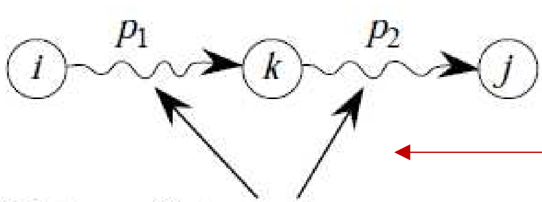
Let $d_{ij}^{(k)}$ = shortest-path weight of any path $i \leadsto j$ with all intermediate vertices in $\{1, 2, \dots, k\}$.

Consider a shortest path $i \overset{p}{\leadsto} j$ with all intermediate vertices in $\{1, 2, \dots, k\}$:

- If k is not an intermediate vertex, then all intermediate vertices of p are in $\{1, 2, \dots, k-1\}$.

  same shortest path as that one existing before adding Vertex k

- If k is an intermediate vertex:



involving "two shortest paths" existing before adding Vertex k

all intermediate vertices of $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ are all in $\{1, 2, \dots, k\text{-}1\}$

All-Pairs Shortest Paths (continued)

- **Recursive Solution**

$d_{ij}^{(k)}$ = shortest-path weight of any path $i \leadsto j$ with all intermediate vertices in $\{1, 2, \ldots, k\}$.

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

// no intermediate node, so any existing path contains 1 link
// minimum of the two cases explained previously

Finally, $D^{(n)} = \left(d_{ij}^{(n)}\right)$, after all vertexes are considered as possible intermediate nodes.

Compute in increasing order of k:

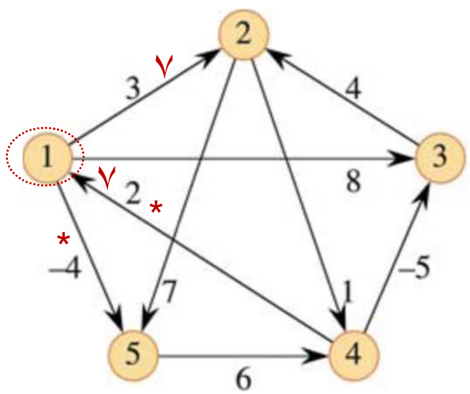
FLOYD-WARSHALL(W, n)

Time complexity: $\Theta(n^3)$.

```
D^(0) = W
for k = 1 to n          // for each additional vertex V_k, it has to check through all <i, j> vertex pairs
    let D^(k) = (d_ij^(k)) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D^(n)
```

All-Pairs Shortest Paths (continued)

- **Example**: matrices $D^{(k)}$ of Fig. 25.1 computed by Floyd-Warshall algorithm



add k^{th} element in Column 1 individually with all in Row 1 of $D^{(0)}$ to get Row k of $D^{(1)}$

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

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

$D^{(1)}_{42}$

entries potentially affected by adding V_1 include <x-1-2> <x-1-3>, <x-1-4>, and <x-1-5>

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

<x-2-4> <x-2-5>

entries potentially affected by adding V_2 include <x-2-1> <x-2-3>, <x-2-4>, and <x-2-5>

For V_4 on $D^{(3)}$ below, the 4th row (or column) Indicates all outgoing (or incoming) legs from (or to) V_4

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

$D^{(3)}_{42} = D^{(2)}_{43} + D^{(2)}_{32} = -5+4$

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

$D^{(5)}_{13} = D^{(4)}_{15} + D^{(4)}_{53}$

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

$D^{(5)}_{14} = D^{(4)}_{15} + D^{(4)}_{54}$

<x-5-2> <x-5-3> <x-5-4>

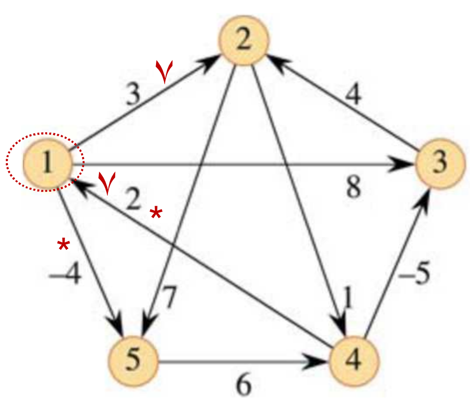
entries potentially affected by adding V_5 include <x-5-1>, <x-5-2>, <x-5-3>, and <x-5-4>

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

for $i = 1$ **to** n
  for $j = 1$ **to** n
    $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$

All-Pairs Shortest Paths (continued)

- **Constructing Shortest Paths**
  - compute predecessor matrix π for a sequence of $\pi^{(0)}$, $\pi^{(1)}$, $\cdots$ $\pi^{(n)} = \pi$ where $\pi_{i,j}^{(k)}$ denotes predecessor of j on its shortest path from i, with intermediate nodes $\in$ {1, 2, …,k}, with $\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)} , \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} . \end{cases}$



$$\begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

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

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

$\pi^{(1)}_{42}$ $\pi^{(1)}_{45}$

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

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

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

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

$\pi^{(5)}(1, 4)$: 1 → 5 → 4

$\pi^{(5)}(5, 2)$: 5 → 4 → 3 → 2

All-Pairs Shortest Paths (continued)

- **Transitive Closure**
  - transitive closure of G: $G^* = (V, E^*)$, with $E^* = \{(i, j) \mid \text{a path } i \rightarrow j \text{ exists in } G\}$
    (1) determine if a path exists from i to j
    (2) two possible methods:
    + by Floyd-Warshall algorithm after assigning each edge weight to “1”:
      if there is a path $i \rightarrow j$, then $d_{i,j} < n$; otherwise, $d_{i,j} = \infty$
    + quicker alternative via logical operations ∨ (OR) & ∧ (AND) to replace “min” & “+” so that a path $i \rightarrow j$ implies $d_{i,j}^{(n)} = 1$ following the recurrence below:

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i, j) \notin E , \\ 1 & \text{if } i = j \text{ or } (i, j) \in E , \end{cases}$$

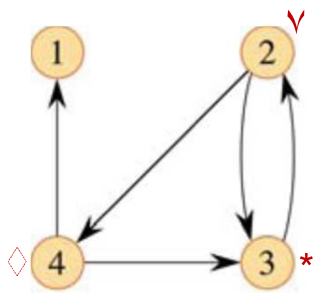
and for $k \geq 1$,

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

All-Pairs Shortest Paths (continued)

- **Transitive Closure Example**
  - quicker Floyd-Warshall algorithm alternative:

    via logical operations ∨ (OR) & ∧ (AND) to replace "min" & "+" operations, respectively
  - we have: $t_{ij}^{(k)} = \begin{cases} 1 & \text{if there is path } i \leadsto j \text{ with all intermediate vertices in } \{1, 2, \ldots, k\} \\ 0 & \text{otherwise .} \end{cases}$



"and" k^{th} element in Column 1 individually with all in Row 1 of $T^{(0)}$ to get Row k of $D^{T1)}$

$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1^{\vee} \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$T^{(2)}_{34} = T^{(1)}_{32} \wedge T^{(1)}_{24}$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1^{*} & 1 & 1 \\ 1 & \boxed{1} & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

$T^{(3)}_{42} = T^{(2)}_{43} \wedge T^{(2)}_{32}$

$T^{(4)}_{21} = T^{(3)}_{24} \wedge T^{(3)}_{41}$

$T^{(4)}_{31} = T^{(3)}_{34} \wedge T^{(3)}_{41}$

TRANSITIVE-CLOSURE(G, n) // same time complexity of $\Theta(n^3)$

```
n = |G.V|
let T^(0) = (t_ij^(0)) be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        if i == j or (i, j) ∈ G.E
            t_ij^(0) = 1
        else t_ij^(0) = 0
for k = 1 to n
    let T^(k) = (t_ij^(k)) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            t_ij^(k) = t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
return T^(n)
```

Maximum Flow

§ Flow Networks and Flows

– directed graph, $G = (V, E)$, with a <u>nonnegative capacity</u> $c(u, v) \geq 0$, $\forall (u, v) \in E$

– if $(u, v) \in E$ then $(v, u) \notin E$ (i.e., <u>no anti-parallel edges</u>)

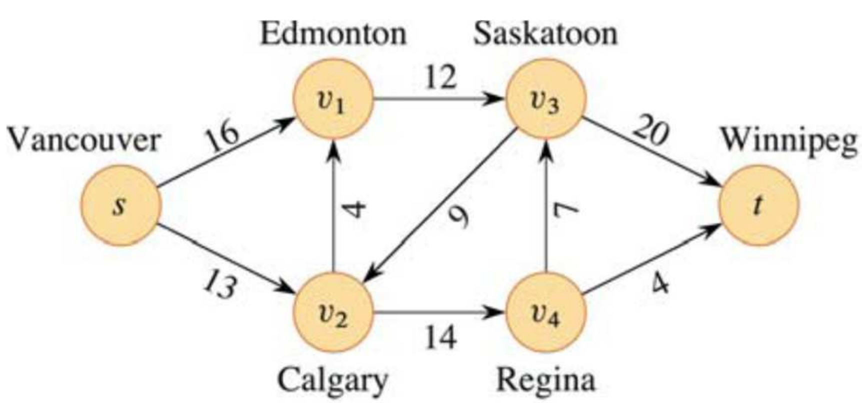
– two distinguished vertices: ***s*** (source) and ***t*** (sink)

– a flow in G satisfies:

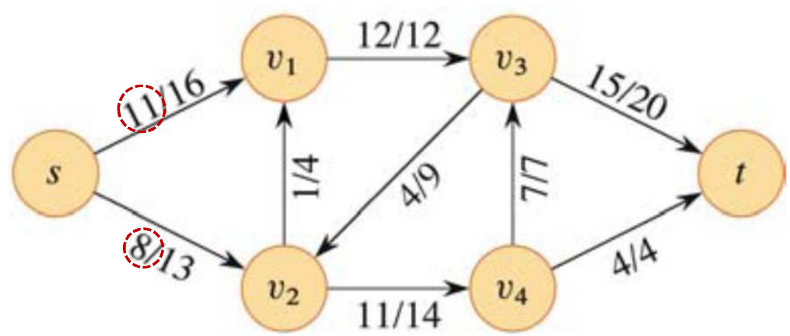
- ***Capacity constraint:*** For all $u, v \in V, 0 \leq f(u, v) \leq c(u, v)$,
- ***Flow conservation:*** For all $u \in V - \{s, t\}$, $\underbrace{\sum_{v \in V} f(v, u)}_{\text{flow into } u} = \underbrace{\sum_{v \in V} f(u, v)}_{\text{flow out of } u}$

Equivalently, $\overset{\text{out from } u}{\sum_{v \in V} f(u, v)} - \overset{\text{in to } u}{\sum_{v \in V} f(v, u)} = 0.$

$$\begin{aligned} \textit{Value of flow } f &= |f| \\ &= \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) \\ &= \text{flow out of source} - \text{flow into source} \end{aligned}$$



(a) Flow network with link capacities shown

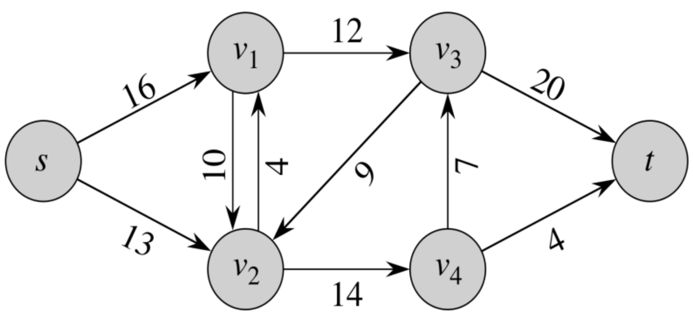


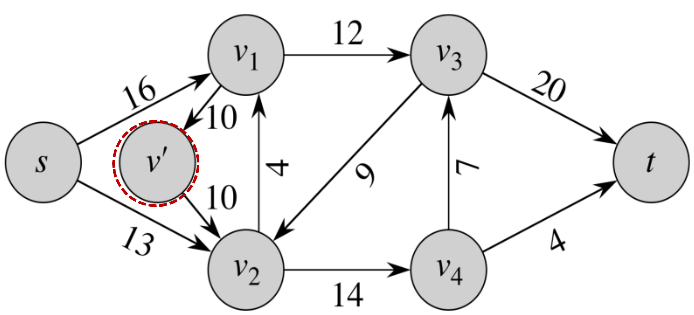
(b) Flow f in the network with $|f|$ = 11+8 = 19

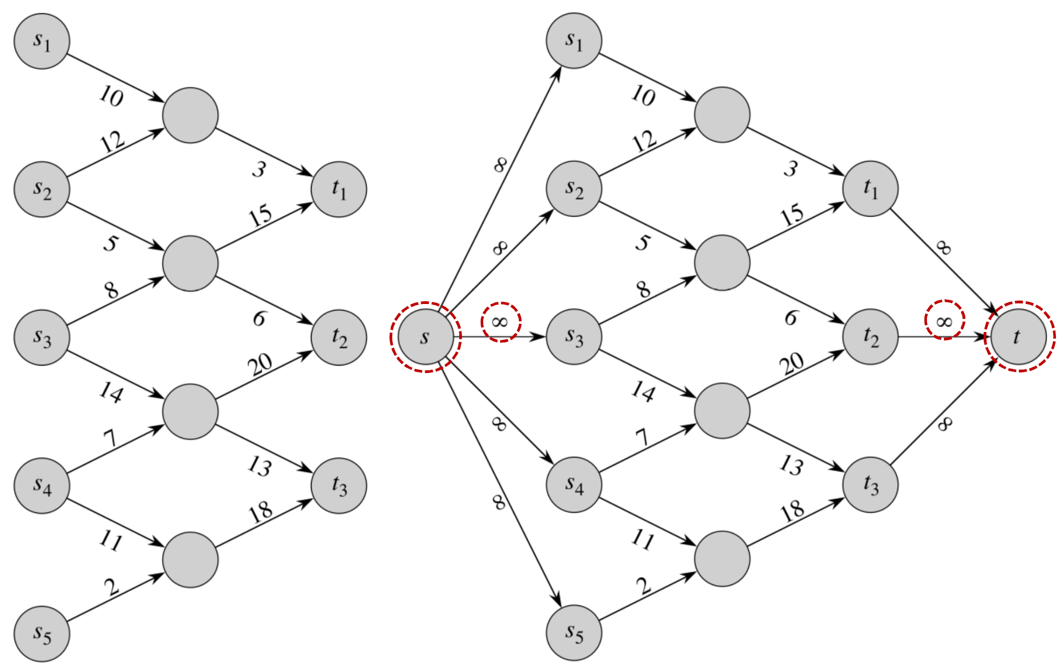
Maximum Flow (continued)

§ Maximum Flow

- Given G, s, t, and c, find a flow whose value is maximum.
- Replacing an antiparallel edge in G with two edges entering and exiting from an extra vertex v'
- Converting a network with multiple sources/destinations to equivalent one with single source and single destination (in (b) below)







(b)

Maximum Flow (continued)

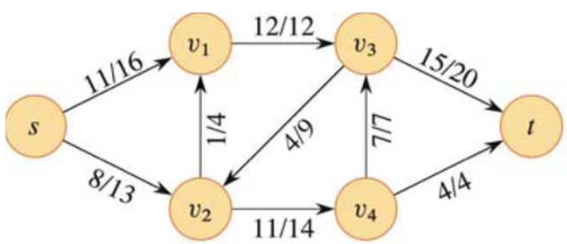
§ Ford-Fulkerson Method (by Lester Ford and Delbert Fulkerson in 1956)

- iteratively increase the flow value, after initializing $f(u, v) = 0$, $\forall u, v \in V$
- for a given flow f in G , determine an augmenting path in associated residual network G_f
- maximum flow obtained when no more augmenting path exists
- residual network G_f with residual capacity $c_f(u, v)$ given by

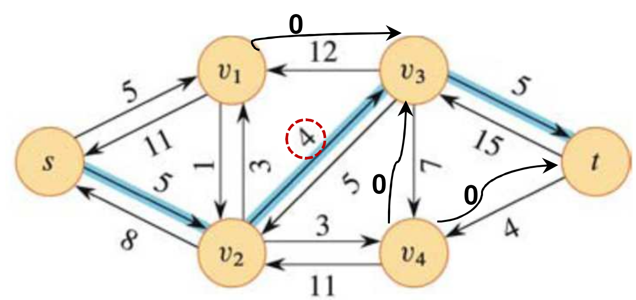
$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \,, \\ f(v,u) & \text{if } (v,u) \in E \,, \\ 0 & \text{otherwise (i.e., } (u,v), (v,u) \notin E) \,. \end{cases}$$

// for edge (v, u) in G, an extra edge (u, v) is added with its capacity $c_f(u, v)$ equal to $f(v, u)$

(a) Network G with flow f and capacity c shown



(b) residual network G_f with augmenting path p marked



FORD-FULKERSON-METHOD(G, s, t)

```
initialize flow f to 0
while there exists an augmenting path p in the residual network G_f
    augment flow f along p
return f
```

Maximum Flow (continued)

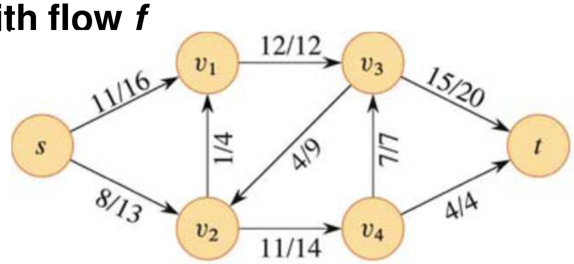
§ Ford-Fulkerson Method

- augmenting flow f via f' (along augmenting path p) by the amount of residual capacity $c_f(p)$
- augmenting path p is a simple path from s to t in residual network G_f

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E, \\ 0 & \text{otherwise} \end{cases}$$

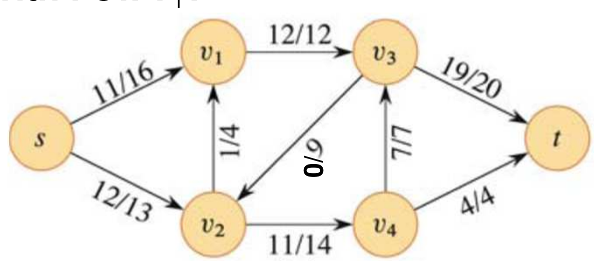
for all $u, v \in V$.

G* with flow *f



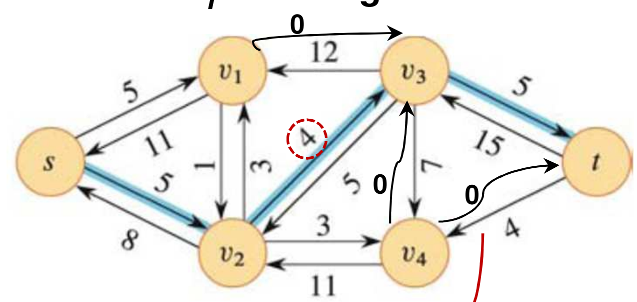
(a)

***G* with flow $f \uparrow f'$**



(c)

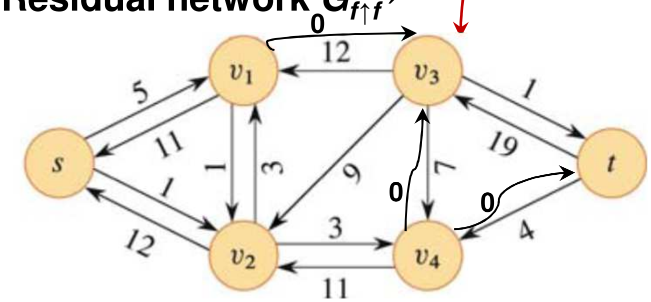
Residual network G_f with augmented flow f' shown



(b)

directly obtained

Residual network $G_{f \uparrow f'}$



Maximum Flow (continued)

§ Cuts of Flow Networks

- cut (S, T) of flow network G = (V, E) <u>fragments V into S and T</u> = $V - S$ such that $s \in S$ and $t \in T$
- <u>capacity</u> of cut (S, T) denoted by <u>C(S, T)</u>, equal to $\sum_{u \in S} \sum_{v \in T} c(u, v)$, <u>counting only from S to T</u>
- net flow across cut (S, T), f(S, T), equals

$$\sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{v \in T} \sum_{u \in S} f(v, u)$$

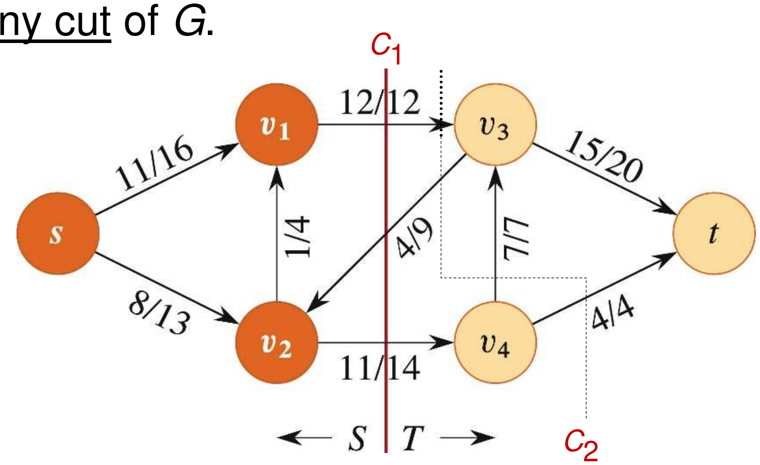
<u>Corollary</u>

Any flow f in G is <u>bounded above</u> by the capacity of <u>any cut</u> of G.

<u>Theorem</u> (max-flow min-cut)

A flow f in G has following equivalences:

1. f is a maximum flow.
2. G_f has no augmenting path.
3. $|f| = c(S, T)$ for some cut (S, T).



$C_1(S, T) = c(v_1, v_3) + c(v_2, v_4)$ = 12+14 = 26

Associated $f(S, T) = f(v_1, v_3) + f(v_2, v_4)$
$- f(v_3, v_2)$ = 12+11-4 = 19

Maximum Flow (continued)

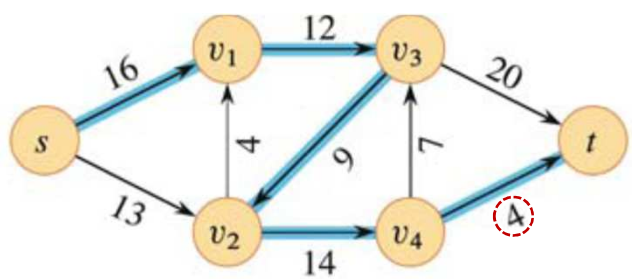
§ Basic Ford-Fulkerson Algorithm

– replacing flow *f* by $f \uparrow f_p$ across augmenting path *p* <u>repeated</u>

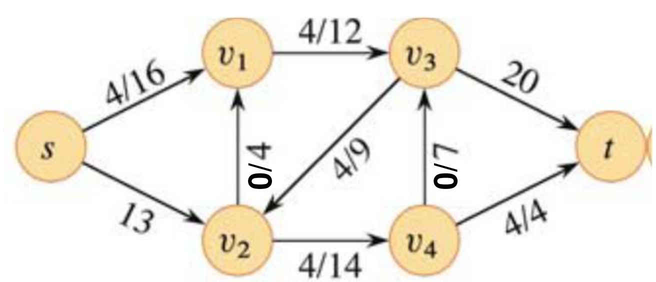
```
FORD-FULKERSON(G, s, t)
for each edge (u, v) ∈ G.E
    (u, v).f = 0                        // there exists one link not involved in the flow obtained so far
while there exists a path p from s to t in the residual network G_f
    c_f(p) = min {c_f(u, v) : (u, v) is in p}
    for each edge (u, v) in p
        if (u, v) ∈ G.E
            (u, v).f = (u, v).f + c_f(p)
        else (v, u).f = (v, u).f − c_f(p)
```

● Example Ford-Fulkerson Algorithm

(a) Input network *G* (with existing edges and an augmenting path marked)



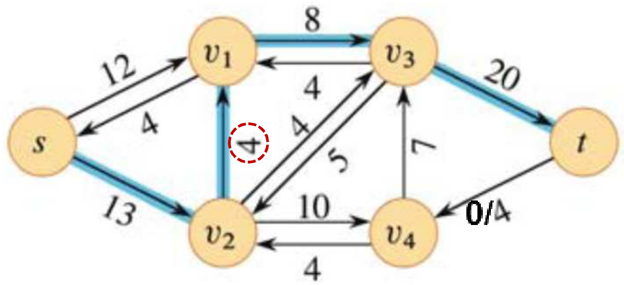
G with new flow *f* augmented by f_p



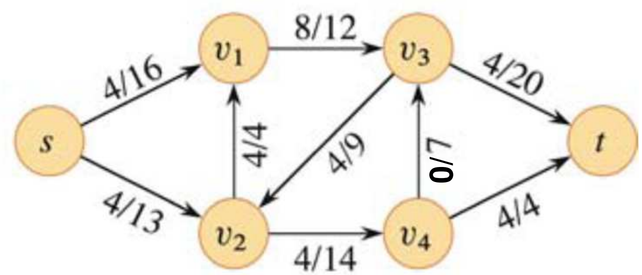
Maximum Flow (continued)

- **Example Ford-Fulkerson Algorithm**

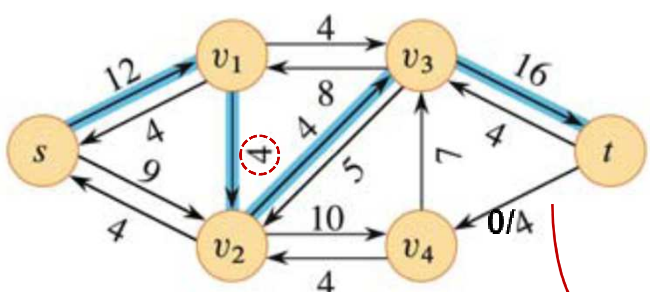
(b) Residual network G_f (with an augmented path marked)



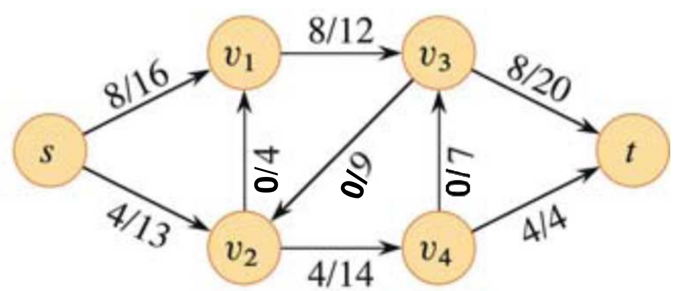
G with new flow *f* augmented by f_p in (b)



(c) Residual network G_f (with an augmenting path marked)

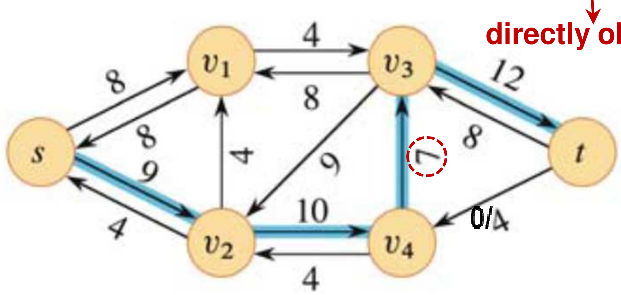


G with new flow *f* augmented by f_p in (c)

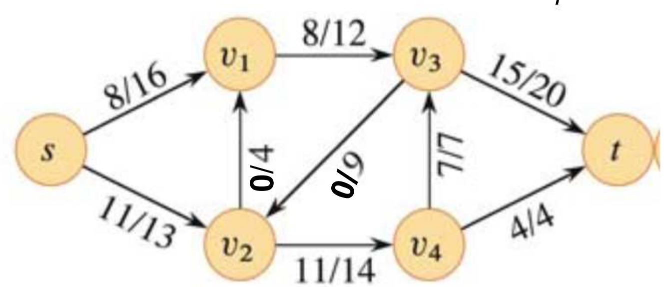


(d) Residual network G_f (with an augmenting path marked)

directly obtained



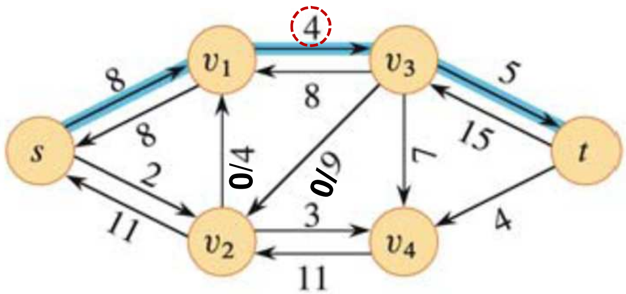
G with new flow *f* augmented by f_p in (d)



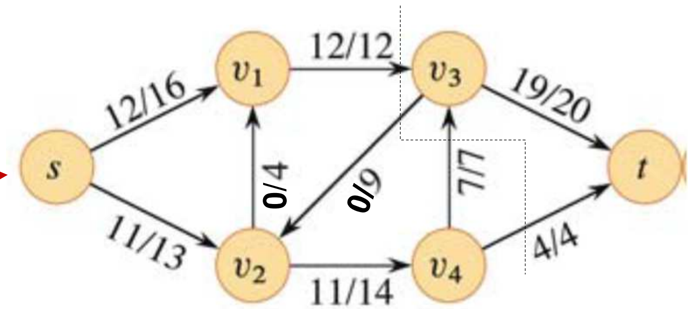
Maximum Flow (continued)

- **Example Ford-Fulkerson Algorithm**

(e) Residual network G_f (with augmenting path marked)

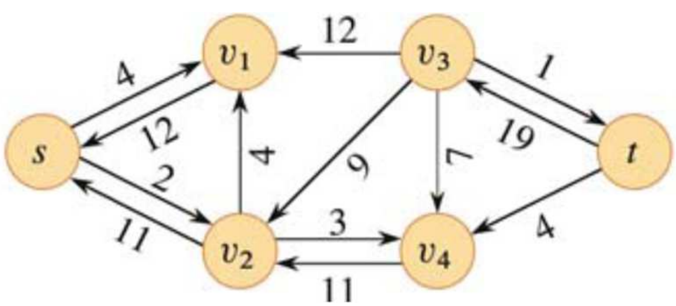


G with new flow f augmented by f_p in (e)



final maximum flow result

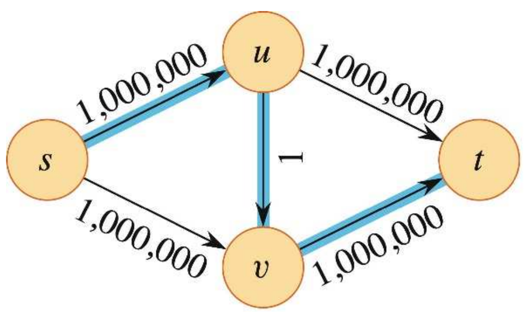
(f) Residual network G_f (**no** <u>augmentation</u> possible)



Maximum Flow (continued)

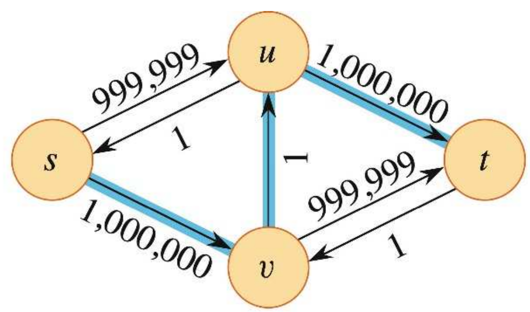
- **Analysis**
  - **Basic** Ford-Fulkerson algorithm takes $O(E \cdot |f^*|)$, where f^* denotes the maximum flow, as each iteration increases the flow amount by at least 1 and the time for finding an augmenting path in the <u>residual network</u> equals $O(V + E') = O(E)$

Note: breadth-first search takes $O(V + E)$.



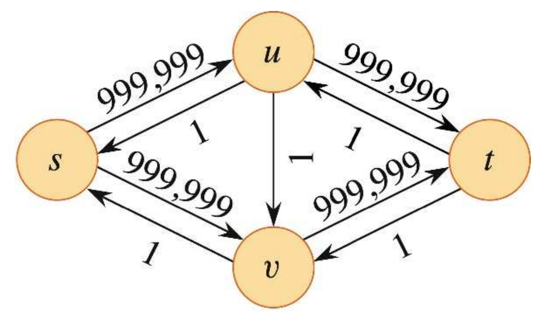
(a)

The initial flow network, which is the same as its residual network, as there is no flow value for any edge yet.



(b)

The residual network after one flow augmentation via the augmenting path marked in (a).



(c)

The residual network after another flow augmentation via the augmenting path marked in (b).

Maximum Flow (continued)

- **Analysis**
  - Edmonds-Karp algorithm runs in $O(V \cdot E^2)$, using breadth-first search to find a desirable augmenting path (with shortest distance) in the residual network from s to t with edges being unit-weighted, based on Theorem below.

Lemma

Given the Edmonds-Karp algorithm run on flow network $G = (V, E)$ with source s and sink t, the shortest-path distance in the residual network G_f for any vertex $v \in V-\{s, t\}$, $\delta_f(s, v)$, increases monotonically with each flow augmentation, i.e., $\delta_f(s, v) \leq \delta_{f'}(s, v)$.

Theorem

The Edmonds-Karp algorithm performs $O(V \cdot E)$ augmentations on flow network $G = (V, E)$.

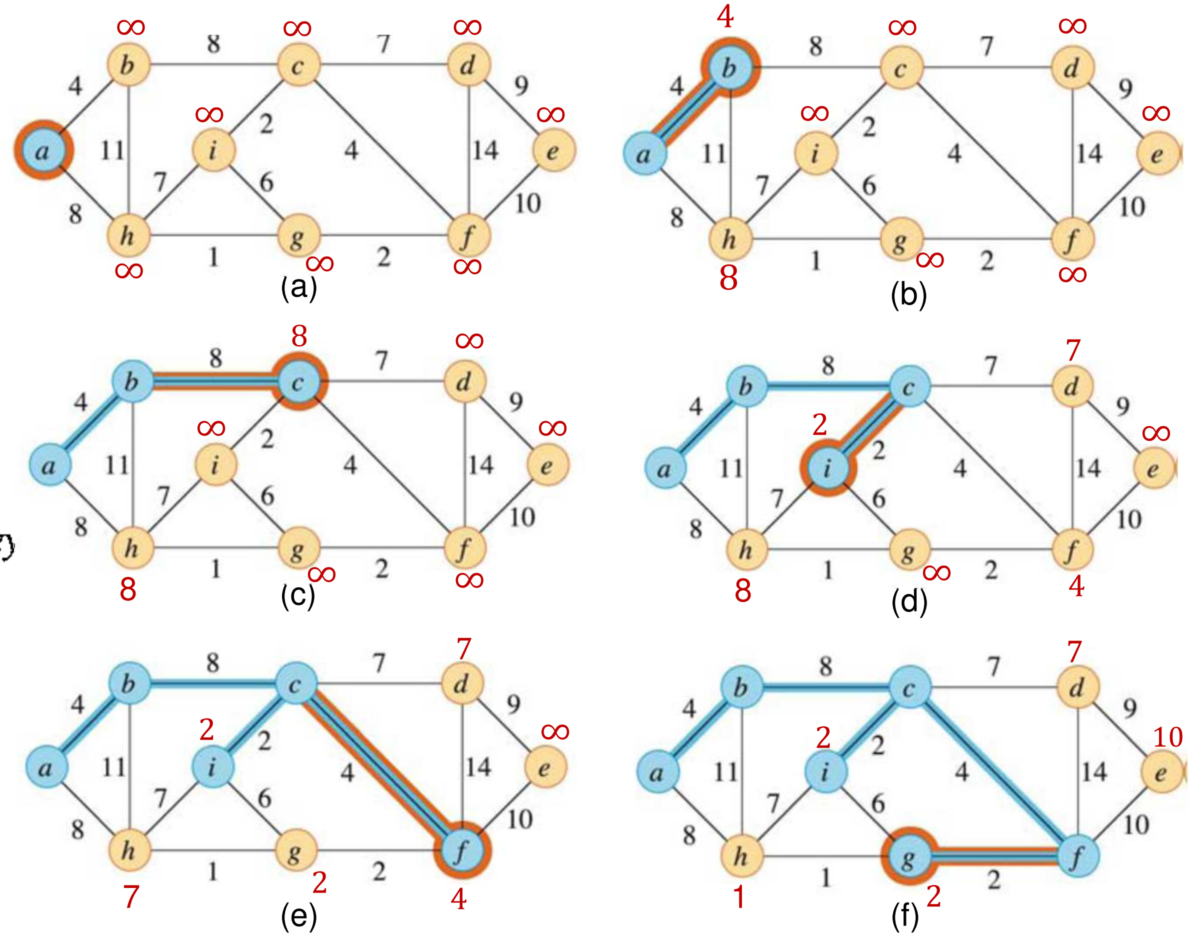
(Note: the proof is based on the fact that each edge may become *critical* up to $V/2$ times, where an edge is critical if it is on an augmentation path with the lowest capacity among all constituent edges of the path.)

Note: as breadth-first search in $G(V, E)$ takes $O(E)$ to find an augmenting path, the complexity of Edmonds-Karp algorithm under the Ford-Fulkerson method equals $O(V \cdot E^2)$.

Minimum Spanning Trees (continued)

- **Prim's algorithm for MST** (minimum spanning trees)
  - *greedy algorithm* by selecting examined edge with smallest weight to add to the tree
  - *v.key* denotes the minimum weight from *v* to the tree established so far, with *v.key* values of all vertices (other than the root) initialized to ∞

```
MST-PRIM(G, w, r)
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for one v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```

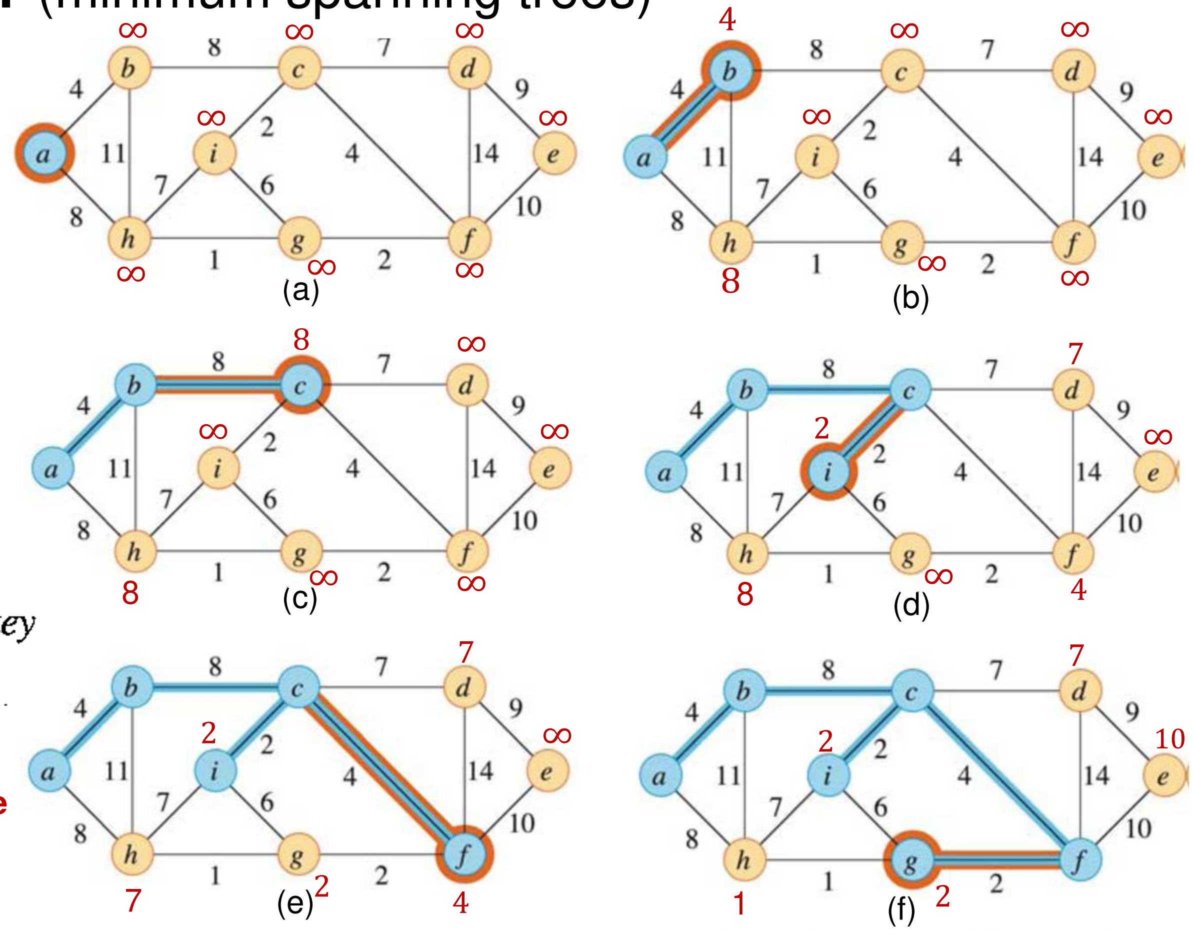


Minimum Spanning Trees (continued)

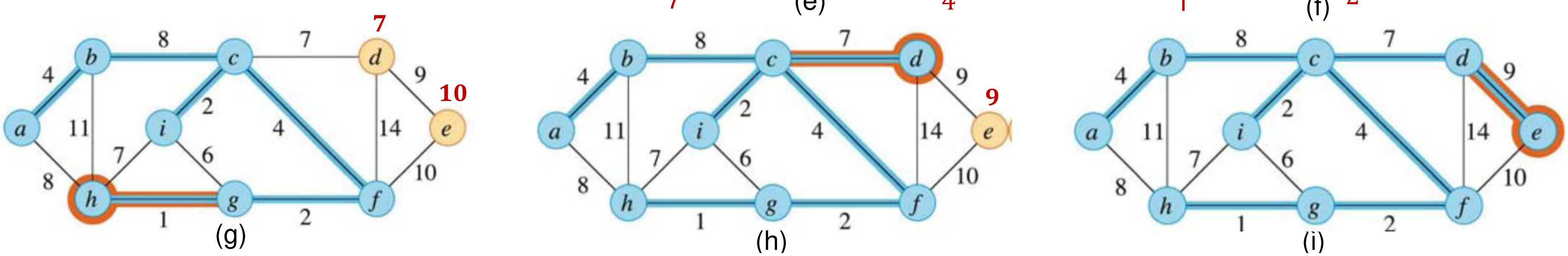
- **Prim's algorithm for MST** (minimum spanning trees)

```
MST-PRIM(G, w, r)
    for each u ∈ G.V
        u.key = ∞
        u.π = NIL
    r.key = 0
    Q = G.V
    while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        for each v ∈ G.Adj[u]
            if v ∈ Q and w(u, v) < v.key
                v.π = u
                v.key = w(u, v)
```

vertices in *G.V–Q* already included in the tree



(a) (b) (c) (d) (e) (f)



(g) (h) (i)